T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

KARMA ÖZNİTELİK KULLANARAK YAZILIM DAVRANIŞLARININ MODELLENMESİ VE TESPİTİ

MERT NAR
YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

GEBZE
2020

T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

KARMA ÖZNİTELİK KULLANARAK YAZILIM DAVRANIŞLARININ MODELLENMESİ VE TESPİTİ

MERT NAR
YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMANI
PROF. DR. İBRAHİM SOĞUKPINAR

GEBZE
2020

T.R.

GEBZE TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

SOFTWARE BEHAVIOR MODELING AND DETECTION BY USING HYBRID FEATURES

MERT NAR

A THESIS SUBMITTED FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER ENGINEERING

THESIS SUPERVISOR

PROF. DR. İBRAHİM SOĞUKPINAR

GEBZE

2020

GEBZE TEKNİK ÜNİVERSİTESİ	**YÜKSEK LİSANS JÜRİ ONAY FORMU**

GTÜ Fen Bilimleri Enstitüsü Yönetim Kurulu'nun 15/01/2020 tarih ve 2020/04 sayılı kararıyla oluşturulan jüri tarafından 23/01/2020 tarihinde tez savunma sınavı yapılan Mert NAR'ın tez çalışması Bilgisayar Mühendisliği Anabilim Dalında YÜKSEK LİSANS tezi olarak kabul edilmiştir.

JÜRİ

ÜYE (TEZ DANIŞMANI)	: Prof. Dr. İbrahim SOĞUKPINAR
ÜYE	:Doç Dr. Mehmet GÖKTÜRK
ÜYE	:Doç Dr. Ali Gökhan YAVUZ

ONAY

Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun

..... tarih ve ..... sayılı kararı.

GEBZE TEKNİK ÜNİVERSİTESİ	**YÜKSEK LİSANS JÜRİ ONAY FORMU**

GTÜ ……Enstitüsü Yönetim Kurulu'nun …..…/…..…/..…..… tarih ve ..……/……..… sayılı kararıyla oluşturulan jüri tarafından …..…/…..…/..…..… tarihinde tez savunma sınavı yapılan ……'ın tez çalışması ……Anabilim Dalında YÜKSEK LİSANS tezi olarak kabul edilmiştir.

JÜRİ

ÜYE
(TEZ DANIŞMANI) :

ÜYE :

ÜYE :

ONAY

Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun

..…/……/…… tarih ve ……/…..... sayılı kararı.

ÖZET

Zararlı yazılımlar sahip oldukları yeteneklerden ötürü bilgisayar ve sistemlere büyük tehlike oluşturmaktadır. Etkin tespit sistemlerinin gelişmesinden aynı şekilde etkilenerek daha tehlikeli ve donanımlı hale gelmektedirler. Bu kedi-fare oyununda savunmacıların en etkin silahı analiz araçlarıdır. Otomatik bir tespit sistemi geliştirmek için de zararlı yazılımlar iyi analiz edilmeli ve gelişim meyilleri doğru tespit edilmelidir. Zararlı yazılımların çalıştığı bilgisayarda yarattığı etkiler ve kod yapısı ayrıntılı incelenmeli ve öyle önlem alınmalıdır. Analiz çalışmaları en genel manada 2'ye ayrılmaktadır; durağan ve dinamik analiz. Durağan analizin temelinde yazılımın kod ve dosya yapısını, çalıştırmadan incelemektir. Dinamik analizin altında yatan felsefe ise zararlının çalışma anı davranışlarını gözlemlemek ve yarattığı etkileri ortaya çıkarmaktır. Her iki analiz yaklaşımının üstünlükleri ve zayıflıkları vardır. Birbirlerinin yerlerine koyulmasının imkânsız olmasının sebebi, her analiz tekniğinin zararlı yazılımlara farklı bir açıdan inceleme yeteneği getirmesidir. Buna karşıt olarak zararlı yazılımlar analiz tekniklerinden kaçınmak için çeşitli taktikler geliştirmişlerdir. Durağan ve dinamik analiz yöntemlerini ve çalışma tekniklerini iyi bilen saldırganlar bu yöntemleri atlatmak ya da bu analizler altında gerçek niyetlerini gizlemek adına zararlı yazılım oluştururken uyguladıkları taktikler vardır.

Zararlı yazılımları tespit etmek için, var olan yöntemleri tekrarlamak yerine, onları engelleyici taktikleri ve teknikleri iyi bilmek ve önlem almak gerekmektedir. Bu çalışmada zararlı yazılım yapısı ve davranış eğilimleri incelenmiştir. Zararlı yazılımların uyguladıkları saklanma, gizlenme ve analizlerden kaçınma taktikleri ayrıntılı olarak araştırılmış ve ele alınmıştır. Bu bilgi ve tecrübeler ışığında zararlı yazılım tespit sistemi önerilmiştir. Önerilen tespit sistemi, zararlı yazılımın hem davranış hem kod yapısı bilgisini kullanarak Markov zinciri yöntemi ile istatistiksel bir anlam çıkarmaktadır. Daha sonra derin öğrenme teknikleri ile temellendirilmiş model melez veri kaynağı ile eğitilmiş ve tespit ortamı hazırlanmıştır. Yaptığımız testler sonucunda önerilen tespit yöntemi %96,8'lik doğruluk göstermiştir.

Anahtar Kelimeler: Zararlı Yazılım, Markov Zinciri, Derin Öğrenme, Evrişimli Sinir Ağları, Dinamik analiz, Durağan analiz, Yazılım Davranış Analizi.

SUMMARY

Malware poses a great danger to computers and systems due to their capabilities. They are also affected by the development of effective detection systems and become more dangerous and equipped. Defenders are the most effective weapon analysis tools in this cat and mouse game. In order to develop an automated detection system, malware should be well analyzed, and the development tendencies should be detected correctly. The effects of malicious software on the computer and code structure should be examined in detail and such precautions should be taken. Analysis studies are divided into 2 in the most general sense; static and dynamic analysis. The basis of static analysis is to examine the code and file structure of the software without running it. The underlying philosophy of dynamic analysis is to observe the working moment behavior of the pest and to reveal its effects. Both analysis approaches have advantages and weaknesses. The reason why it is impossible to replace each other is that each analysis technique brings the ability to examine malware from a different angle. In contrast, malware has developed several tactics to avoid analysis techniques. Attackers who are familiar with static and dynamic methods of analysis and work techniques have tactics to circumvent these methods or to create malware to conceal their true intentions.

To detect malware, rather than repeating the existing methods, tactics and techniques to prevent them should be well known and taken precautions. In this study, malware structure and behavioral trends are examined. The hiding, hiding, and avoidance analysis tactics applied by malware have been investigated and discussed in detail. In the light of this knowledge and experience, malware detection system has been proposed. The proposed detection system makes a statistical meaning with the Markov chain method using both the behavior and code structure information of the malware. Then, model based on deep learning techniques was trained with hybrid data source and detection environment was prepared. As a result of our tests, the recommended detection method showed an accuracy of 96.8%.

Key words: Malware, Markov Chain, Deep learning, Convolutional Neural Network, Dynamic Analysis, Static Analysis, Software Behavior Analysis.

TEŞEKKÜR

Başta, yüksek lisans eğitimimde ve akademik hayatımda desteğini ve yardımlarını hiçbir zaman esirgemeyip bilgisi ile bu çalışmanın oluşmasının yolunu açan danışmanım Prof. Dr. İbrahim SOĞUKPINAR'a,

Bütün çalışmam boyunca yanımda olan, bilgi ve tecrübelerini benimle paylaşan değerli hocam Dr. Arzu KAKIŞIM'a,

Çalışmamın başında bilgi ve tecrübesini bana aktararak çalışmalarıma hız kazandıran sayın Necmettin ÇARKACI'ya,

ve vermiş olduğu destekten dolayı sevgili eşim Nazlı NAR'a en içten teşekkürlerimi sunarım.

İÇİNDEKİLER

SİMGELER ve KISALTMALAR DİZİNİ

Simgeler ve Kısaltmalar		Açıklamalar
API	:	Uygulama Programlama Arayüzü
DLL	:	Dinamik Bağlanan Kütüphane
OpCode	:	İşlem Kodu
RAM	:	Rastgele Erişimli Bellek
CPU	:	Merkezi İşlem Birimi
LIFO	:	Son Giren İlk Çıkar
GTÜ	:	Gebze Teknik Üniversitesi
ELF	:	Çalıştırılabilir Bağlanabilir Dosya Formatı
DDoS	:	Dağıtık Hizmet Engelleme
FTP	:	Dosya nakil protokolü
HTTP	:	Hiper metin transfer aktarım protokolü
ISA	:	Komutlar Kümesi Mimarisi
DFT	:	Veri akışı izleme
DFA	:	Dinamik Renk tonu Analizi
CFG	:	Kontrol Akış Grafiği
HMM	:	Gizli Markov Modeli
SVM	:	Destek Vektör Makinesi
KNN	:	K En Yakın Komşu
PPM	:	Parçalı Eşleşme İle Öngörü
PCA	:	Temel Bileşen Analizi
CNN	:	Evrişimsel Sinir Ağı
MLP	:	Çok Katmanlı Perseptron
TP	:	Gerçek Pozitif
FN	:	Yanlış Negatif
TPR	:	Gerçek Pozitif Oranı
FPR	:	Yanlış Pozitif Oranı
IDT	:	Kesme Tanımlayıcı Tablosu
IDTR		Kesme Tanımlayıcı Tablo Kayıtçısı
PE	:	Zaman Uzayı Sonlu Farklar

ŞEKİLLER DİZİNİ

TABLOLAR DİZİNİ

1. GİRİŞ

Kötücüllerin saklanma ve atlatma yöntemleri günden güne değişmekte ve literatürde var olan tespit yöntemleri güncelliğini kaybetmektedir. Bulunan her yeni tespit yöntemine karşılık yeni saklanma yöntemleri geliştirilmekte ve otomatik kötücül yazılım tespiti saf dışı bırakılmaya çalışılmaktadır [1]. Bunun yanında kötücül davranışlar iyi huylu diye düşünülen yazılımların yeni güncellemelerine ya da korsan hallerine enjekte edildiği gözlenmektedir. MalwarebytesLabs [2] tarafından yayınlanan rapora göre, 2018 yılında yaklaşık 750,3 milyon kötü amaçlı yazılım tespit edilmiştir. Ayrıca, bir başka rapora göre [3] her gün 350.000'in üzerinde yeni kötü amaçlı programın internete salındığı bildirilmiştir. Bu raporlar göz önünde bulundurulup genel bir değerlendirme yapacak olursak, saldırganlarla savunmacılar arasında bir silahlanma yarışı olduğu söylenebilir. Zararlı yazılım yazarlarının, zararlı yazılım yapım kitleri, kaçınma teknikleri, mevcut zararlı yazılım kaynak kodları vb. gibi silahları vardır [4]. Dolayısıyla, kötü amaçlı yazılım saldırılarının yayılmasını ve yaygınlaşmasını, imza tabanlı tespit sistemleri kullanarak kovuşturmak ve engellemek mümkün değildir. İçerisinde bulunduğumuz bu yarışta, mücadeleye devam edebilmek için savunucuların zararlı yazılım tespit teknikleri için geleneksel tespit sistemi yöntemlerini tek yol olarak kullanmaktan vazgeçmesi gerekir. Zararlı yazılımların imza eşleştirme yöntemi ile tespit edilmesinin yerine niteleyici ortak örüntü bularak tespit edilmesi gerekir. Bu nedenle, literatürde benzer örüntü arayan pek çok zararlı yazılım tespit yöntemi önerilmiştir [4, 5, 6, 7].

Benzer örüntü bulmak hususunda düşünülecek ilk konu zararlı yazılım analizidir [8]. Zararlı yazılım analizi ile, işlem kodları (OpCodes), uygulama programlama arabirim (API) fonksiyonlarının çağrımı, davranış bilgileri, başlık bilgileri, çalışma süresi, üzerinde çalıştığı bilgisayardaki etkiler gibi öznitelikler çıkarılır. Yazılım analizi genel olarak yazılım davranışı ve kod yapısı hakkında bilgi vermektedir. Bu bilgiyi otomatize bir şekilde çıkarmak ve öğrenmek için analiz ile öznitelikler çıkarılır. Öznitelikler zararlı yazılımın ne yaptığını ve nasıl kullanıldığını açıklar. Kötü amaçlı yazılım analizi, statik ve dinamik olmak üzere iki yaklaşımla gerçekleştirilir. Statik teknikler, ikili kodu çalıştırmadan analiz eder; dinamik

teknikler ise zararlı yazılım kontrollü bir ortamda çalışırken bilgisayarda yarattığı etkiyi gözlemleyerek davranışlarını incelemeye dayanır [9]. Bu yaklaşımlardan herhangi biri tek başına uygulanması zararlı yazılımlar hakkında bilgi kaybına sebebiyet verebilir [10]. Çünkü, farklı öznitelik tipleri yazılımın farklı bir yönünü göstermektedir ve belki o bakış açısı zararlı yazılımı açık eder. Yani, her iki analiz türünde de bazı avantajlar ve dezavantajlar vardır [11].

Silahlanma yarışının diğer tarafından bakıldığında, saldırganların ve zararlı yazılım yazarlarının analizden kaçınmak için kullandıkları yöntemler vardır. Durağan analizden kaçınmak için en popüler ve güçlü yöntem kendi kendini değiştirme yöntemidir. Zararlı yazılım kendi kodunda yapısal değişiklikler yaparak farklı bir imza ve kod yapısına sahip benzerlerini üretebilir [12]. Böylece durağan analiz yapan tespit sistemleri ve geleneksel imza tabanlı tespit yöntemleri atlatılabilir. Diğer taraftan, bazı yöntemler ile (hedefleme, tersine ayar testi, erteleme, tetikleme bazlı kodlama ve dosyasız (AVT) saldırı gibi [8]) dinamik analizden de kaçmak mümkündür. Analizden kaçınmak bir yana, Demetrio ve diğerleri [13] belirttiği gibi derin öğrenme teknikleri uygulanmış ileri düzey zararlı yazılım tespit sistemlerini aldatmak da mümkündür. Bu aldatma öznitelik mühendisliği yapılmadığı durumlarda gerçekleşir. İşlenmemiş ham veri ile eğitilmiş derin öğrenme modelleri, literatürde ortaya çıkarılan taktikler ile kandırılabileceği gösterilmiştir [14]. Bu yüzden zararlı yazılımlar ve aileleri dikkatlice incelenmelidir.

Kötü amaçlı yazılım analizi bir kedi-fare oyunu olduğu unutulmamalıdır. Yeni kötü amaçlı yazılım analiz teknikleri geliştirildikçe, yeni saklanma, kaçınma ya da zararlı aktiviteler geliştiği gözlemlenmiştir. Bir kötü amaçlı yazılım tespit sisteminin olarak başarılı olması için, bu teknikleri tanıması, anlayabilmesi ve yenebilmesi ve kötü amaçlı yazılım analizi alanındaki değişikliklere yanıt verebilmesi gerekir. Bu tez çalışmasının amacı, zararlı yazılımları enine boyuna incelemek ve davranışlarını modelleyerek tespit ortamı yaratmaktır. Bu kapsamda saldırgan ve savunmacı bakış açısından zararlı yazılımlar incelenmiştir. Zararlı yazılım türleri ve davranışları ayrıntılı olarak analiz edilmiş, akabinde literatürde var olan tespit yöntemleri ve analiz çalışmaları çalışma yapıları ve başarım oranları itibariyle tetkik edilmiştir. Bulguların ışığında özgün bir zararlı yazılım tespit sistemi önerilmiştir.

2. KURAMSAL TEMELLER ve İLGİLİ ÇALIŞMALAR

Günümüzde Zararlı yazılımlar büyük tehlikeler oluşturmaktadır. Zararlı yazılım tehdidine karşı koymak ve zarar risklerini en düşük boyuta indirmek için onları iyi tanımak gerekir. Bu tanıma ve tanımlamanın ilk adımı zararlı yazılım analizidir. Zararlı yazılım analizi doğru ve etkili yapmak ve çıkan sonuçları iyi yorumlamak için bazı temel bilgilere ihtiyaç duyulmaktadır. Bu bölümde zararlı yazılımlarla mücadele etmek ve onları anlamak için temel anlamda gerekli kuramsal bilgiler verilecektir.

2.1. Windows Çalıştırılabilir Dosyalar

Zararlı yazılımları analiz etmeden önce Windows ortamında ikili dosyaların nasıl çalıştığını bilmek gerekmektedir. Bu yüzden önce çalışma ortamı tanımlanmalı ve bilgisayarın nasıl çalıştığı incelenmelidir. Daha sonra ikili çalıştırılabilir dosya Windows bilgisayarlarda nasıl çalıştığı sorusu cevaplanmalıdır. Böylece çalıştırılabilir dosyalar olan zararlı yazılımların kod bilgisi ve Windows ortamında nasıl etkiler yarattığı daha ayrıntılı olarak incelenebilecektir.

2.1.1. Soyutlama

Bilgisayar üreticileri farklı donanım parçaları kullanarak kendi tasarımları ile bilgisayar üretirler [15]. Bu üretimler doğal olarak birbirlerinden farklıdır. Aynı yazılımın farklı bilgisayarlarda çalışması soyutlama yöntemi ile mümkün olmaktadır [16]. Örneğin, Windows işletim sistemini birçok farklı donanım türünde çalıştırabilirsiniz, çünkü temel donanım işletim sisteminden çıkarılmıştır. Geleneksel bilgisayar mimarisinde, bir bilgisayar sistemi, uygulama ayrıntılarını gizlemenin bir yolunu oluşturan birkaç soyutlama seviyesi olarak temsil edilmektedir. Bilgisayar sistemleri genel olarak aşağıdaki altı farklı soyutlama seviyesi ile tanımlanmaktadır [17]. Bu seviyeleri alttan başlayarak aşağıdaki gibi sıralanmaktadır.

- Donanım

Donanım düzeyi, yalnızca fiziksel seviye, XOR, AND, OR ve NOT kapılar gibi, mantıksal operatörlerin karmaşık kombinasyonlarını dijital mantık olarak bilinen elektrik devrelerinden oluşur [17]. Fiziksel doğası nedeniyle, donanım yazılım tarafından kolayca idare edilemez.

- Mikro kod

Mikro kod seviyesi ayrıca üretici yazılımı olarak da bilinir. Mikro kod yalnızca tasarlandığı tam devre üzerinde çalışır. Donanımla ara yüz oluşturmanın bir yolunu sağlamak için daha yüksek makine kodu seviyesinden çeviri yapan mikro yapılar içerir [17]. Kötü amaçlı yazılım analizi yaparken, genellikle mikro koddan endişe edilmez, çünkü genellikle yazıldığı bilgisayar donanımına özgüdür.

- Makine kodu

Makine kodu seviyesi, işlemciye ne yapmak istediğini söyleyen onaltılık basamaklardan oluşan ikili kodlardan oluşmaktadır [16]. Makine kodu tipik olarak birkaç mikro kod talimatıyla uygulanmaktadır, böylece temel donanım kodları yürütebilir. Makine kodu, yüksek seviyeli bir dilde yazılmış bir bilgisayar programı derlendiğinde oluşturulur.

- Düşük seviyeli diller

Düşük seviyeli bir dil, bir bilgisayar mimarisinin talimat setinin insan tarafından okunabilen bir versiyonudur [18]. En yaygın düşük seviye dil bir assembly dilidir. Kötü amaçlı yazılım analistleri düşük seviyeli dillerde çalışır, çünkü makine kodunun bir insan tarafından anlaşılması zordur. Mov ve jmp gibi basit anımsatıcılardan oluşan düşük seviyeli bir dil metni oluşturmak için bir çözücü (disassembler) kullanılmaktadır. Birçok farklı assembly dili lehçesi vardır. Assembly kodu, kaynak kodu mevcut olmadığında derleme, makine kodundan güvenilir ve tutarlı bir şekilde kurtarılabilecek en üst seviye dildir [19].

- Üst seviye diller

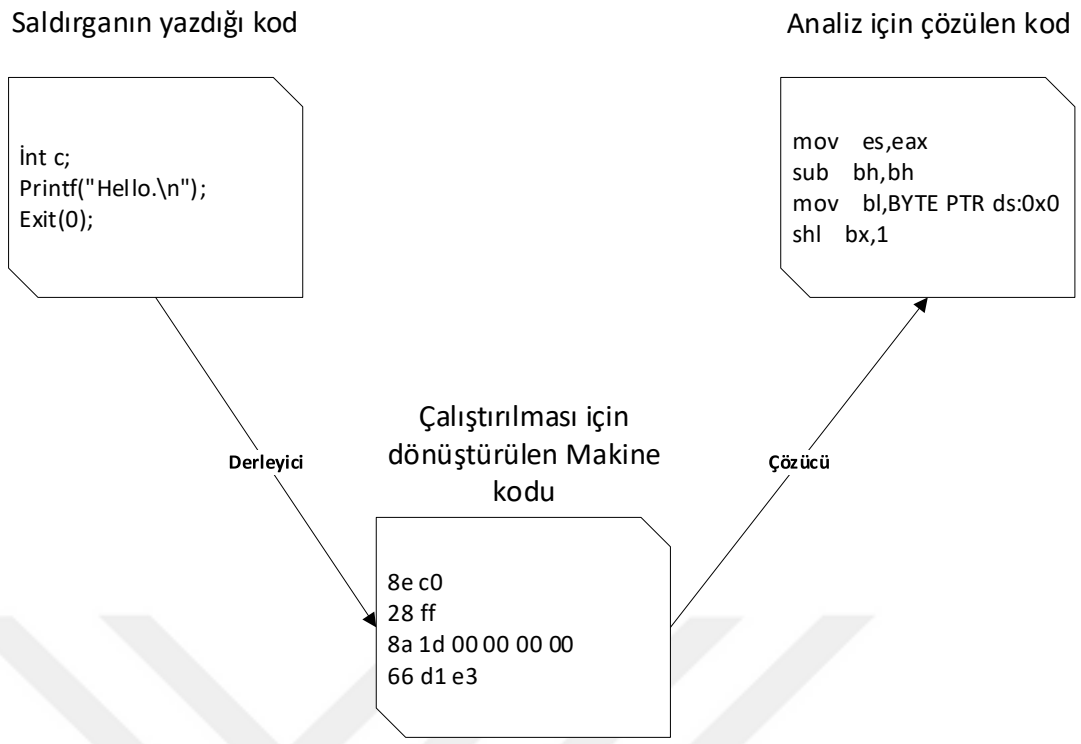
Çoğu bilgisayar programcısı üst seviye dil seviyesinde çalışır. Üst seviye diller C,C++ vb. dillerdir. Üst düzey diller, makine seviyesinden güçlü bir soyutlama sağlar ve programlama mantığı ve akış kontrol mekanizmalarının kullanımını

kolaylaştırır [20]. Bu diller genellikle derleyici tarafından derleme olarak bilinen bir işlemle derleyici tarafından makine koduna dönüştürülür.

- Tercüme Diller

Sözlü diller en üst seviyededir. Birçok programcı C #, Perl, .NET ve Java gibi tercüme edilmiş dilleri kullanır. Bu seviyedeki kod makine kodunda derlenmemiştir; bunun yerine, bayt koduna çevrilir. Bayt kodu, programlama diline özgü bir ara ifadedir. Bayt kodu, çalışma zamanında anında bayt kodunu çalıştırılabilir makine koduna çeviren bir program olan tercüman içinde yürütülür [21]. Tercüman, geleneksel derlenmiş kodla karşılaştırıldığında otomatik bir soyutlama seviyesi sağlar, çünkü işletim sisteminden bağımsız olarak hataları ve bellek yönetimini kendi başına idare edebilmektedir.

Şekil 2.1 kötü amaçlı yazılım analizine katılan üç kodlama seviyesini göstermektedir. Kötü amaçlı yazılım yazarları, yüksek dil düzeyinde ya da assembly dili seviyesinde programlar oluşturur ve CPU tarafından çalıştırılacak makine kodunu oluşturmak için bir derleyici kullanır. Tersine, zararlı yazılım analistleri ve tersine çalışan mühendisler düşük seviyeli dil ile çalışır. Bir programın nasıl çalıştığını anlamak, okuyabileceğimiz ve analiz edebileceğimiz assembly kodunu oluşturmak için bir çözücü (disassembler) kullanılmaktadır [22].



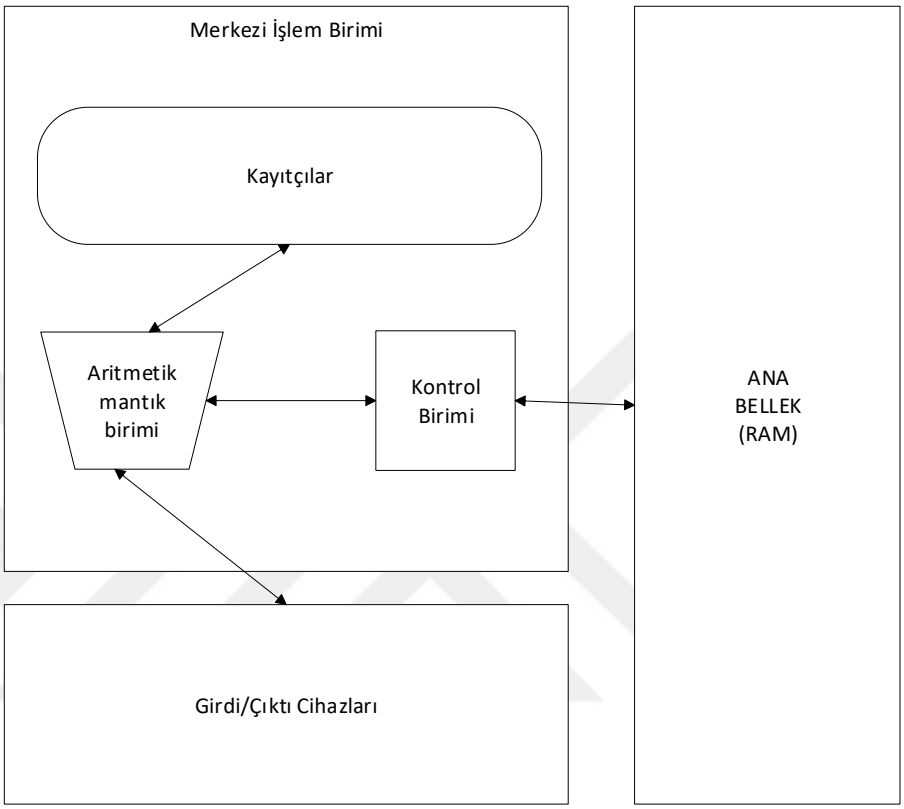
Şekil 2.1: Kod soyutlamanın zararlı yazılımlarda kullanım örneği

Kötü amaçlı yazılımlar genellikle makine kodu düzeyinde ikili biçimde bir diskte depolanmaktadır. Açıklandığı gibi, makine kodu, bilgisayarın hızlı ve verimli bir şekilde çalıştırabileceği kod biçimidir [23]. Kötü amaçlı yazılımı çözerken (Şekil 2.1'de gösterildiği gibi), kötü amaçlı yazılımı ikili kod girdi olarak alır ve genellikle bir çözücü(disassembler) ile birlikte assembly kodu çıktı olarak oluşturulur. Assembly dili aslında bir dil sınıfıdır. Her bir assembly lehçesi tipik olarak, x86, x64, SPARC, PowerPC, MIPS ve ARM gibi tek bir mikroişlemci ailesini programlamak için kullanılır. x86, PC'ler için bugüne kadarki en popüler mimaridir [16]. 32-bit kişisel bilgisayarların çoğu, Intel IA-32 olarak da bilinen x86'dır ve Microsoft Windows'un tüm modern 32-bit sürümleri, x86 mimarisinde çalışacak şekilde tasarlanmıştır.

2.1.2. x86 Mimarisi

Modern bilgisayar mimarilerinin çoğunun içindekiler (x86 dahil), Şekil 2.2'de gösterilen Von Neumann mimarisini izler. Üç donanım bileşenine sahiptir:

- Merkezi işlem birimi (CPU) kodu yürütür.
- Sistemin ana hafızası (RAM) tüm verileri ve kodları saklar.
- Bir giriş / çıkış sistemi (I / O), sabit diskler, klavyeler ve monitörler gibi cihazlarla arayüz oluşturur.



Şekil 2.2: Von Neumann Mimarisi

CPU şu birimleri içermektedir: kontrol birimi (control unit), kayıtçı(register), Aritmetik mantık birimi (ALU) [23]. Kontrol ünitesi, yürütülecek talimatın adresini saklayan bir kayıtçı (talimat göstericisi) kullanarak RAM'den çalıştırma talimatlarını alır. Kayıtçılar (registry), CPU'nun temel veri depolama birimleridir ve zaman kazanmak için kullanılır, böylece CPU'nun RAM'e erişmesine gerek kalmaz. Aritmetik mantık birimi (ALU), RAM'den alınan bir komutu yürütür ve sonuçları kayıtlara veya hafızaya yerleştirir [24]. Bir komut çalıştırıldıktan sonra talimatın alınmasından sonra talimatın alınması ve yürütülmesi işlemi tekrarlanır.

Veri(Data), Bir program ilk yüklendiğinde yeri ayrılan değerleri içeren hafıza bölümüne atıfta bulunmak için kullanılmaktadır. Bu değerlere bazen statik değerler

denir, çünkü program çalışırken değişmez veya programın herhangi bir bölümünde kullanılabildiklerinden global değerler olarak adlandırılabilirler.

Kod (Code), programın görevlerini yerine getirmek için CPU tarafından alınan talimatları içerir. Kod, programın ne yaptığını ve programın görevlerinin nasıl düzenleneceğini kontrol eder.

Öbek (Heap), program yürütme sırasında dinamik bellek için, yeni değerler oluşturmak (tahsis etmek) ve programın artık ihtiyaç duymadığı (boş) değerleri ortadan kaldırmak için kullanılır. Öbek dinamik bellek olarak adlandırılır çünkü program çalışırken içeriği sık sık değişebilir.

Yığın (Stack), yerel değişkenler ve işlevler için parametreler için ve program akışının kontrol edilmesine yardımcı olmak için kullanılır. Yığın kavramı ayrıntılı olarak incelenecektir.

Zararlı yazılım analizinde bu kavramlar büyük önem taşımaktadır [19]. Zararlı yazılımlar yöntemleri atlatmalarda ya da zararlı aktivitelerini saklamakta bu tür kod düzenlemesinde görevli veri yapılarından faydalanmaktadırlar. Bu sebepledir ki, komutlar ve komutların kullanımı zararlı yazılım davranış ve yatkınlık analizinde önemli bir yere sahiptir.

2.1.3. Komutlar

Bir anımsatıcı ve sıfır veya daha fazla işlenenden oluşan bir talimattır. Tablo 2.1'de gösterildiği gibi, hatırlatıcı, verileri taşıyan mov gibi yürütülmesi talimatını tanımlayan bir kelimedir [16]. İşlenenler tipik olarak, kayıtlar veya veriler gibi talimatlar tarafından kullanılan bilgileri tanımlamak için kullanılır.

Bir komut; işlem kodu, 0,1 ya da daha fazla işlenenden meydana gelmektedir. İşlem kodu çalışacak kodun ne olduğunu ifade eder. Mesela mov işlem kodu verileri taşıyan işlemin erişim kodudur. İşlenenler ise o işlemden etkilenen verilerdir. Genellikle ilk işlenen hedef veri adresini, sonraki işlenen ise kaynak veri ya da veri adresini gösterir. İşlenenler veri, hafıza adresi ya da kayıtçı olabilmektedir. İşlenenler, bir komut tarafından kullanılan verileri tanımlamak için kullanılır. Üç tür işlenen kullanılabilir:

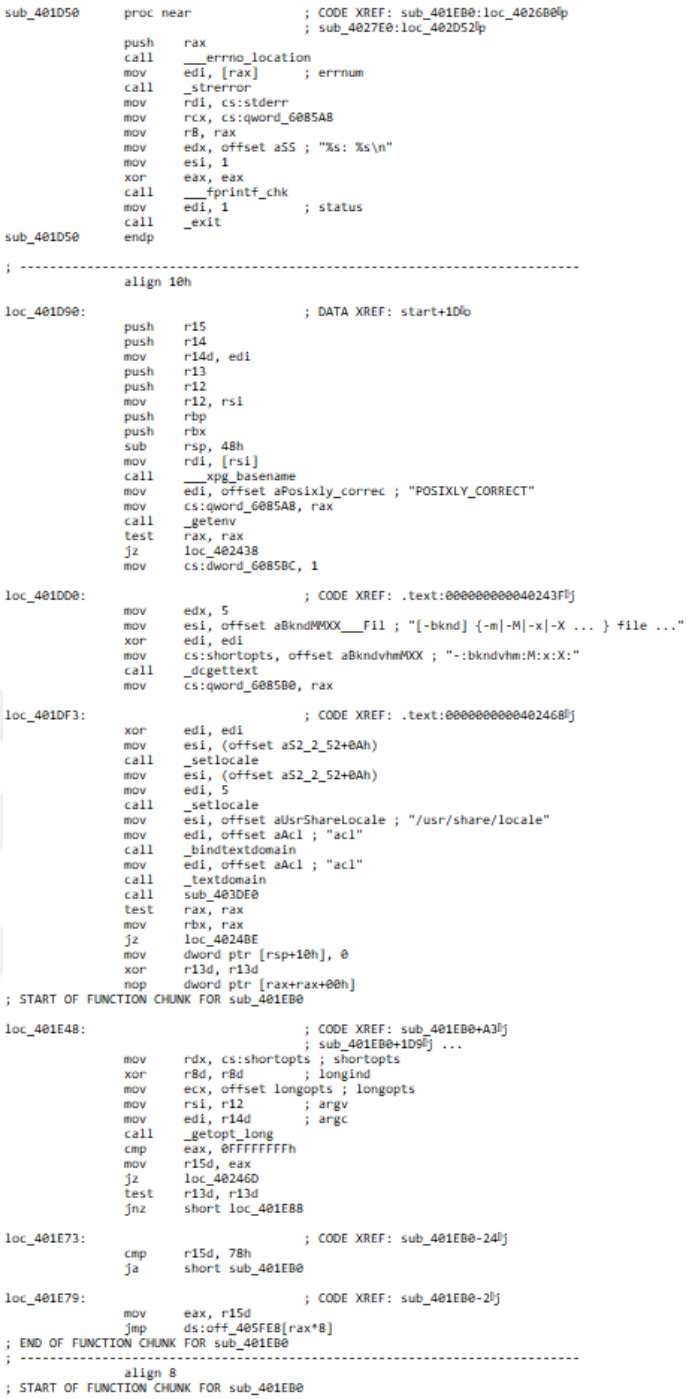
- Anlık işlenenler sabit değerlerdir.

- Kayıtçı işlenenler, ecx gibi kayıtçıları gösterir.
- Hafıza adresi işlenenleri, [eax] gibi parantezler arasında genellikle bir değer, kayıt ya da denklemle belirtilen ilgilenilen değeri içeren bir hafıza adresini ifade eder.

Bir kayıtçı, içeriğine başka bir yerde mevcut olandan daha hızlı bir şekilde erişilebilen CPU içindeki küçük boyutlu veri deposudur. x86 işlemcilerinde geçici depolama veya çalışma alanı olarak kullanılabilen bir kayıt listesi bulunur. Aşağıdaki dört kategoriye giren en yaygın x86 kayıtlarını göstermektedir:

- RegGenel kayıtlar yürütme sırasında CPU tarafından kullanılır.
- Eg Kayıt kayıtları, hafıza bölümlerini izlemek için kullanılır.
- FlagsStatus bayrakları karar vermek için kullanılır.
- PoİşaretçiIer, bir sonraki komutun yürütülmesi için kullanılır.

EAX genellikle fonksiyon çağrıları için dönüş değerini içerir.



```
sub_401D50      proc near               ; CODE XREF: sub_401EB0:loc_4026B0↑p
                                        ; sub_4027E0:loc_402D52↑p
                push    rax
                call    ___errno_location
                mov     edi, [rax]      ; errnum
                call    _strerror
                mov     rdi, cs:stderr
                mov     rcx, cs:qword_6085A8
                mov     r8, rax
                mov     edx, offset aSS ; "%s: %s\n"
                mov     esi, 1
                xor     eax, eax
                call    ___fprintf_chk
                mov     edi, 1          ; status
                call    _exit
sub_401D50      endp

; ---------------------------------------------------------------------------
                align 10h

loc_401D90:                             ; DATA XREF: start+1D↑o
                push    r15
                push    r14
                mov     r14d, edi
                push    r13
                push    r12
                mov     r12, rsi
                push    rbp
                push    rbx
                sub     rsp, 48h
                mov     rdi, [rsi]
                call    ___xpg_basename
                mov     edi, offset aPosixly_correc ; "POSIXLY_CORRECT"
                mov     cs:qword_6085A8, rax
                call    _getenv
                test    rax, rax
                jz      loc_402438
                mov     cs:dword_6085BC, 1

loc_401DD0:                             ; CODE XREF: .text:0000000000402431↑j
                mov     edx, 5
                mov     esi, offset aBkndMMXX___Fil ; "[-bknd] {-m|-M|-x|-X ... } file ..."
                xor     edi, edi
                mov     cs:shortopts, offset aBkndvhmMXX ; "-:bkndvhm:M:x:X:"
                call    _dcgettext
                mov     cs:qword_6085B0, rax

loc_401DF3:                             ; CODE XREF: .text:0000000000402468↓j
                xor     edi, edi
                mov     esi, (offset aS2_2_52+0Ah)
                call    _setlocale
                mov     esi, (offset aS2_2_52+0Ah)
                mov     edi, 5
                call    _setlocale
                mov     esi, offset aUsrShareLocale ; "/usr/share/locale"
                mov     edi, offset aAcl ; "acl"
                call    _bindtextdomain
                mov     edi, offset aAcl ; "acl"
                call    _textdomain
                call    sub_403DE0
                test    rax, rax
                mov     rbx, rax
                jz      loc_4024BE
                mov     dword ptr [rsp+10h], 0
                xor     r13d, r13d
                nop     dword ptr [rax+rax+00h]
; START OF FUNCTION CHUNK FOR sub_401EB0

loc_401E48:                             ; CODE XREF: sub_401EB0+A3↓j
                                        ; sub_401EB0+1D9↓j ...
                mov     rdx, cs:shortopts ; shortopts
                xor     r8d, r8d        ; longind
                mov     ecx, offset longopts ; longopts
                mov     rsi, r12        ; argv
                mov     edi, r14d       ; argc
                call    _getopt_long
                cmp     eax, 0FFFFFFFFh
                mov     r15d, eax
                jz      loc_40246D
                test    r13d, r13d
                jnz     short loc_401E88

loc_401E73:                             ; CODE XREF: sub_401EB0-24↓j
                cmp     r15d, 78h
                ja      short sub_401EB0

loc_401E79:                             ; CODE XREF: sub_401EB0-2↓j
                mov     eax, r15d
                jmp     ds:off_405FE8[rax*8]
; END OF FUNCTION CHUNK FOR sub_401EB0
; ---------------------------------------------------------------------------
                align 8
; START OF FUNCTION CHUNK FOR sub_401EB0
```

Şekil 2.3: Assembly Kod örneği – setfacl zararlısından alınmış kod parçası

2.3.1.1. Yığın Komutları

Fonksiyon çağrımlarında, lokal değişkenler ve akış kontrolü yönetiminde yığın (stack) veri yapısı kullanılır. Öğeleri yığına iter ve sonra bu öğeleri çıkarırsınız. Bir yığın son giren ilk çıkar (LIFO) yapısıdır. Örneğin, 1, 2 ve ardından 3 sayılarını

(sırayla) basarsanız, ilk çıkarılan öğe 3 olacaktır, çünkü yığının üzerine itilen son öğedir. X86 mimarisinin yığın mekanizması için kayıtçı desteği vardır. Tutamaç desteği ESP ve EBP kayıtlarını içerir. ESP, yığın göstericisidir ve genellikle yığının üstüne işaret eden bir bellek adresi içerir. Bu kaydın değeri, öğeler itildikçe ve yığından çıkarıldığında değişir. EBP belirli bir işlev içinde tutarlı kalan temel işaretçidir. Program yerel değişkenlerin ve parametrelerin konumunu izlemek için bir yer tutucu olarak kullanılmaktadır. Yığın yönetimi push, pop, call, leave, enter ve ret komutları ile sağlanmaktadır. Yığın, bellekte yukarıdan aşağıya bir biçimde tahsis edilir ve en yüksek adresler önce tahsis edilir ve kullanılır. Değerler yığına itildiğinde, daha küçük adresler kullanılır. Yığın yalnızca kısa süreli depolama için kullanılır. Sık sık yerel değişkenleri, parametreleri ve dönüş adresini saklar. Birincil kullanımı, işlev çağrıları arasında değiştirilen verilerin yönetimi içindir. Bu yönetimin uygulanması derleyiciler arasında farklılık gösterir, ancak en yaygın kural, yerel değişkenler ve EBP'ye göre referans alınacak parametrelerdir. Yığın yönetimi zararlı yazılımlar tarafından büyük öneme sahiptir. Gizlenme, saklanma, enjeksiyon gibi faaliyetlerini ya da analizden kaçınma taktiklerini bu komutlar ile başarırlar [25].

2.3.1.2. Koşul Komutları

Tüm programlama dilleri karşılaştırma yapma ve bu karşılaştırmalara dayanarak karar verme yeteneğine sahiptir. Şartlı Cümleler karşılaştırmayı yapan talimatlardır. En popüler iki şartlı talimat test ve cmp'dir. Ek olarak hem koşul hem dallanma bildiren komutlar da vardır.

2.1.3.3. Dallanma Komutları

Dallanma komutları, programın akışına bağlı olarak koşullu olarak yürütülen bir kod dizisidir. Dallanma terimi bir programın kolları boyunca kontrol akışını tanımlamak için kullanılır. Dallanma oluşumunun en popüler yolu, atlama yönergeleridir. Assembly dilinde “if” işlem kodu bulunmamaktadır. Bunun yerine dallanma işlem kodları kullanılır. Bu, koşullu atlama işlem kodları ile sağlanır. Koşullu atlamalar, atlamak veya bir sonraki talimata devam etmek için bayrakları

kullanır. 30'dan fazla tip koşullu atlama işlem kodu vardır, ancak bunlar içinden yalnızca küçük bir kümelik işlem kodu sıklıkla kullanılmaktadır [19, 20].

2.1.3.4. Arabellek Yönetim Komutları

Rep komutları, veri arabelleklerini (data buffer) değiştirmek için bir talimat setidir [26]. Genellikle bir bayt dizisi biçimindedir, ancak tek veya çift kelime de olabilirler. En yaygın , veri arabellek manipülasyon talimatları, movsx, cmpsx, stosx ve scasx'tir, burada sırasıyla bayt, kelime veya çift kelime için x = b, w veya d'dir. movsb, cmpsb vb. kulanılmaktadır. ESI ve EDI kayıtçıları bu işlemlerde kullanmaktadır. ESI, kaynak dizin kaydı ve EDI, hedef dizin kaydıdır. ECX sayım değişkeni olarak kullanılır. X86'da yineleme önekleri çok baytlı işlemler için kullanılır [27]. Temsilci talimatı, ESI ve EDI ofsetlerini arttırır ve ECX kaydını azaltır. Rep öneki ECX = 0 olana kadar devam eder. Repe / repz ve repne / repnz önekleri ECX = 0 olana veya ZF = 1 veya 0 olana kadar devam eder. Bu nedenle, çoğu veri arabelleği manipülasyon talimatında, ESI, EDI ve ECX, rep komutunun faydalı olması için uygun şekilde başlatılmalıdır [28].

2.1.3.5. Fonksiyon Çağrım Komutları

İşlevler, bir programdaki belirli bir görevi yerine getiren ve kalan koddan nispeten bağımsız olan kod bölümleridir. Ana kod çağırır ve geçici olarak ana koda dönmeden önce yürütmeyi işlevlere aktarır. Yığının bir program tarafından nasıl kullanıldığı, verilen bir ikili dosya boyunca tutarlıdır. İşlevler genellikle bir prolog içerir [28]. Prolog fonksiyonun başlangıcında birkaç kod satırıdır. Prolog yığını kullanır ve kayıtlar işlev içinde kullanılmak üzere hazırlanır. Aynı şekilde, bir fonksiyonun sonundaki bir epilog yığını istifler ve fonksiyon çağrılmadan önceki durumlarını kaydeder. Aşağıdaki liste, fonksiyon çağrıları için en yaygın uygulamanın akışını özetlemektedir. Bir süre sonra, istiflerin düzenini açıklığa kavuşturan ayrı bir istif karesi için istif düzeninin bir diyagramını gösterir.

i) Argümanlar push komutları kullanılarak yığına yerleştirilir.

ii) Call bellek_adresi kullanılarak bir işlev çağrılır. Bu, geçerli komut adresinin (EIP kayıtçısının içeriği) yığına itilmesine neden olur. İşlev bittiğinde bu adres ana koda dönmek için kullanılacaktır. İşlev başladığında, EIP değeri, bellek_adresi (işlevin başlangıcı) olarak ayarlanır.

iii) Bir işlev prologunun kullanılmasıyla, yerel değişkenler için yığında boşluk bırakılır ve EBP (temel işaretçi) yığının üzerine itilir. Bu, arama işlevinde EBP'yi kaydetmek için yapılır.

iv) Fonksiyon çalışmalarını gerçekleştirir.

v) Bir fonksiyon epilogunun kullanımıyla istif geri yüklenir. ESP, yerel değişkenleri serbest bırakacak şekilde ayarlanır ve EBP, geri çağırma işlevinin değişkenlerini doğru bir şekilde ele alması için geri yüklenir. Ayrılma komutu (ret) bir epilog olarak kullanılabilir çünkü ESP'nin değeri EBP'ye aktarılır ve EBP'yi yığından çıkarır.

vi) İşlev, ret komutunu çağırarak döner. Bu, geri dönüş adresini yığından ve EIP'ye getirir; böylece program, orijinal çağrının yapıldığı yerden çalışmaya devam eder.

vii) Yığın, daha sonra tekrar kullanılmayacaklarsa gönderilen argümanları kaldırmak için ayarlanır.

Bir çağrı yapıldığında, yeni bir yığın çerçevesi oluşturulur. Bu yeni oluşturulan yığın, çağrılan fonksiyon için hafızada dinamik bir alan oluşturmaktadır. Bir işlev çağrıldığı bellek adresine geri dönene kadar kendi yığın çerçevesini korur. Bellekte açılan yığın çerçevesinde en altta fonksiyonun girdi değişkenleri olmak üzere yukarı doğru fonksiyon içinde değişken tanımlandıkça yeni bellek alanları açılmaktadır. Bellekte fonksiyon için açılan yığın alanı, fonksiyon bitene kadar (retn komutu) yer tutmaya devam edecektir. Fonksiyon bittiğinde çağıranın yığın çerçevesi geri yüklenir ve yürütme tekrar çağrı işlevine aktarılır.

Push veya pop komutlarını kullanmadan veri yığınından okumak mümkündür. Örneğin, mov eax, ss: [esp] komutu doğrudan yığının üstüne erişir. Bu, ESP kaydı etkilenmemesi dışında, pop eax ile aynıdır. Kullanılan kural, derleyiciye ve derleyicinin nasıl yapılandırıldığına bağlıdır. Bu alternatif kullanımlar, fonksiyon çağrımını gizlemek için zararlı yazılımcılar tarafından kullanılması tercih edilebilmektedir.

Tablo 2.1: Kategorilerine göre ayrılmış işlem kodları

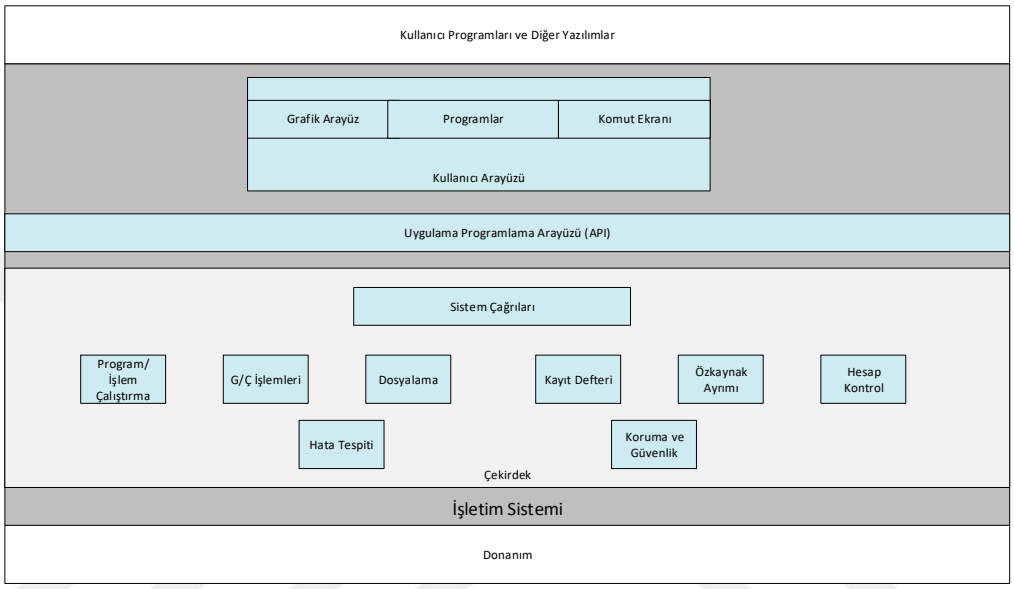
Kategori	Komut
Aritmetik	AAA, AAD, AAS, ADC, ADD, ADDPD, ADDPS, ADDSD, ADDSS, DEC, DIV, CLC, CLD, CLI, CMC, DAA, DAS, DAA, DAS, DEC, DIV, DIVPD, DIVPS, DIVSD, DIVSS, SUB, FSUB, IDIV, IMUL, INC, NEG, SAL,SAR,
Koşul	AND, OR, ORPS, ORPD, XOR, XORPD, XORPS, BT, CMP, CMPPD, CMPPS, CMPS, TEST, UD2, UCOMISD, UCOMISS, FTST, PCMPGTB, PAND, PANDN
Dallanma	JMP,JA, JAE, JB, JBE, JC, JCXZ, JECXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ
Arabellek Yönetimi	REP, REPE, REPNE, INS, OUTS, MOVS, LODS, STOS, CMPS, SCAS, SCASB, SCAS
Yığın Komutları	POP, PUSH, CALL, RET, RETN, ENTER,FINCSTP, LEAVE, PUSHA, PUSHAD, PUSHF, PUSHFD
Bellek Kontrol	MOV, MOVAPD, MOVAPS, MOVD, MOVDQA, MOVDQU, MOVDQ2Q, MOVHLPS, MOVHPD, STORE, CBW, CWDE, CMOVcc, FBLD, FBSTP, LEA, PINSRW

X86 mimarisi, en popülerleri pusha ve pushad olan çekme ve itme için ek talimatlar sunar. Bu talimatlar, tüm kayıtları istifin üzerine iter ve genellikle tüm kayıtları istiften çeken popa ve popad ile kullanılır. Bu talimatlar tipik olarak, kayıtların mevcut durumunu yığına kaydetmek istediğinde kabuk kodunda karşılaşılır, böylece daha sonra geri yüklenebilirler. Derleyiciler bu talimatları nadiren kullanmaktadır.

2.1.4. Windows Uygulama Programlama Arayüzleri (API)

Windows API, kötü amaçlı yazılımın Microsoft kitaplıklarıyla etkileşim kurma biçimini düzenleyen geniş bir işlevsellik kümesidir. İşletim sistemi yönetim

araçlarına ve bilgisayar kaynaklarına erişim için kullanılır [24]. Windows API, o kadar geniştir ki, yalnızca Windows uygulamalarının geliştiricilerinin üçüncü taraf kitaplıklara çok az ihtiyacı vardır. Kullanım amaçlarına göre API'leri kategorilere ayırırsak, şu türleri saymak mümkündür: Tutamaçlar (handles), dosya yönetimi, kayıt defteri, Ağ yönetimi, işlem ve iş parçacığı (processes and threads), muteks kontrolü, arka plan hizmetleri.



Şekil 2.4: Windows işletim sisteminde API ve Sistem Fonksiyonları

2.1.4.1. Tutamaçlar

Tutamaçlar, işletim sistemi içerisinde pencere, işlem, modül, menü, dosya vb. Gibi açılan veya oluşturulan öğelerdir [29]. tutamaçlar, başka bir yerde bir nesneye ya da bellek konumuna gönderme yaptıkları için işaretçiler gibidir. Ancak, işaretçilerden farklı olarak, tutamaçlar aritmetik işlemlerde kullanılamaz ve nesnenin adresini her zaman temsil etmezler. Bir tanıtıcıyla yapabileceğiniz tek şey, aynı nesneye başvurmak için daha sonra işlev çağrısında saklamak ve kullanmaktır. Tutamaç yönetimi *CloseHandle, CompareObjectHandles, DuplicateHandle, SetHandleInformation* gibi fonksiyonlar ile yapılmaktadır.

2.1.4.2. Dosya yönetimi

Kötü amaçlı yazılımın sistemle etkileşime girmesinin en yaygın yollarından biri, dosyalar oluşturmak veya bunları değiştirmektir [19]. Dosya yönetimi *CreateFile, ReadFile, WriteFile, CreateFileMapping and MapViewOfFile* gibi fonksiyonlarla sağlanmaktadır. Windows, normal dosyalara benzer şekilde erişilebilen ancak sürücü harfleri ve klasörleriyle (c: \ docs gibi) erişilemeyen çok sayıda dosya türüne sahiptir. Kötü amaçlı programlar genellikle özel dosyalar kullanır. Bazı özel dosyalar normal dosyalardan daha gizli olabilir, çünkü dizin listelerinde görünmezler. Bazı özel dosyalar sistem donanımına ve dahili verilere daha fazla erişim sağlayabilir.

2.1.4.3. Kayıt Defteri

Windows kayıt defteri, işletim sistemi ve programların yapılandırma bilgilerini depolamak için kullanılır. Dosya sistemi gibi, kayıt defterleri iyi bir ana bilgisayar tabanlı göstergeler kaynağıdır ve kötü amaçlı yazılımın işlevselliği hakkında faydalı bilgiler ortaya koyabilir [30]. Windows'un önceki sürümleri .ini dosyalarını yapılandırma bilgilerini depolamak için kullanırdı. Kayıt defteri, performansı artırmak için hiyerarşik bir bilgi veri tabanı olarak oluşturulmuş ve daha fazla uygulama bilgisini depolamak için kullandıkça önemi artmıştır. Ağ, sürücü, başlangıç, kullanıcı hesabı ve diğer bilgiler de dahil olmak üzere neredeyse tüm Windows yapılandırma bilgileri kayıt defterinde depolanır [29]. Kötü amaçlı yazılım, kalıcılık veya yapılandırma verileri için genellikle kayıt defterini kullanır [31,19,32]. Kötü amaçlı yazılım, bilgisayar başlatıldığında otomatik olarak çalışmasına izin verecek kayıt defterine girişler ekler. Kayıt defteri o kadar büyük ki kötü amaçlı yazılımın kalıcı olabilmek için kayıt defteri kullanımı için pek çok yol vardır.

Kayıt defterine incelemeden önce, Microsoft belgelerini anlamak için bilinmesi gereken birkaç önemli kayıt defteri terimi vardır: Anahtar, alt-anahtar, kök-anahtar, değer kaydı, değer (veri). Anahtar kayıt defteri içinde yer alan klasörlere verilen addır. Kök-anahtar özel klasör tipidir. 5 farklı özel dosya tipi vardır. Alt-anahtar ise kayıt defteri içinde yer alan ana klasörlerin içindeki klasörlerdir. Anahtar adı verilen klasörlerde ya alt-anahtarlar ya da değer kayıtları bulunur. Değer kayıtları isim ve değerden oluşmaktadır. İsim o kayıtın ismidir. Değer (veri) ise o değer kaydındaki değere tekabül etmektedir.

RegOpenKeyEx Düzenlemek ve sorgulamak için bir kayıt defteri açar. Bir kayıt defteri anahtarını önce açmadan sorgulamanıza ve düzenlemenize izin veren işlevler vardır, ancak çoğu program yine de *RegOpenKeyEx'i* kullanır. *RegSetValueEx* Kayıt defterine yeni bir değer ekler ve verilerini ayarlar. *RegGetValue* Kayıt defterinde bir değer girişi için verileri döndürür. kötü amaçlı yazılım kodu, Kayıt anahtarından Çalıştır anahtarını açar ve programın Windows her başlatıldığında çalışması için bir değer ekler. Altı parametre alan *RegSetValueEx* işlevi, bir kayıt defteri değeri girişini düzenler veya yoksa yeni bir tane oluşturur.

2.1.4.4. Ağ Fonksiyonları

Winsock API'sine ek olarak, *WinINet* API adı verilen daha üst düzey bir API vardır. *WinINet* API işlevleri Wininet.dll dosyasında depolanır. Bir program bu DLL'den işlevleri alırsa, daha yüksek düzeyde ağ API'leri kullanıyordur. WinINet API, uygulama katmanında HTTP ve FTP gibi protokolleri uygular. Açılan bağlantılara dayanarak kötü amaçlı yazılımın ne yaptığını anlayabilirsiniz. İnternet bağlantısını başlatmak için *InternetOpen* kullanılır. *InternetOpenUrl* bir URL'ye bağlanmak için kullanılır (bu bir HTTP sayfası veya bir FTP kaynağı olabilir). *InternetReadFile*, programın Internet'ten indirilen bir dosyadan verileri okumasını sağlayan *ReadFile* işlevi gibi çalışır.

2.1.4.5. İşlemler

Kötü amaçlı yazılım, yeni bir işlem oluşturarak veya mevcut olanı değiştirerek mevcut programın dışında da kod çalıştırabilir [33]. İşlem, Windows tarafından yürütülen bir programdır. Her işlem, açık tutamaç ve bellek gibi kendi kaynaklarını yönetir. Bir işlem CPU tarafından yürütülen bir veya daha fazla iş parçacığı içerir. Geleneksel olarak, kötü amaçlı yazılım kendi bağımsız işleminden oluşur, ancak daha yeni kötü amaçlı yazılım kodunu başka bir işlemin bir parçası olarak daha sık uygular [34].

Windows, kaynakları yönetmek ve ayrı programların birbiriyle karışmasını önlemek için işlemleri kapsayıcı olarak kullanır [35]. Herhangi bir zamanda bir Windows sisteminde çalışan, tümü CPU, dosya sistemi, bellek ve donanım dahil olmak üzere aynı kaynakları paylaşan en az 20 ila 30 işlem vardır. Her programın

diğerleriyle paylaşımını yönetmesi gerekiyorsa, program yazmak çok zor olacaktır. İşletim sistemi tüm işlemlerin bu kaynaklara birbirlerini etkilemeden erişmelerini sağlar. Süreçler aynı zamanda bir programdaki hataları veya çökmeleri diğer programları etkilemesini önleyerek dengeye katkıda bulunur.

İşletim sisteminin işlemler arasında paylaşması için özellikle önemli olan bir kaynak sistem belleğidir. Bunu başarmak için, her bir işleme diğer tüm işlemlerden ayrı olan ve işlemin kullanabileceği bir bellek adresleri toplamı olan bir bellek alanı verilir. İşlem bellek gerektirdiğinde, işletim sistemi bellek ayırır ve işleme belleğe erişmek için kullanabileceği bir adres verir. İşlemler bellek adreslerini paylaşabilir ve sıklıkla yaparlar. Örneğin, bir işlem bir şeyi 0x00400000 bellek adresinde saklarsa, diğeri bir şey bu adreste saklayabilir ve işlemler çakışmaz. Adresler aynı, ancak verileri saklayan fiziksel bellek aynı değildir. Kötü amaçlı yazılımlar tarafından yeni bir işlem oluşturmak için en sık kullanılan işlev *CreateProcess*. *CreateProcess* işlevinin parametrelerinden biri olan *STARTUPINFO* yapısı, bir işlem için standart girdi, standart çıktı ve standart hata akışlarını içeren bir tanıtıcı içerir.

2.1.4.6. İş Parçacıkları

İşlemler yürütme kabıdır, ancak iş parçacıkları Windows işletim sisteminin yürüttüğü işlemdir. İş parçacıkları, diğer iş parçacıkları beklemeden CPU tarafından yürütülen bağımsız komut dizileridir [35]. Bir işlem, işlemin içindeki kodun bir kısmını yürüten bir veya daha fazla iş parçacığı içerir. Bir işlemdeki iş parçacıklarının tümü aynı bellek alanını paylaşır, ancak her birinin kendi işlemci kayıtları ve yığını vardır.

CreateThread işlevi yeni iş parçacıkları oluşturmak için kullanılır. İşlevin arayanı, genellikle başlatma işlevi olarak adlandırılan bir başlangıç adresi belirtir. Kötü amaçlı yazılımlar, *CreateThread'i* aşağıdakiler gibi birden çok şekilde kullanabilir:

- Kötü amaçlı yazılım, *CreateThread* adlı bir işleme ve başlangıç adresi olarak *LoadLibrary* adresiyle yeni bir kötü amaçlı kitaplık yüklemek için *CreateThread'i* kullanabilir. (*CreateThread'e* iletilen argüman, yüklenecek kütüphanenin adıdır. Yeni DLL, işlemdeki belleğe yüklenir ve *DllMain* çağrılır.)

- Kötü amaçlı yazılım giriş ve çıkış için iki yeni iş parçacığı oluşturabilir: biri soket veya borudan dinlemek ve daha sonra bunu bir sürecin standart girdisine çıkarmak, diğeri standart çıktıdan okumak ve bunu bir sokete veya boruya göndermek. Kötü amaçlı yazılımın amacı, çalışan uygulama ile sorunsuz iletişim kurmak için tüm bilgileri tek bir sokete veya boruya göndermektir.

2.1.4.7. Muteks

İş parçacıkları ve işlemler çekirdekdeyken mutant olarak adlandırılan muteksleri veri paylaşımı için kullanmaktadır. Muteksler, çoklu işlemleri ve konuları koordine eden global nesnelerdir [29]. Muteksler çoğunlukla paylaşılan kaynaklara erişimi kontrol etmek için kullanılır ve genellikle kötü amaçlı yazılımlar tarafından kullanılır. Örneğin, iki iş parçacığı bir bellek yapısına erişmesi gerekiyorsa, ancak bir seferde yalnızca bir tanesi güvenle erişebiliyorsa, erişimi denetlemek için bir muteks kullanılabilir.

Bir seferde yalnızca bir iş parçacığı bir muteks sahibi olabilir. Mutexes kötü amaçlı yazılım analizi için önemlidir, çünkü genellikle ana bilgisayar tabanlı göstergeler yapan sabit kodlanmış adlar kullanırlar. Sabit kodlu adlar yaygındır, çünkü bir muteksin başka bir şekilde iletişim kurmayan iki işlem tarafından kullanılıyorsa bir muteks adının tutarlı olması gerekir. Iş parçacığı, *WaitForSingleObject* çağrısı ile mutex erişim kazanır ve daha sonra erişmeye çalışan herhangi bir iş parçacığı beklemeniz gerekir. Bir iş parçacığı bir muteks kullanılarak tamamlandığında, *ReleaseMutex* işlevini kullanır. *CreateMutex* işleviyle bir muteks oluşturulabilir. Bir işlem, *OpenMutex* çağrısını kullanarak başka bir işlemin muteksini işleyebilir.

2.1.4.8. Hizmetler

Kötü amaçlı yazılımın ek kod yürütmesi için bir başka yol da onu hizmet olarak yüklemektir. Windows, arka plan uygulamaları olarak çalışan hizmetleri kullanarak görevlerin kendi işlemleri veya iş parçacıkları olmadan çalışmasına izin verir; kod, kullanıcı girişi olmadan Windows servis yöneticisi tarafından

programlanır ve çalıştırılır [30]. Bir Windows işletim sisteminde herhangi bir zamanda, birkaç servis çalışıyor.

OpenSCManager Servisle ilgili tüm fonksiyon çağrıları için kullanılan servis kontrol yöneticisine bir tanıtıcı döndürür. Servislerle etkileşime girecek tüm kodlar bu fonksiyonu çağırır. *CreateService* Servis kontrol yöneticisine yeni bir servis ekler ve çağıranın, servisin açılışta otomatik olarak başlayıp başlamayacağını ya da elle başlatılması gerektiğini belirtmesine izin verir. *StartService* bir servisi başlatır ve yalnızca servis el ile başlatılacak şekilde ayarlandıysa kullanılır.

2.1.4.9. Çekirdek - Kullanıcı Düzeyi

Windows iki işlemci ayrıcalık düzeyi kullanır: çekirdek modu ve kullanıcı modu [24]. Sistem, işlemler, iş parçacıkları, tutamaçlar ve diğer öğeler hakkında bilgi almak için kullanılabilecek bir dizi çekirdek seviyesinde API çağrısı vardır; *NtQuerySystemInformation*,*NtQueryInformationProcess*,*NtQueryInformationThread*, *NtQueryInformationFile* ve *NtQueryInformationKey*. Bu çağrılar, mevcut herhangi bir Win32 çağrısından çok daha ayrıntılı bilgi sağlar ve bu işlevlerden bazıları, dosyalar, işlemler, iş parçacığı vb. İçin ince taneli öznitelikler ayarlamanıza izin verir. Kötü amaçlı yazılım yazarları ile popüler olan bir başka Yerel API işlevi de *NtContinue*'dur. Bu işlev bir kural dışı durumdan(exception) geri dönmek için kullanılır ve bir kural dışı durum işlendikten sonra yürütmeyi bir programın ana iş parçacığına geri aktarmak anlamına gelir.

2.1.5. Dosya Formatı

Zararlı yazılımların incelenmesine geçmeden önce, yazılımların ne yapıda olduğu ve nasıl çalıştığı üstünde durulmalıdır. İşletim sistemine bağlı olarak dosya yapısı farklılık göstermektedir. İşletim sistemi kendi tanıdığı ikili dosyaları çalıştırabilir. Zararlı yazılımlar da bu dosya yapısında olmak zorundadırlar. Windows ortamında PE dosya yapısına sahip ikili dosyalar çalıştırılabilirken [30], linuxta ise ELF dosya yapısı bir ikilinin çalışabilmesi için sahip olması gereken yapıyı nitelemektedir. Bu kısımda dosya yapıları anlatılacak ve Windows ortamında çalışan PE dosya yapısı ayrıntılı olarak incelenecektir.

Tablo 2.2: İşletim Sistemi Özkaynak türleri ve API çağrımları.

İşletim Sistemi Kaynak Türü	API çağrımları listesi
Hizmet	OpenSCManager, OpenService, StartService
İşlem	NtOpenSection, ZwMapViewOfSection, NtFreeVirtualMemory, NtCreateSection, CreateProcessInternal, ExitProcess
Dosya Sistemi	NtCreateFile, NtReadFile, NtSetInformationFile, NtOpenFile, NtWriteFile, DeviceIoControl, CreateDirectory, DeleteFile, FindFirstFile, NtDeviceIoControlFile, NtQueryInformationFile
Kayıt Defteri	RegOpenKey, RegSetValue, RegCloseKey, RegDeleteValue, RegQueryValue, RegCreateKey, NtOpenKey, NtQueryValueKey, RegEnumValue, RegEnumKey, NtQueryKey, RegQueryInfoKey
Sistem	NtDelayExecution, FindWindow, SetWindowsHook, RemoveDirectory, GetSystemMetrics, LookupPrivilegeValue
Ağ	WSAStartup, getaddrinfo

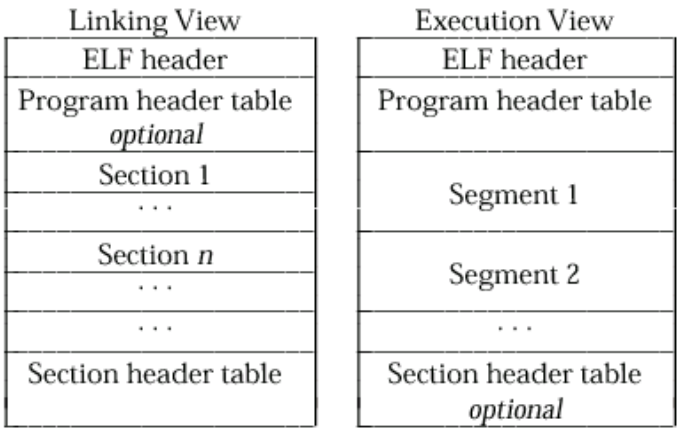
2.1.5.1. İkili Dosya

Modern bilgisayarlar hesaplarını, tüm sayıları birler ve sıfırlar dizisi olarak ifade eden ikili sayısal sistemi kullanarak yaparlar. Bu sistemlerin yürüttüğü makine koduna ikili kod denir [20]. Her program ikili koddan (makine talimatları) ve verilerden (değişkenler, sabitler ve benzeri) oluşmaktadır. Belirli bir sistemdeki tüm farklı programları takip etmek için, her bir programa ait tüm kodu ve verileri tek bir bağımsız dosyada saklamak için bir yol gerekir. Bu dosyalar çalıştırılabilir ikili programlar içerdiğinden, bunlar çalıştırılabilir ikili dosyalar veya yalnızca ikili dosyalar olarak adlandırılır.

İkili dosyalar, C veya C ++ gibi insan tarafından okunabilen kaynak kodlarını işlemcinizin çalıştırabileceği makine koduna çevirme işlemi olan derleme yoluyla üretilir [20]. Şekil x, C kodu için tipik bir derleme işleminde yer alan adımları göstermektedir (C++ derlemesi için adımlar benzerdir). C kodunu derlemek, tam derleme işleminde olduğu gibi dört aşamayı içerir. Aşamalar ön işleme, derleme, asamble ve bağlamadır. Modern derleyiciler genellikle bu aşamaların bir kısmını veya tamamını birleştirmektedir.

2.1.5.2. Çalıştırılabilir ve Bağlanabilir Dosya Formatı (ELF)

ELF Linux işletim sistemlerinde yürütülebilir dosyalar, nesne dosyaları, paylaşılan kütüphaneler ve çekirdek dökümleri için kullanılan ikili dosya formatıdır.



Şekil 2.5: EFL dosya yapısı

2.1.5.3. Taşınabilir Yürütülebilir Dosya Formatı

Taşınabilir Yürütülebilir (PE) dosya formatı, Windows yürütülebilir dosyaları, nesne kodu ve DLL dosyaları tarafından kullanılır [29]. PE dosya formatı, Windows işletim sistemi yükleyicisinin sarılmış yürütülebilir kodu yönetmesi için gerekli bilgileri içeren bir veri yapısıdır. Neredeyse Windows tarafından yüklenen çalıştırılabilir kod içeren her dosya PE dosya biçimindedir, ancak bazı eski dosya biçimleri kötü amaçlı yazılımlarda nadiren görülür. PE dosyaları kod, uygulama türü, gerekli kütüphane işlevleri ve alan gereksinimleri hakkında bilgiler içeren bir başlık ile başlar.

Bir PE dosyası, dosyanın başında, PE'nin hangi dosyaları bulması gerektiğini ve bu dosyaların içinde hangi işlevleri gerektirdiğini belirten bir alma(import) ve verme(export) tablosuna sahiptir. Verme, DLL'nin hangi işlevleri sağladığını gösterir. Ayrıca, işlevleri bulmak için dosyanın bir kez belleğe yüklendiğini de gösterir. İçe aktarma tablosu, PE dosyasının DLL'lerde kullandığı tüm işlevleri ve bunun yanı sıra, içe aktarılan işlevin bulunduğu DLL'nin adını listeler.

PE dosya başlığı ayrıca bir yürütülebilir dosya tarafından kullanılan belirli işlevler hakkında bilgi içerir. Bu Windows işlevlerinin adları, yürütülebilir dosyanın yaptığı hakkında önemli nitelikler taşımaktadır. Microsoft, Windows API'sini Microsoft Geliştirici Ağı (MSDN) kitaplığı yoluyla belgelendirme konusunda mükemmel bir iş çıkarmaktadır. Zararlı yazılımların davranış analizinde bu işlevler büyük önem taşımaktadır.

İçe aktarmalar gibi, DLL'ler ve EXE'ler diğer programlarla ve kodlarla etkileşimde bulunmak için işlevleri dışa aktarır. Genellikle, bir DLL bir veya daha fazla işlev uygular ve bunları sonradan alıp kullanabilen bir yürütülebilir dosya tarafından kullanılmak üzere verir. PE dosyası, bir dosyanın hangi işlevleri dışa aktardığı hakkında bilgi içerir. DLL'ler EXE tarafından kullanılan işlevleri sağlamak için özel olarak uygulandığından, dışa aktarılan işlevler en çok DLL dosyalarında yaygındır. EXE'ler diğer EXE'ler için işlevsellik sağlamak üzere tasarlanmamıştır ve dışa aktarılan işlevler nadirdir. Dışa aktarılanları bir yürütülebilir dosyada bulursanız, genellikle yararlı bilgiler sağlar.

Çoğu durumda, yazılım yazarları verilen işlevlerini yararlı bilgiler sağlayacak şekilde adlandırırlar. Ortak bir kural, Microsoft belgelerinde kullanılan adı kullanmaktır. Örneğin, bir programı servis olarak çalıştırmak için önce bir *ServiceMain* işlevi tanımlamanız gerekir. *ServiceMain* adı verilen verilen bir işlevin varlığı, zararlı yazılımın bir hizmetin parçası olarak çalıştığını söyler.

PE dosya başlıkları, yalnızca içe aktarmadan çok daha fazla bilgi sağlayabilmektedir. PE dosya formatı bir başlık ve bunu takiben bir dizi bölüm içerir. Üstbilgi, dosyanın kendisiyle ilgili meta veriler içerir. Başlığın ardından, her biri yararlı bilgiler içeren dosyanın asıl bölümleri vardır. Bir PE dosyasındaki en yaygın ve ilginç bölümler tablo 2.3'de gösterildiği gibidir.

Tablo 2.3: PE dosya kısımları

Kısım	Açıklama
File header	Dosya ve DOS, magic bilgilerini içerir
.text	Çalıştırılabilir kodları içerir
.rdata	Programın herhangi bir yerinden ulaşılabilecek salt okunur verileri tutar
.data	Global verileri saklar
.idata	İçeri aktarılan fonksiyonların bilgisini tutar
.edata	Dışarı verilen fonksiyonların bilgisini tutar
.pdata	Sadece 64-bit dosyalarda bulunur ve hata yönetim bilgilerini içerir
.rsrc	Uygulama içi öz kaynaklar burada bulunur
.reloc	Kullanılacak olan kütüphane bilgilerini belirtir

2.1.5.4. Dinamik Bağlantılı Kitaplıkları (DLL)

Dinamik bağlantı kitaplıkları (DLL), birden çok uygulama arasında kodu paylaşmak için kullanmanın mevcut Windows yoludur. Bir DLL, yalnız çalışmayan ancak diğer uygulamalar tarafından kullanılabilecek işlevleri dışa aktaran yürütülebilir bir dosyadır. Statik kütüphaneler, DLL'lerin kullanılmasından önce standarttır ve statik kütüphaneler hala mevcuttur, ancak bunlar daha az yaygındır. DLL'leri statik kitaplıklar üzerinde kullanmanın temel avantajı, DLL'ler tarafından kullanılan belleğin çalışan işlemler arasında paylaşılabilmesidir. Örneğin, bir kitaplık iki farklı çalışan işlem tarafından kullanılıyorsa, statik kitaplığın kodu iki kat daha fazla bellek alır, çünkü iki kez belleğe yüklenir. DLL'leri kullanmanın diğer bir büyük avantajı, bir yürütülebilir dosyayı dağıtırken, onları yeniden dağıtmaya gerek duymadan, ana bilgisayar Windows sisteminde olduğu bilinen DLL'leri kullanabilmenizdir. Bu, yazılım geliştiricilerin ve kötü amaçlı yazılım yazarlarının, yazılım dağıtımlarının boyutunu en aza indirmesine yardımcı olur. DLL'ler de kullanışlı bir kod yeniden kullanım mekanizmasıdır. Örneğin, büyük yazılım şirketleri, uygulamalarının çoğunda ortak olan bazı işlevlere sahip DLL'ler oluşturacaktır. Ardından, uygulamaları dağıttıklarında, ana .exe dosyasını ve uygulamanın kullandığı tüm DLL'leri dağıtırlar. Bu, tek bir ortak kod kütüphanesini korumalarını ve yalnızca gerektiğinde dağıtmalarını sağlar.

Başlık (header) kısmının altında, DLL dosyaları neredeyse tam olarak .exe dosyaları gibi görünmektedir. DLL'ler PE dosya biçimini kullanır ve yalnızca tek bir bayrak dosyanın bir DLL olduğunu ve bir .exe olmadığını belirtir. DLL'ler genellikle daha fazla dışa aktarma yapar ve genellikle daha az içe aktarma işlemi yapar. Bunun dışında bir DLL ve bir .exe arasında gerçek bir fark yoktur. Ana DLL işlevi DllMain'dir. Etiketi yoktur ve DLL dosyasında dışa aktarma değildir, ancak PE başlığında dosyanın giriş noktası olarak belirtilir. Bir işlem kitaplığı yüklediğinde veya kaldırdığında, yeni bir iş parçacığı oluşturduğunda veya varolan bir iş parçacığını tamamladığında, işlev DLL'e bildirmek için çağrılır. Bu bildirim, DLL'nin işlem başına veya iş parçacığı başına kaynakları yönetmesini sağlar. Çoğu DLL'de iş parçacığı başına kaynaklar yoktur ve iş parçacığı etkinliğinin neden olduğu DLLMain çağrıları yoksayılır. Bununla birlikte, DLL iş parçacığı başına yönetilmesi gereken kaynaklara sahipse, bu kaynaklar DLL'in amacına ilişkin bir analiste ipucu verebilmektedir. Zararlı yazılımlar DLL'leri 3 şekilde kullanmaktadırlar; kötü niyetli kodun saklanması, temel Windows dll'lerinin kullanımı, üçüncü taraf dll kullanımı.

İlk akla gelen yöntem kötü niyetli kodun saklanmasıdır. Bazen, kötü amaçlı yazılım yazarları, kötü amaçlı kodu bir .exe dosyasında değil, bir DLL dosyasında saklamayı daha avantajlı bulmaktadır. Bazı kötü amaçlı yazılımlar diğer işlemlere eklenir, ancak her işlem yalnızca bir .exe dosyası içerebilir. Kötü amaçlı yazılım bazen kendini başka bir işleme yüklemek için DLL kullanır.

Bir diğer kullanım şekli ise temel Windows DLL'lerinin kullanımıdır. Neredeyse tüm kötü amaçlı yazılımlar her sistemde bulunan temel Windows DLL'lerini kullanır. Windows DLL, işletim sistemi ile etkileşim için gerekli işlevselliği içerir. Kötü amaçlı bir programın Windows DLL'lerini kullanma şekli genellikle kötü amaçlı yazılım analizcisine ilişkin muazzam bir iç görü sunar.

Kötü amaçlı yazılım, diğer programlarla etkileşim kurmak için üçüncü taraf DLL'leri de kullanabilir. Bir üçüncü taraf DLL'sinden işlevleri alan kötü amaçlı yazılım gördüğünüzde, hedeflerine ulaşmak için bu programla etkileşime girdiğini görülmektedir. Örneğin, doğrudan Windows API üzerinden bağlanmak yerine bir sunucuya bağlanmak için Mozilla Firefox DLL kullanabilir. Zararlı yazılım, kurbanın makinesinde yüklü olmayan bir kitaplıktan işlevselliği kullanmak için özelleştirilmiş bir DLL ile de dağıtılabilir; örneğin, bir DLL olarak dağıtılan şifreleme işlevini kullanmak için.

2.2. Zararlı Yazılımlar

Kötü amaçlı yazılımlar veya kötü amaçlı yazılımlar, çoğu bilgisayarda izinsiz giriş ve güvenlik olaylarında rol oynar. Bir kullanıcıya, bilgisayara veya ağa zarar verebilecek herhangi bir yazılım, virüs, truva atı, solucan, kök saklacı aracı, korkutucu yazılımlar ve casus yazılım gibi zararlı yazılım olarak kabul edilebilir. Bu tezde günümüzde kullanılan en yaygın işletim sistemi olan Windows işletim sisteminde bulunan kötü amaçlı yazılımlara odaklanılmaktadır.

2.2.1. Zararlı Yazılımların Sınıflandırılması

- Arka kapı: Saldırganın erişmesine izin vermek için kendisini bir bilgisayara yükleyen kötü amaçlı kod. Arka kapılar genellikle saldırganın bilgisayara kimlik doğrulaması olmadan veya hiç kimlik doğrulama olmadan bağlanmasına izin verir ve yerel sistemde komutları yürütür.
- Zombi Ağları: Arka kapıya benzer şekilde, saldırganın sisteme erişmesine izin verir, ancak aynı zombi ağları ile bulaşmış tüm bilgisayarlar tek bir emret-ve-yönet sunucusundan aynı talimatları alır.
- İndirici: Yalnızca diğer kötü amaçlı kodları indirmek için mevcut olan kötü amaçlı kod. İndiriciler genellikle bir sisteme ilk eriştiğinde saldırganlar tarafından yüklenir. İndirme programı ek zararlı kodları indirip yükleyecektir.
- Bilgi Hırsızlık Yazılımı: Bir kurbanın bilgisayarından bilgi toplayan ve genellikle saldırgana gönderen kötü amaçlı yazılım. Örnek olarak, dinleyici, şifre kırıcı ve klavye tuş basım kayıtçısı gibi zararlılar verilmektedir. Bu kötü amaçlı yazılım genellikle e-posta veya çevrimiçi bankacılık gibi çevrimiçi hesaplara erişmek için kullanılır.
- Çalıştırıcı Kötü amaçlı program, diğer kötü amaçlı programları başlatmak için kullanılır. Genellikle, çalıştırıcılar bir sisteme gizli veya daha fazla erişim sağlamak için diğer kötü amaçlı programları başlatmak için geleneksel olmayan teknikleri kullanırlar.
- Kök Gizleme Aygıtı: diğer kodların varlığını gizlemek için tasarlanmış kötü amaçlı programlardır. Kök gizleme aygıtı genellikle saldırgana uzaktan erişime

izin vermek ve kurbanın tespit etmesini zorlaştırmak için arka kapı gibi diğer kötü amaçlı yazılımlarla eşleştirilir. [36]Gizli görevlerini yerine getirmek için, çekirdek Kök gizleme aygıtları, işletim sisteminin orijinal kontrolünün veya veri akışının kötü amaçlı yazılımın bileşenlerinden herhangi birinin varlığını açığa çıkardığı yerde kontrol akışını veya çekirdeğin sistem çağrılarının veri akışını değiştirmelidir. örneğin, dosyalar) veya çalışan görevlerinden veya eserlerinden herhangi biri (örneğin çekirdek veri yapıları). Bunu yapmak için, Kök gizleme aygıtları tipik olarak kodlarını sistem çağrısı uygulamasının çalıştırma yolu üzerinde bir yere enjekte eder; bu kod kancalarının yerleştirilmesi, Kök gizleme aygıtlarının en öğretici yönlerinden biridir.

- Korkutucu Yazılım: Kötü amaçlı bir yazılımı, virüslü bir kullanıcıyı bir şey satın almaya korkutmak için tasarlanmıştır. Genellikle anti virüs veya diğer güvenlik programları gibi görünmesini sağlayan bir kullanıcı ara yüzü vardır. Kullanıcılarına, sistemlerinde kötü niyetli kodlar olduğunu ve ondan kurtulmanın tek yolunun "yazılımlarını" satın almaları gerektiğini ve gerçekte, sattığı yazılımın başka hiçbir bu ikazı durdurmaktan başka bir iş yapmadığını bildirmektedir.
- Spam Gönderen Zararlı Yazılım: Bir kullanıcının makinesine bulaşan ve daha sonra spam göndermek için bu makineyi kullanan kötü amaçlı yazılım. Bu kötü amaçlı yazılım, saldırganlar için spam gönderme hizmetleri satmalarına izin vererek gelir sağlar.
- Solucan veya virüs Kendini kopyalayabilen ve ek bilgisayarlara bulaştırabilecek kötü amaçlı kod.

Kötü amaçlı yazılımlar genellikle birden fazla kategoriyi kapsar. Örneğin, bir program, şifreleri toplayan bir tuş basış kayıtçısı ve spam gönderen bir solucan bileşenine sahip olabilir. Kötü amaçlı yazılım, saldırganın hedefinin kitlesel veya hedefli olmasına bağlı olarak da sınıflandırılabilir. Korkutucu yazılım gibi toplu kötü amaçlı yazılımlar, av tüfeği yaklaşımını benimser ve mümkün olduğunca çok sayıda makineyi etkileyecek şekilde tasarlanmıştır. Genellikle güvenlik yazılımı hedeflediği için algılanması ve savunulması daha az karmaşık ve kolaydır.

2.2.2. İsimlerine Göre Sınıflandırma

Zararlı yazılımlar genellikle ailelerine göre sınıflandırılırlar. Bu aileler uzmanlar tarafından, üretildikleri yer, davranışlar, kod yapısı benzerliğine göre isimlendirilmişlerdir. Sıfırdan zararlı yazılım yapılması oldukça zordur. O yüzden genellikle zararlı yazılımlar öncekilerin kod yapısı, davranışları ve kaçınma taktiklerini kullanarak ve geliştirilerek oluşturulurlar. Bu tarz benzerlikler aile yapılarını oluşturmaktadır. Bunun dışında zararlı yazılımlar kendi kodlarını değiştirebilir ve başka bir kod yapısı ile aynı davranışı sergileyecek şekilde internette bulunabilmektedirler. Bu tür zararlılar da aynı aile adı ile anılmaktadır.

İsimlerine göre sınıflandırma konusunda problem olduğu [37] çalışmasında da gösterilmiştir. Zararlı yazılımlara ve zararlı yazılımlara verilen etiketler, zararlı yazılım tespitine etkisine bakıldığında, dinamik çalışma zamanında gerçekleşen OpCode'ların dizisi ile tespit modelleri yapılmasına başarım oranını düşürücü etki ettiği görülmüştür.

Tablo 2.4**:** Zararlı yazılım aile tablosu

Aile	Tür
Anset	Solucan
Kazy	Truva Atı
Klizan	Virüs
Zusy	Truva Atı
SoftPulse	Reklam Yazılımı
Yukon	Virüs
Mikey	Truva Atı
Amonetize	Reklam Yazılımı
Eldorado	Truva Atı
Kryptik	Truva Atı
Trivial	Virüs
Nerte	Arka Kapı

2.2.3. Davranış Analizi

Zararlı yazılım davranışlarını daha iyi anlamak için 6 aşamalı olarak incelemek gerekir; bulaşma, aktivasyon, zararlı aktivite, yerleşme, yayılma ve izleri temizleme. Her zararlı yazılımın bir hikayesi ve bu aşamalarda gösterdiği davranışlar vardır. Bu aşamalarda zararlı yazılımlar farklılık göstermektedir. Zararlı yazılım incelemede bu aşamalar davranış değişim noktası olarak ele alınmalıdır ve ayrıntıda boğulup kaybolmamak için farklılıklar bu aşamalara göre incelenmesi büyük kolaylık sağlar.

2.2.3.1. Bulaşma

Kötü amaçlı yazılım, bulaşma davranışını türlerine göre farklılaştırır. Örneğin, virüs yük olarak bulaşırken, solucanlar tam bir uygulama olarak indirilir [38]. Truva atları iyi huylu bir uygulama gibi davranır ve kötü niyetli etkinliklerini yaparlar.

2.2.3.2. Aktivasyon

Bu aşamada türlere göre farklılık gösterebilir. Virüsler kendilerini aktivite edecek başka bir uygulamaya ya da kullanıcının kullanımını beklerken, solucanlar kendi kendilerini bulaştıkları anda aktifleştirebilir ve harekete geçebilir. Tespit edilmemek için hali hazırda kullanılmakta olan iyicil kütüphaneler ya da hizmetlere de enjekte olan zararlılar vardır [2].

Zararlı yazılımlar, dinamik analizden kaçmak için bu aşamada bazı değişiklikler yapabilirler. Aktivasyon öncesi ya da hemen sonrası çalıştığı ortamı analiz etmek, davranış ertelenmesi ya da durum ve zamana göre asıl niyetinden farklı olarak iyicil taklidi yapmak gibi. Statik analizden kaçmak için de bu aşamadaki davranışlar önemlidir. Zararlı dosya kendini şifrelemişse ya da paketlemişse aktivasyon aşamasında açılacaktır. Bunun dışında daha farklı statik analizden kaçınma yoluna başvurmuş olabilir. Zararlı yazılımın kendi kod yapısı tam anlamıyla incelenmesi için bu aşaması iyi tespit edilmeli ve aktivasyondan sonraki kodları ele geçirilmelidir.

2.2.3.3. Zararlı Aktivite

Zararlı davranış genel olarak şöyle açıklanabilir: kullanıcının haberi olmayan ya da bilgisayara zarar veren davranış. Örnek olarak veri çalma, sistem çökertme, reklam çıkarma gibi istenmeyen aktiviteler verilebilir.

Arka kapı, bir saldırgana kurbanın makinesine uzaktan erişim sağlayan bir kötü amaçlı yazılım türüdür. Arka kapılar en yaygın bulunan kötü amaçlı yazılım türüdür ve çok çeşitli yeteneklerle tüm şekil ve boyutlarda gelir. Arka kapı kodu genellikle tam bir özellik kümesi uygular, bu nedenle arka kapı saldırganları kullanırken genellikle ek kötü amaçlı yazılım veya kod indirmesi gerekmez. Arka kapılar, kayıt defteri anahtarlarını değiştirme, görüntüleme pencerelerini numaralandırma, dizin oluşturma, arama dosyaları vb. Gibi ortak bir işlev kümesiyle birlikte gelir. Kullandığı ve içe aktardığı Windows işlevlerine bakarak bu özelliklerden hangilerinin bir arka kapı tarafından uygulanacağını belirleyebilirsiniz.

Uzaktan kabuk bağlantısı (*reverse shell*), virüs bulaşmış bir makineden kaynaklanan ve saldırganların bu makineye kabuk erişimi sağlayan bir bağlantıdır. Uzaktan kabuklar, hem tek başına kötü amaçlı yazılım hem de daha karmaşık arka kapıların bileşenleri olarak bulunur. Popüler olarak zararlı yazılımlar üstünden 3 çeşit uzak bağlantı kurulmaktadır: netcat, Windows API, RAT.

Zombi Ağları, zombiler olarak bilinen ve genellikle bir botnet denetleyicisi olarak bilinen bir sunucunun kullanılmasıyla tek bir varlık tarafından kontrol edilen, güvenliği ihlal edilmiş ana bilgisayar koleksiyonudur. Bir zombi ağının amacı, ek kötü amaçlı yazılım veya spam yaymak için kullandığı büyük bir zombi ağı oluşturmak veya dağıtılmış bir hizmet reddi (DDoS) saldırısı oluşturmak için mümkün olduğunca çok sayıda ana bilgisayardan ödün vermektir.

Hırsızlık olarak bilinen saldırılarda saldırganlar genellikle üç şekilde kötü amaçlı yazılımla kimlik bilgilerini çalmaktadırlar: Bir kullanıcının kimlik bilgilerini çalmak için oturum açmasını bekleyen programlar, Windows'ta depolanan ve parola karmaları gibi bilgileri döken Programlar, doğrudan kullanılmak üzere veya çevrimdışı kırık, tuş vuruşlarını kaydeden programlar. Windows karmalarını boşaltmak, kötü amaçlı yazılımların sistem kimlik bilgilerine erişmesi için popüler bir yoldur. Saldırganlar, onları çevrimdışı kırmak veya karma geçiş saldırısında kullanmak için bu karmaları yakalamaya çalışır. Karma geçiş saldırısı, oturum açmak için düz metin parola almak üzere karmaların şifresini çözmeye veya kırmaya gerek kalmadan uzak bir ana bilgisayarda kimlik doğrulaması yapmak için LM ve NTLM karmalarını kullanır (NTLM kimlik doğrulaması kullanarak). Tuş basış kaydı, kimlik

bilgisi çalmanın klasik bir şeklidir. Tuş basış kaydı yaparken, kötü amaçlı yazılım tuş vuruşlarını kaydeder, böylece bir saldırgan kullanıcı adları ve şifreler gibi yazılı verileri gözlemleyebilir. Windows kötü amaçlı yazılımları birçok tuş basış kayıt biçimi kullanır: Çekirdek Tabanlı Tuş basış kayıtçıları, Kullanıcı Alanı Tuş basış kayıtçıları, Dizeler Listelerindeki Tuş basış kayıtçıları belirleme

2.2.3.4. Yerleşme

Bulaştığı bilgisayarda kendini saklama, izlerini silme ve herhangi bir silinme çabası oluştuğunda kendini tekrar başlatma işlemlerinin genel adıdır. Zararlı yazılımlar bulaştığı bilgisayarda kalıcı olmak isterler. Bu davranış, kalıcılık olarak bilinir. Kalıcılık mekanizması yeterince benzersizse, belirli bir kötü amaçlı yazılım parçasını parmak izi için harika bir yol olarak bile hizmet edebilir. en sık elde edilen kalıcılık yöntemi sistem kayıt defterinin değiştirilmesidir. Kötü amaçlı yazılım yazarları, AppInit_DLL adlı özel bir kayıt defteri konumu aracılığıyla DLL'leri için kalıcılık kazanabilir. AppInit_DLLs User32.dll yüklenen her işlem yüklenir ve kayıt defterine basit bir ekleme AppInit_DLLs kalıcı olur. Kötü amaçlı yazılım yazarları, oturum açma, oturum kapatma, başlatma, kapatma ve kilit ekranı gibi belirli bir Winlogon etkinliğine kötü amaçlı yazılım bağlayabilir. Bu, kötü amaçlı yazılımın güvenli modda yüklenmesine bile izin verebilir.

Tüm hizmetler kayıt defterinde tutulmaktadır ve kayıt defterinden kaldırılırsa hizmet başlatılmaz. Kötü amaçlı yazılım genellikle bir Windows hizmeti olarak yüklenir, ancak genellikle yürütülebilir bir dosya kullanır. Bir svchost.exe DLL dosyası olarak kalıcılık için kötü amaçlı yazılım yüklemek, kötü amaçlı yazılımın işlem listesine ve kayıt defterine standart bir hizmetten daha iyi uyum sağlamasına neden olur. Svchost.exe, DLL'lerden çalışan hizmetler için genel bir ana bilgisayar işlemidir ve Windows sistemlerinde genellikle aynı anda çalışan birçok svchost.exe örneği vardır. Her biri birtakım hizmetleri hizmet grubu olarak içerir. Svchost'un görevi Hizmetlerin geliştirilmesi, test edilmesi ve çalıştırılmasında kolaylık sağlamaktır. Zararlı yazılımlar genellikle bu hizmet gruplarından birine girmeyi amaçlamaktadırlar. Böylece fark edilmeleri ve silinmeleri zorlaşır. *Netsvcs* ve *CreateServiceA* API fonksiyonları kullanılarak bu gerçekleştirilir. Kötü amaçlı yazılımın kalıcılık kazanmasının bir başka yolu da sistem ikili dosyalarını Truva atı yerleştirilmesidir. Bu teknikle, kötü amaçlı yazılım, etkilenen ikili dosyayı bir

sonraki çalıştırıldığında veya yüklendiğinde sistemi kötü amaçlı yazılımı yürütmeye zorlamak için bir sistem ikili dosyasının baytlarını yamalar. Zararlı yazılım yazarları genellikle normal Windows işleminde sık kullanılan bir sistem ikili dosyasını hedefler. DLL'ler popüler bir hedeftir. Diğer teknikler; DLL Yükleme Sırası Ele Geçirme, yetki Yükselmesi ve *SeDebugPrivilege* Kullanımıdır.

Enjeksiyon: En popüler gizli çalıştırma tekniği proses enjeksiyonudur. Adından da anlaşılacağı gibi, bu teknik kodu başka bir çalışan işleme enjekte eder ve bu işlem farkında olmadan kötü amaçlı kodu yürütür. En yaygın olarak yapılan DLL enjeksiyonudur. Uzak bir işlemin kötü amaçlı bir DLL yüklemeye zorlandığı bir işlem enjeksiyon biçimi olan DLL enjeksiyonu, en yaygın kullanılan gizli yükleme tekniğidir [99]. DLL enjeksiyonu, *LoadLibrary'yi* çağıran uzak bir sürece kod enjekte ederek çalışır ve böylece bir DLL'yi bu işlem bağlamında yüklenmeye zorlar. Güvenliği ihlal edilen işlem kötü amaçlı DLL dosyasını yükledikten sonra, işletim sistemi otomatik DLL'nin *DllMain* işlevini çağırır. Bu fonksiyon zararlı DLL yazarının yazdığı fonksiyondur. Kötü amaçlı DLL dosyasını bir ana bilgisayar programına enjekte etmek için, başlatıcı kötü amaçlı yazılımın önce mağdur sürecine bir tanıtıcı alması gerekir. En yaygın yol, işlem hedefini enjeksiyon hedefi için aramak için *createToolhelp32Snapshot, Process32First* ve *Process32Next* Windows API çağrılarını kullanmaktır. *CreateRemoteThread* işlevi, başlatıcı kötü amaçlı yazılımların uzak bir işlemde yeni bir iş parçacığı oluşturmasına ve yürütmesine izin vermek için DLL enjeksiyonu için yaygın olarak kullanılır. *CreateRemoteThread* kullanıldığında, üç önemli parametreden geçirilir: enjekte edilen iş parçacığının başlangıç noktası (*lpStartAddress*) ve o iş parçacığı için bir argüman (*lpParameter*) ile birlikte *OpenProcess* ile elde edilen işlem tanıtıcısı (*hProcess*). Örneğin, başlangıç noktası *LoadLibrary* olarak ayarlanmış ve kötü amaçlı DLL adı bağımsız değişken olarak geçirilmiş olabilir. Bu, *LoadLibrary*'nin kötü amaçlı DLL'nin bir parametresiyle mağdur işleminde çalıştırılmasını tetikler ve böylece DLL'nin mağdur işlemine yüklenmesine neden olur.

Kötü amaçlı yazılım yazarları, genellikle kötü amaçlı kitaplık adı dizesi için alan oluşturmak için *VirtualAllocEx* kullanır. *VirtualAllocEx* işlevi, bu işleme bir tanıtıcı sağlanmışsa, uzak bir işlemde yer ayırır. *CreateRemoteThread* çağrılmadan önce gereken son kurulum işlevi *WriteProcessMemory*'dir. Bu işlev, kötü amaçlı kitaplık adı dizesini *VirtualAllocEx* ile ayrılan bellek alanına yazar.

Bir diğer yol ise direk enjeksiyondur. DLL enjeksiyonu gibi, doğrudan enjeksiyon da uzak bir işlemin bellek alanına kod tahsis edilmesini ve eklenmesini içerir. Doğrudan enjeksiyon, DLL enjeksiyonuyla aynı Windows API çağrılarının çoğunu kullanır. Fark, ayrı bir DLL yazmak ve uzak işlemi onu yüklemeye zorlamak yerine, doğrudan zararlı yazılımın kötü amaçlı kodunu doğrudan uzak sürece enjekte etmesidir. Doğrudan enjeksiyon DLL enjeksiyonundan daha esnektir, ancak ana bilgisayar işlemini olumsuz etkilemeden başarılı bir şekilde çalışmak için çok sayıda özelleştirilmiş kod gerektirir. Bu teknik, derlenmiş kodu enjekte etmek için kullanılabilir, ancak daha sıklıkla kabuk kodunu enjekte etmek için kullanılır. Doğrudan enjeksiyon durumunda yaygın olarak üç işlev bulunur: *VirtualAllocEx, WriteProcessMemory* ve *CreateRemoteThread*. *VirtualAllocEx* ve *WriteProcessMemory* için genellikle iki çağrı olur. Birincisi uzak evre tarafından kullanılan verileri tahsis eder ve yazar ve ikincisi uzak evre kodunu tahsis eder ve yazar. *CreateRemoteThread* çağrısı uzak iş parçacığı kodunun (lpStartAddress) ve verilerin (lpParameter) konumunu içerecektir. Uzak iş parçacığı tarafından kullanılan veri ve işlevlerin mağdur sürecinde bulunması gerektiğinden, normal derleme prosedürleri çalışmaz. Örneğin, dizeler normal .data bölümünde değildir ve önceden yüklenmemiş işlevlere erişmek için *loadLibrary* / *GetProcAddress* öğesinin çağrılması gerekir. Başka kısıtlamalar da mevcuttur. Temel olarak, doğrudan enjeksiyon, yazarların ya yetenekli montaj dili kodlayıcıları olmasını ya da sadece nispeten basit bir kabuk kodu enjekte etmelerini gerektirir.

İşlem Değiştirme yönteminde bazı kötü amaçlı yazılımlar, bir ana bilgisayar programına kod enjekte etmek yerine, çalışan bir işlemin bellek alanının üzerine kötü amaçlı yürütülebilir bir dosyanın üzerine yazmak için işlem değiştirme olarak bilinen bir yöntem kullanır. İşlem değiştirme, bir kötü amaçlı yazılım yazarı, işlem enjeksiyonunu kullanarak bir işlemi çökme riski olmadan kötü amaçlı yazılımı meşru bir işlem olarak gizlemek istediğinde kullanılır. Bu teknik, kötü amaçlı yazılıma değiştirdiği işlemle aynı ayrıcalıkları sağlar. Örneğin, bir kötü amaçlı yazılım parçası svchost.exe'ye bir işlem değiştirme saldırısı gerçekleştirecek olsaydı, kullanıcı svchost.exe'nin C: \ Windows \ System32 adresinden çalışan bir işlem adı görür ve muhtemelen hiçbir şey düşünmezdi. İşlem oluşturulduktan sonra, bir sonraki adım, genellikle parametre olarak geçirilen bir bölümün işaret ettiği tüm belleği serbest bırakmak için *ZwUnmapViewOfSection* kullanarak kurban işleminin belleğini kötü amaçlı yürütülebilir dosya ile değiştirmektir. Bellek eşleştirildikten sonra, yükleyici

kötü amaçlı yazılım için yeni bellek ayırmak için *VirtualAllocEx* gerçekleştirir ve kötü amaçlı yazılım bölümlerinin her birini genellikle döngü içinde olmak üzere kötü amaçlı yazılım bölümlerine yazmak için *WriteProcessMemory* kullanır. Son adımda, kötü amaçlı yazılım kurbanın işlem ortamını geri yükler, böylece kötü amaçlı kod *SetThreadContext*'i çağırarak giriş noktasını kötü amaçlı koda işaret edecek şekilde ayarlar. Son olarak, kurban sürecinin yerini alan kötü amaçlı yazılımı başlatmak için *ResumeThread* çağrılır.

Kanca enjeksiyonu, uygulamalar için hedeflenen iletileri yakalamak için kullanılan Windows kancalarından yararlanan kötü amaçlı yazılım yüklemenin bir yolunu tanımlar. Kötü amaçlı yazılım yazarları, iki şeyi gerçekleştirmek için kanca enjeksiyonunu kullanabilir: kesme kodu geldiğinde kötü amaçlı kodun çalışacağından emin olmak için, ihtiyaç duyulan zararlı DLL'nin kurban işleminin bellek alanına yükleneceğinden emin olmak için.

Kanca enjeksiyonu, tuş vuruşlarını kaydeden tuş kaydedici olarak bilinen kötü amaçlı uygulamalarda sıklıkla kullanılır. Tuş vuruşları, sırasıyla WH_KEYBOARD veya WH_KEYBOARD_LL kanca prosedürü tipleri kullanılarak yüksek veya düşük seviyeli kancalar kaydedilerek yakalanabilir. WH_KEYBOARD prosedürleri için, kanca genellikle uzak bir işlem bağlamında çalışır, ancak kancayı takan işlemde de çalışabilir. WH_KEYBOARD_LL yordamları için, olaylar doğrudan kancayı takan işleme gönderilir, böylece kanca kancayı oluşturan işlem bağlamında çalışır. Kanca tiplerinden birini kullanarak, bir tuş basış kaydedici tuş vuruşlarını yakalayabilir ve bunları bir dosyaya kaydedebilir veya işleme veya sisteme geçirmeden önce değiştirebilir. Uzaktan Windows kancasını gerçekleştirmek için kullanılan temel işlev çağrısı *SetWindowsHookEx*'dir.

2.2.3.5. Yayılma

Zararlı yazılımlar etkilerini arttırmak için ağdaki diğer makinalara ya da iletişim veya bağlantı kurulan herhangi başka bir bilgisayara bulaşma eğilimindedirler. İnternetten başka bir zararlı kod parçası indirerek senaryo dahilinde bir zararlı aktivite başlatabilirler. Ya da başka bir zararlı yazılımın tetiklenmesi veya çalıştırılması için kullanılmaktadırlar. Bu tarz davranışlara indirici/çalıştırıcı denmektedir. Bu davranışlar genellikle *URLDownloadtoFileA* veya *WinExec* fonksiyon çağrımları etraflı kod parçaları ve API çağrım dizilimleri ile

yapılmaktadır. İyicil olarak sıklıkla kullandığımız yazılımlarda da bu davranışa rastlanmaktadır. Hedef alınarak yapılan saldırılarda, hedef bilgisayarda zafiyet tespit edilip indirici/çalıştırıcı kod parçası enjekte edilerek zarar verecek bir senaryo başlatılır. Bir başka zararlı aktivite ise fırlatıcı/çalıştırıcı (launcher) davranışıdır. Bu tür yazılımlar genellikle iyicil gibi gözükürken içlerinde zararlı yazılım kodlarını barındırır ve fırsatı bulduğunda bu yazılımı çalıştıkları bilgisayara yükler. Kaynak bölümü (rsrc section) sıkıştırılmış veya şifrelenmişse, kötü amaçlı yazılım yüklenmeden önce kaynak bölümü çıkarma işlemini gerçekleştirmelidir. Bu genellikle başlatıcıyı *FindResource, LoadResource* ve *SizeofResource* gibi kaynak işleme API işlevlerini kullandığını göreceğiniz anlamına gelmektedir. Zararlı yazılım başlatıcıları, genellikle bu ayrıcalıklara sahip olmak için yönetici ayrıcalıklarıyla çalıştırılmalı veya kendilerini yükseltmelidir.

2.2.3.6. İzleri Silme

Kötü amaçlı yazılımlar genellikle çalışan işlemlerini ve kalıcılık mekanizmalarını kullanıcılardan gizlemek için büyük çaba harcar. Kötü amaçlı etkinlikleri gizlemek için kullanılan en yaygın araç Kök gizleme aygıtı olarak adlandırılır [36]. Kök gizleme aygıtları birçok biçimde olabilir, ancak çoğu işletim sisteminin iç işlevlerini değiştirerek çalışır. Bu değişiklikler dosyaların, işlemlerin, ağ bağlantılarının veya diğer kaynakların diğer programlara görünmez olmasına neden olur. Bazı Kök gizleme aygıtları kullanıcı alanı uygulamalarını değiştirir, ancak çoğunluk çekirdeği değiştirir, çünkü saldırı önleme sistemleri gibi koruma mekanizmaları çekirdek düzeyinde kurulur ve çalışır. Hem Kök gizleme aygıtı hem de savunma mekanizmaları, çekirdek düzeyinde çalıştırıldıklarında kullanıcı düzeyinde değil, daha etkilidir. Çekirdek düzeyinde, Kök gizleme aygıtları sistemi kullanıcı seviyesinden daha kolay bozabilir. SSDT (SSDT kancalama) ve IRP kancalarının, IAT kancalama ve satır içi kancalama çekirdek modunda başlıca izleri silme teknikleridir.

Kök gizleme aygıtları, varlıklarını gizlemek için işletim sisteminin iç işlevlerini değiştirir. Bu değişiklikler dosyaları, işlemleri, ağ bağlantılarını ve diğer kaynakları çalışan programlardan gizleyerek anti virüs ürünlerinin, yöneticilerinin ve güvenlik analistlerinin kötü amaçlı etkinlikleri keşfetmesini zorlaştırabilir. Kullanılan Kök gizleme aygıtların çoğu bir şekilde çekirdeği değiştirerek çalışır. Kök gizleme

aygıtları çok çeşitli teknikler kullanabilmesine rağmen, pratikte bir teknik diğerlerinden daha fazla kullanılır: Sistem Hizmeti Tanımlayıcı Tablo kancalama (System Service Descriptor Table hooking). Bu teknik birkaç yaşındadır ve diğer Kök gizleme aygıtı tekniklerine göre tespit edilmesi kolaydır. Bununla birlikte, hala kötü amaçlı yazılım tarafından kullanılmaktadır, çünkü anlaşılması kolay, esnek ve uygulanması kolaydır.

Kesinti, işlemciye derhal ilgilenilmesi gereken bir olayı gösteren bir giriş sinyalidir [24]. Bir kesme sinyali işlemciyi uyarır ve işlemcinin o anda yürütülen kodu kesmesi için bir istek olarak hizmet eder, böylece olay zamanında işlenebilir. Kesmeler bazen kod yürütmek için sürücüler veya Kök gizleme aygıt'ları tarafından kullanılır [36]. Bir sürücü *IoConnectInterrupt'ı* belirli bir kesme kodu için bir işleyici kaydetmek üzere çağırır ve daha sonra kesme kodunun her oluşturulduğu zaman işletim sisteminin arayacağı bir kesme servisi rutini (ISR) belirtir. Kesme Tanımlayıcı Tablosu (IDT), !*İdt* komutuyla görüntüleyebileceğiniz ISR bilgilerini depolar.

Tablo 2.5: Davranışların gerçekleştirilebilmesi için API dizi tablosu

Kötü Niyetli Aktivite	API çağrım dizisi
DLL enjeksiyonu	CreateRemoteThread OpenProcess, VirtualAllocEx,WriteProcessMemory, CreateRemoteThread
IAT Kancalama	LoadLibrary, (strcmp, strncmp, stricmp, strnicmp), VirtualProtect
Hata Ayıklayıcı Engelleme	IsDebuggerPresent, CheckRemoteDebuggerPresent, OutputDebugStringA, OutputDebugStringW
Ekran Yakalama	GetDC, GetWindowDC, CreateCompatibleDC, CreateCompatibleBitmap, SelectObject, BitBlt, WriteFile

2.2.4. Gizlenme Taktikleri

Yazılımın analizini zorlaştırma, asıl fonksiyonelliklerin görülmesini engelleme ve zararlı yazılım tespit sistemlerine yakalanmamak için uygulanan taktiklerdir. Bu kısım 4 başlık altında anlatılacaktır; statik analizden kaçma, dinamik analizden kaçma, paketleme ve kendini değiştirme yöntemleri.

2.2.4.1. Statik Analizden Kaçınma

Anti-statik analiz tekniklerinin temel amacı, bir analistin, programı gerçekten çalıştırmadan programın doğasını anlamasını önlemektir [39]. Bunlar, IDA, distorm3 gibi çözücüleri hedef alan tekniklerdir. Burada çeşitli anti-statik analiz teknikleri tartışılmaktadır.

- Çözücü Engelleme: Kötü amaçlı yazılım yazarları, kötü amaçlı kodların analizini geciktirmek veya önlemek için sökme önleme tekniklerini kullanır [19]. Başarıyla yürütülen herhangi bir kod tersine mühendislikle değiştirilebilir, ancak kodlarını sökme ve takma önleme teknikleriyle kodlayarak, kötü amaçlı yazılım yazarları kötü amaçlı yazılım analizcisinin ihtiyaç duyduğu beceri düzeyini artırır.
- Sökme önleme teknikleri: sökme işlemlerinin varsayımlarından ve sınırlamalarından yararlanarak çalışır [39]. Örneğin, sökücüler bir programın her bir baytını bir seferde yalnızca bir talimatın parçası olarak temsil edebilir. Sökme cihazı yanlış ofsette sökme işlemine kandırılırsa, geçerli bir talimat görünümden gizlenebilir.
- Jump Komutu ile Aynı Hedefe Zıplama: Vahşi doğada görülen en yaygın sökme önleme tekniği, her ikisi de aynı hedefe işaret eden iki arka arkaya koşullu atlama talimatıdır [40]. Örneğin, bir jz loc_512, jnz loc_512 tarafından takip edilirse, loc_512 konumu her zaman atlanacaktır. Jz'nin jnz ile birleşimi, aslında, koşulsuz bir jmp'dir, ancak sökme aygıtı, bir seferde yalnızca bir talimatı sökmediği için bunu tanımıyor. Çözücü, jnz ile karşılaştığında, pratikte asla yürütülmeyecek olmasına rağmen, bu talimatın sahte bölümünü sökmeye devam eder.
- Sabit Koşul ile Jump komutu Dallanması: Genel olarak vahşi doğada bulunan başka bir çözücü önleme tekniği, koşulun her zaman aynı olacağı şekilde

yerleştirilmiş tek koşullu bir atlama komutundan oluşur [41]. Daha önce de tartışıldığı gibi, çözücüler ilk önce sahte dalı işleyecek, gerçek dal ile çakışan kod üretecek ve önce sahte dalı işlediği için o dalı daha fazla güvenecektir.

- İmkânsız Çözücü Taktiği: Tartıştığımız basit çözücü önleyici teknikler, koşullu bir atlama komutundan sonra stratejik olarak yerleştirilmiş bir veri baytı kullanır; bu bayt'da başlayan çözülme, izlenen gerçek komutun çözülmesini önleyeceği düşüncesiyle yerleştirilir. Çok baytlı bir komuttur. Buna haydut bayt denir çünkü programın bir parçası değildir ve sadece sökücüyü atmak için koddadır. Bu örneklerin hepsinde, haydut bayt ihmal edilebilir. Ama ya haydut bayt göz ardı edilemezse? Ya çalışma zamanında gerçekten yürütülen meşru bir talimatın parçasıysa? Burada, herhangi bir baytın yürütülen çoklu komutların bir parçası olabileceği zor bir senaryo ile karşı karşıyayız. Şu anda piyasada bulunan hiçbir çözücü cihazı, iki komutun parçası olarak tek bir baytı temsil etmemektedir, ancak işlemcinin böyle bir sınırı yoktur [19].

2.2.4.2. Dinamik Analizden Kaçma

Zararlı yazılım davranışlarını ve çalıştıkları bilgisayarda yarattıkları etkilere bakmak için dinamik analiz yapılmaktadır. Bu kapsamda zararlı yazılım güvenli ve izlenebilir bir ortamda çalıştırılır. Genel olarak 3 önemli dinamik analiz yöntemi vardır; hata ayıklayıcı, kum havuzu ve etiketlenmiş veri analizidir. Zararlı yazılımlar bu kapsamda dinamik analizden kaçmak için sanal ortamda çalıştığını tespit etmeye veya hata ayıklayıcı kullanıldığını tespit etmeye çalışır. Eğer tespit ederse zararlı davranışlarını göstermekten kaçınır ya da kendilerini durdururlar.

Kötü amaçlı yazılım, Windows API kullanımı, yapıtları ayıklamak için bellek yapısını manuel olarak kontrol etmek ve sistemde bir hata ayıklayıcı tarafından kalan kalıntıları aramak dahil olmak üzere, bir hata ayıklayıcının eklendiğini göstermek için çeşitli teknikler kullanır. Hata ayıklayıcı algılaması, kötü amaçlı yazılımın hata ayıklama işlemini gerçekleştirmesinin en yaygın yoludur. Hata ayıklayıcıyı tespit etmenin birkaç yolu vardır.

Windows API işlevlerinin kullanımı, hata ayıklama önleme tekniklerinden en belirgin olanıdır. Windows API [42], program tarafından hata ayıklanıp kaynaklanmadığını belirlemek için kullanılabilecek çeşitli işlevler sunar. Bu fonksiyonlardan bazıları hata ayıklayıcı tespiti için tasarlanmıştır; diğerleri farklı

amaçlar için tasarlandı, ancak bir hata ayıklayıcısını tespit etmek için yeniden konumlandırılabilir. Bu işlevlerden birkaçı, API'de belgelenmeyen işlevleri kullanır.

Bir hata ayıklayıcıyı tespit etmek için zamanlama kontrollerini kullanmanın birkaç yolu vardır:

- Bir zaman damgası kaydedilmeli, birkaç işlem yapılmalı, başka bir zaman damgası alınmalı ve sonra iki zaman damgasını karşılaştırılmalıdır. Bir gecikme varsa, bir hata ayıklayıcının varlığını varsayılır.
- Bir istisna oluşturmadan önce ve sonra bir zaman damgası alınır. Bir işlem hata ayıklanmıyorsa, istisna gerçekten hızlı bir şekilde ele alınacaktır; Bir hata ayıklayıcı, istisnayı çok daha yavaş ele alır. Varsayılan olarak, çoğu hata ayıklayıcı, olağanüstü gecikmelere neden olan istisnaları ele almak için insan müdahalesi gerektirir. Birçok hata ayıklayıcı, istisnaları görmezden gelmenize ve bunları programa geçirmenize izin verirken, bu gibi durumlarda yine de büyük bir gecikme yaşanacaktır.

En yaygın zamanlama kontrol yöntemi, son sistemin EDX: EAX'a yerleştirilen 64-bit bir değer olarak yeniden başlatılmasından bu yana kenelerin sayısını geri döndüren rdtsc komutunu (OpCode 0x0F31) kullanır. Kötü amaçlı yazılım, bu komutu yalnızca iki kez uygular ve iki okuma arasındaki farkı karşılaştırır.

Hata ayıklayıcının engellenmesi zamanlama kontrolü yapmak için rdtsc gibi iki Windows API işlevi kullanılır. Bu yöntem, işlemcilerin yüksek çözünürlüklü performans sayaçlarına (işlemcide gerçekleştirilen etkinlik sayısını saklayan kayıtlara) sahip olmasına dayanır. *QueryPerformanceCounter*, bir karşılaştırmada kullanım için zaman farkı elde etmek amacıyla bu sayacı iki kez sorgulamak için çağrılabilir. İki arama arasında çok fazla zaman geçmişse, bir hata ayıklayıcının kullanıldığı varsayılmaktadır.

Bu tekniklerle kötü amaçlı yazılım, sanal bir makinede çalıştırılıp çalıştırılmadığını tespit etmeye çalışır. Sanal bir makine algılanırsa, farklı davranabilir veya basitçe çalışmayabilir. Bu, elbette, analist için sorunlara neden olmaktadır.

Sanal Ortam Gereçleri: Sanal makine ortamı, özellikle araçlar kurulduğunda, sistem üzerinde birçok esere neden olur [43]. Kötü amaçlı yazılım, sanal makineyi tespit etmek için dosya sisteminde, kayıt defterinde ve işlem listesinde bulunan bu

eserleri kullanmaktadır. Bu kodu daha fazla analiz ederek, işlem listesini, *ListToolhelp32Snapshot*, *Process32Next* ve benzeri fonksiyonlarla taradığını fark ediyoruz. At konumundaki strncmp, işlem adının işlem listesinde olup olmadığını belirlemek için, VmwareTray.exe dizesini processentry32.szExeFile öğesini ASCII'ye dönüştürmenin sonucu ile karşılaştırıyor. İşlem listesinde VmwareTray.exe bulunursa, program derhal sonlandırılır

Hafıza Gereçlerinin Kontrolü: sanal makine yazılımı, sanallaştırma işleminin bir sonucu olarak birçok eseri bellekte bırakır. Bazıları, sanal bir makinede taşınmaları veya değiştirilmeleri nedeniyle tanınabilir ayak izleri bırakan kritik işlemci yapılarıdır. Bellek yapılarını tespit etmek için yaygın olarak kullanılan bir teknik, sanal makine yazılım dizisi için fiziksel bellek üzerinde yapılan ve yüzlerce örneği tespit edebileceğini tespit ettiğimiz bir tekniktir [43].

Kırmızı Hap, *IDTR* kaydının değerini almak için sid komutunu uygulayan bir sanal makine engelleme tekniğidir [44]. Sanal makine monitörü, sunucunun *IDTR*'si ile çakışmamak için konuğun *IDTR*'sini yeniden yerleştirmelidir. Sanal makine sid komutunu çalıştırdığında sanal makine monitörü bildirilmediğinden, sanal makine için *IDTR* döndürülür. Kırmızı Hap, sanal makine yazılım kullanımını saptamak için bu tutarsızlığı test eder.

Halen kullanılmakta olan en popüler sanal makine engelleme tekniği I / O iletişim portunu sorgulamaktır [44]. Bu teknik, Storm solucanı ve Phatbot gibi solucan ve botlarda sıkça karşılaşılır.

Str komutu, bölüm seçiciyi, o anda çalışmakta olan görevin görev durumu segmentine (TSS) işaret eden görev kayıt defterinden alır. Zararlı yazılım yazarları, bir sanal makinenin varlığını tespit etmek için str komutunu kullanabilir, çünkü talimat tarafından döndürülen değerler, yerel makineye göre sanal makinede farklılık gösterebilir [1]. Bu teknik çok işlemcili donanımda çalışmamaktadır.

Zararlı yazılımlar sanal makine algılama gerçekleştirmediği sürece bu talimatları çalıştırmaz ve bu algılamadan kaçınmak, bu talimatları çağırmaktan kaçınmak için ikili dosya ekleme kadar kolay olabilir. Bu talimatlar, kullanıcı modunda yürütüldüğünde temel olarak kullanışsızdır, bu nedenle onları görürseniz, sanal makine engelleyici kodunun bir parçası olabilirler [45].

2.2.4.3. Paketleyiciler

Paketleyici olarak bilinen paket programları, kötü amaçlı yazılımın virüsten koruma yazılımından gizlenmesini, kötü amaçlı yazılım analizini karmaşıklaştırmasını ve kötü amaçlı bir yürütülebilir dosyanın boyutunu küçültmesini sağladığı için kötü amaçlı yazılım yazarları arasında oldukça popüler hale gelmiştir. Çoğu paketleyicinin kullanımı kolaydır ve serbestçe kullanılabilir. Temel statik analiz, paketlenmiş bir program için kullanışlı değildir; Paketlenmiş kötü amaçlı yazılımlar, statik olarak analiz edilmeden önce açılmalıdır, bu da analizi daha karmaşık ve zorlu hale getirir.

Paketleyiciler çalıştırılabilir ürünlerde iki ana nedenden ötürü kullanılır: programları küçültmek veya algılama veya analizi engellemek için. Çok çeşitli paketleyiciler olmasına rağmen, hepsi benzer bir modeli izlerler: Dönüştürülmüş yürütülebilir dosyayı veri olarak depolayan ve işletim sistemi tarafından adlandırılan bir açma saplaması içeren yeni bir yürütülebilir dosya oluşturmak için bir yürütülebilir dosyayı dönüştürürler.

Tüm paketleyiciler, yürütülebilir bir dosyayı girdi olarak alır ve çıktı olarak yürütülebilir bir dosya üretir. Paketlenmiş çalıştırılabilir kod sıkıştırılır, şifrelenir veya başka şekilde dönüştürülür, bu sayede tanınması ve tersine mühendislik yapılması zorlaşır. Çoğu paketleyici, orijinal yürütülebilir dosyayı sıkıştırmak için bir sıkıştırma algoritması kullanır. Dosyanın analiz edilmesini zorlaştırmak için tasarlanmış bir paketleyici, orijinal çalıştırılabilir dosyayı şifreleyebilir ve çözücü önleyici, hata ayıklama önleyici veya sanal makine engelleyici gibi geriye dönük mühendislik önleme teknikleri kullanabilir. Paketleyiciler, tüm veriler ve kaynak bölümleri dahil olmak üzere tüm yürütülebilir dosyaları paketleyebilir veya yalnızca kod ve veri bölümlerini paketleyebilir. Orijinal programın işlevselliğini korumak için bir paket programın içe aktarma bilgilerini kaydetmesi gerekir. Bilgiler herhangi bir formatta saklanabilir ve bu bölümde daha sonra ayrıntılı olarak ele alınan birkaç ortak strateji vardır. Bir programı paketten çıkarırken, içe aktarma bölümünü yeniden oluşturmak bazen zorlu ve zaman alıcı olabilir, ancak programın işlevselliğini analiz etmek için gereklidir.

Kötü amaçlı yazılımları analiz ederken ilk adım, paketlendiğini kabul etmektir. Kötü amaçlı yazılımın daha önceki bölümlerde paketlenip paketlenmediğini tespit etmek için teknikler ele aldık. Burada bir inceleme yapacağız ve ayrıca yeni bir teknik tanıtacağız. Aşağıdaki listede, kötü amaçlı yazılımların paketlenip paketlenmediğini belirlerken aranacak işaretleri özetlenmiştir.

- Programın çok az içe aktarımı var ve özellikle de yalnızca içe aktarılan *LoadLibrary* ve *GetProcAddress* ise.
- Program çözücüde açıldığında, otomatik analiz tarafından yalnızca az miktarda kod tanınır.
- Program dinamik hata ayıklayıcıda açıldığında, programın paketlenebileceği konusunda bir uyarı vardır.
- Program, belirli bir paketleyiciyi (UPX0 gibi) belirten bölüm adlarını gösterir.
- Program, 0 Ham Veri Boyutu ve sıfır olmayan Sanal Boyut gibi bir .text bölümü gibi anormal bölüm boyutlarına sahiptir.
- PeiD gibi paket tespit araçları da bir çalıştırılabilir paket olup olmadığını belirlemek için kullanılabilir.

Entropi Hesaplama, paketlenmiş çalıştırılabilirler, entropi hesaplaması olarak bilinen bir teknikle de tespit edilebilir. Entropi, bir sistem veya programdaki bozukluğun bir ölçüsüdür ve entropiyi hesaplamak için iyi tanımlanmış bir standart matematiksel formül olmasa da, dijital veriler için pek çok iyi oluşturulmuş entropi ölçümü vardır. Sıkıştırılmış veya şifrelenmiş veriler rastgele verilere daha yakındır ve bu nedenle yüksek entropiye sahiptir; Şifrelenmemiş veya sıkıştırılmamış çalıştırılabilir dosyalarda daha düşük entropi vardır.

2.2.4.4. Polimorfizm

Virüsten koruma programlarının kötü amaçlı programları belirlemesinin en kolay yolu, benzersiz imzalar kullanmaktır. Virüsten koruma programı, bilinen her kötü amaçlı yazılım programı için benzersiz bir kimlik belirlemeyi amaçlayan, sık sık güncellenen virüs imza veri tabanını tutar. Bu tanımlama, kötü niyetli programın belirli bir dizisinde bulunan benzersiz bir diziye dayanmaktadır. Polimorfizm, imza temelli tanımlama programlarını, program kodunu orijinal işlevselliğini koruyacak şekilde rasgele kodlayarak veya şifreleyerek engelleyen bir tekniktir [46]. Polimorfizme en basit yaklaşım, programı rastgele bir anahtar kullanarak şifrelemek ve çalışma zamanında şifresini çözmek. Bir virüsten koruma programının programı imzası için ne zaman taradığına bağlı olarak, bu, her kopyasının tamamen farklı

olduğundan (rasgele bir şifreleme anahtarı kullanarak şifrelenmiş olduğundan) kötü amaçlı bir programın doğru bir şekilde tanımlanmasını önleyebilir.

2.2.4.5. Metamorfizm

Polimorfizm, kötü amaçlı yazılımın şifre çözme kodundaki çok yüzeysel değişikliklerle sınırlı olduğundan, virüsten koruma programlarının, kodu analiz ederek ve ondan bazı belirli düzeydeki bilgileri ayıklayarak polimorf kodunu tanımlaması sağlanabilmektedir.

Metamorfizmin resme girdiği yer burasıdır. Metamorfizm, polimorfizmden sonraki bir sonraki mantıksal adımdır. Programın gövdesini şifrelemek ve şifre çözme motorunda küçük değişiklikler yapmak yerine, her programda her kopyalandığında değişiklik yapmak mümkündür. Metamorfizmin faydası (bir kötü amaçlı yazılım yazıcısının perspektifinden), kötü amaçlı yazılımın her sürümünün, diğer sürümlerden radikal biçimde farklı görünebilmesidir. Bu, antivirüs yazarlarının kötü niyetli programı tanımlamak için her türlü imza eşleştirme tekniklerini kullanmasını çok zorlaştırır [47].

Komut ve Kayıtçı seçimi, metamorfik motorlar aslında kötü niyetli programı bütünüyle analiz edebilir ve tüm programın kodunu yeniden oluşturabilmektedir [48]. Kod yeniden bir araya getirilirken metamorfik motor, belirli komut seçimini (genellikle tek bir işlemi gerçekleştirmek için kullanılabilecek birden fazla komut vardır) ve kayıtların seçimi de dahil olmak üzere, kodla ilgili çeşitli parametreleri rastgele seçebilir.

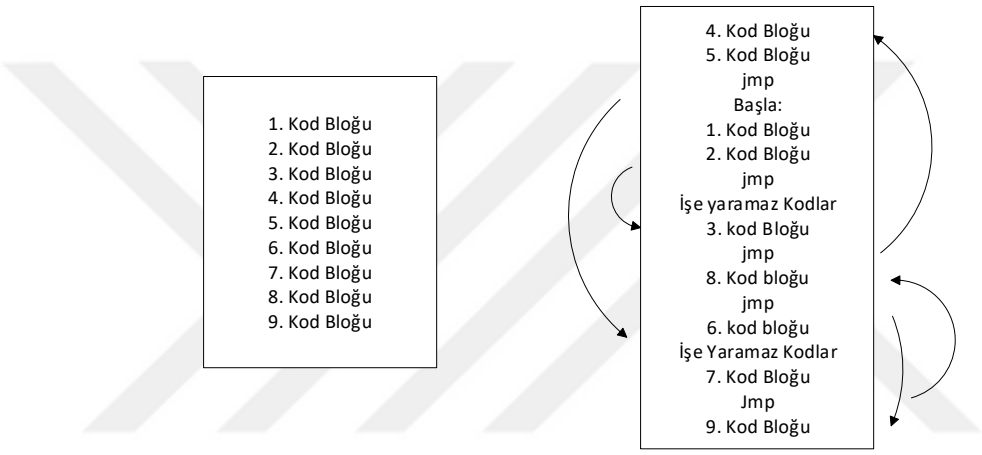
Komut sıralama, metamorfik motorlar söz konusu talimatlar birbirinden bağımsız olduğu sürece, bir fonksiyon içindeki talimatların sırasını bazen rastgele değiştirebilir.

Koşul tersinleme; kötü amaçlı yazılım kodunu ciddi şekilde değiştirmek için metamorfik bir motor, programda kullanılan koşullu ifadelerin bir kısmını tersine çevirebilmektedir [49]. Bir koşulu tersine çevirmek (örneğin), iki işlenenin eşit olup olmadığını kontrol eden bir ifade kullanmak yerine, eşit olmadıklarını kontrol etmeniz anlamına gelir (bu derleme işleminde düzenli olarak derleyiciler tarafından yapılır; bkz. Ek A). Bu, program kodunun önemli bir şekilde yeniden düzenlenmesine yol açar, çünkü metamorfik motoru, koşullu blokları tek bir fonksiyon içinde yeniden konumlandırmaya zorlar. Buradaki fikir, virüsten koruma

programı, metamorfik bir motorun öngörülmesi için programın bir tür üst düzey taramasını kullansa da programı tanımlamakta zorlanacağı yönündedir.

Faydasız kod ekleme; virüsten koruma tarayıcılarını daha fazla karıştırmak için program boyunca ilgisiz verileri işleyen çöp yönergelerini rastgele eklemek mümkündür [50]. Bu aynı zamanda metamorfik programı analiz etmeye çalışan insan ters çevricileri için belirli bir karışıklık yaratmaktadır.

İşlev sırası; işlevlerin modülde saklanma sırası, programın çalışma zamanında program için çok az önemli olduğunu ve rastgele olarak programlanmasının programın tanımlanmasını biraz zorlaştırabilmektedir [50].



Şekil 2.6: Metamorfizm uygulamaları

2.3. Analiz Teknikleri

Genel olarak zararlı yazılım analizinin ilk amacı ve temel hedefi zararlı yazılımın ne olduğunu tam anlamıyla anlamak ve bulaşmış makinede ne etkilere sebep olduğunu ne gibi aktivitelerde bulunduğunu tespit etmektir. Analiz edilen yazılımın neler yapabileceğini bilmek gerekmektedir. Zararlı yazılıma karşı hazırlıklı olmak ve tespit edebilmek için incelenmesi gerekmektedir.

Çoğu zaman, kötü amaçlı yazılım analizi yaparken, yalnızca insan tarafından okunamayan kötü amaçlı yazılımın çalıştırılabildiğini görürsünüz. Anlamak için, her biri az miktarda bilgi açığa çıkaran çeşitli araçlar ve püf noktaları kullanacaksınız. Resmin tamamını görmek için çeşitli araçlar kullanmanız gerekir. Kötü amaçlı yazılım analizine iki temel yaklaşım vardır: statik ve dinamik. Statik analiz, kötü

amaçlı yazılımı çalıştırmadan incelemeyi içerir. Dinamik analiz, kötü amaçlı yazılımı çalıştırmayı içerir.

Zararlı yazılım analizi, zararlı yazılımın farklı bileşenlerini parçalayarak incelenmesi ve onun davranışı ve kod yapısı üstüne yapılan çalışmaların tümüdür. Zararlı yazılım tespiti ve sınıflandırma üstüne çalışan neredeyse tüm araştırmacılar zararlı yazılım analizi yapmışlardır [5, 6, 8, 9]. Bununla birlikte, Ortaya çıkan yeni türler ile zararlı yazılım türlerini sınıflandırmak veya tespit etmek daha zor hale gelmektedir [13]. Veyahut analizden kaçınma yöntemleri, yapay zekâ teknolojileri ve benzeri yöntemler ile zararlı yazılımlar daha donanımlı hale gelmektedir.

Kötü amaçlı yazılım analizi, kötü amaçlı yazılımın nasıl çalıştığını, nasıl tanımlanacağını ve nasıl yenileceğini veya nasıl ortadan kaldırılacağını anlamak için ayırma sanatıdır. Her gün milyonlarca kötü niyetli programda ve her gün daha çok karşılaşılan zararlı yazılım analizi, bilgisayar güvenliği olaylarına yanıt veren herkes için kritik öneme sahiptir. Ayrıca, zararlı yazılım analizi uzmanlarının azlığı ile, yetenekli zararlı yazılım analisti ve otomatik zararlı yazılım analiz ve tespit araçları ciddi talep görmektedir.

Saldırganlar ile Savunucular arasındaki bu silahlanma yarışında, ne yazık ki, kötü niyetli davranış araştırmacıların zararlı davranışların sadece tespitin özniteliği olarak kullandığı, davranışları tespit etmedikleri sürece donanımlı ve önceden bilinmeyen bir zararlının yayılmasını engellemek mümkün değildir. Bu kısımda, buna binaen zararlı yazılım tespit çalışmalarını incelemeden önce analiz yöntemleri açıklanmış, çıkarılan öznitelik verilerinin işlenme teknikleri başlıca yöntemler üstünden ayrıntılı olarak anlatılmıştır. Daha sonrasında literatürdeki çalışmalar incelenmiştir.

Benzer örüntü bulmak hususunda düşünülecek ilk konu zararlı yazılım analizidir [8]. Zararlı yazılım analizi ile, işlem kodları (OpCodes), uygulama programlama arabirim (API) fonksiyonlarının çağrımı, davranış bilgileri, başlık bilgileri, çalışma süresi, üzerinde çalıştığı bilgisayardaki etkiler gibi öznitelikler çıkarılır. Yazılım analizi genel olarak yazılım davranışı ve kod yapısı hakkında bilgi vermektedir. Bu bilgiyi otomatize bir şekilde çıkarmak ve öğrenmek için analiz ile öznitelikler çıkarılır. Öznitelikler zararlı yazılımın ne yaptığını ve nasıl kullanıldığını açıklar. Kötü amaçlı yazılım analizi, statik ve dinamik olmak üzere iki yaklaşımla gerçekleştirilir. Statik teknikler, ikili kodu çalıştırmadan analiz eder; dinamik teknikler ise zararlı yazılım kontrollü bir ortamda çalışırken bilgisayarda yarattığı

etkiyi gözlemleyerek davranışlarını incelemeye dayanır [9]. Bu yaklaşımlardan herhangi biri tek başına uygulanması zararlı yazılımlar hakkında bilgi kaybına sebebiyet verebilir [10]. Çünkü, farklı öznitelik tipleri yazılımın farklı bir yönünü göstermektedir ve belki o bakış açısı zararlı yazılımı açık eder. Yani, her iki analiz türünde de bazı avantajlar ve dezavantajlar vardır [11].

2.3.1. Durağan Analiz

Analiz edilen yazılım çalıştırılmadan yapılan analizdir. En önemli aracı tersine mühendislik konusunu da kapsayan çözücü yardımı ile yazılımın assembly kodlarına erişilmesidir. Temel anlamda durağan analiz çalıştırılabilir dosyanın ne olduğunu ve ne yaptığını anlamak için çalıştırmadan tetkik etmektir [25]. Temel durağan analiz incelenen yazılımın işlevleri hakkında bilgi sağlar. dosya format yapısından faydalanılarak, başlık bilgisi, fonksiyon bilgisi, boyut bilgisi, özüt değerleri vb. bilgiler çıkarılır. Daha karmaşık analizde ise bulanıklaştırılmış ya da makine koduna çevrilmiş dosyanın assembly kodları çıkarılır ve bunlar üstünden analiz yapılır. Bu assembly kod dosyası ile yazılım kod içeriğini [51], kontrol akış diyagramını [52], içinde var olan dizgi yazılarını [53], çağrılan fonksiyonları [54], içe aktarılan sistem dosyalarını [55] çıkarılmakta ve analiz edilmektedir. Bu tarz veriler yazılım hakkında zararlı faaliyette bulunup bulunmayacağı hakkında bilgi verirken, bilgisayara zarar verecek yapıda olup olmayacağı anlaşılmasına yardımcı olur.

Statik analiz tersine mühendislik olarak da anılır. Bu işlem çözücü (disassembler) yardımı ile yapılır. Ayırıcıya yüklenen program assembly dosyasına dönüştürülür. Bu dosya makinenin direk anlayacağı assembly komutlarını içermektedir. Bu komutlar sanki çalışır gibi izlendiğinde programın kontrol akış diagramı ve grafı ortaya çıkmaktadır. Grafın düğümleri kod blokları, bağlantılar ise kod blokların birbirleri arasındaki bağlar olmaktadır. Bu bağlar, *jump* veya *call* vb. komutlar ile sağlanmaktadır. Bu komutlar indis yazmaçı değerini değiştirir ve programın devam edeceği adresi belirler. İndisin her değişimi, gittiği adresteki kod bloğunun başlangıcı, ayrıldığı adresteki kod bloğunun ise bitişini belirler. Dolayısıyla program akışı dallanmış olur. Bu diagram zararlı yazılım tespitinde sıklıkla kullanılmaktadır [56].

Bir çalıştırılabilir dosya hakkında toplayabileceğimiz en yararlı bilgi parçalarından biri de içe aktardığı işlevlerin listesidir. İçe aktarılan fonksiyonlar, bir

program tarafından kullanılan ve çoğu program için ortak işlevsellik içeren kod kitaplıkları gibi farklı bir programda saklanan işlevlerdir [54]. Kod kütüphaneleri ana çalıştırılabilir dosyaya bağlanarak çalıştırılmaktadır. Tüm bağlantı yöntemlerinden, dinamik bağlantı kötü amaçlı yazılım analistleri için en yaygın ve en ilginç olanıdır. Kütüphaneler dinamik olarak bağlandığında, ana bilgisayar işletim sistemi program yüklendiğinde gerekli kütüphaneleri arar. Program bağlantılı kütüphane işlevini çağırdığında, bu işlev kitaplığın içinde yürütülür. PE dosya başlığı, yüklenecek her kitaplık ve program tarafından kullanılacak her işlev hakkında bilgi depolar. Kullanılan kütüphaneler ve çağrılan işlevler genellikle bir programın en önemli kısımlarıdır ve bunları tanımlamak özellikle önemlidir, çünkü programın ne yaptığını tahmin etmemize olanak tanır. Örneğin, bir program *URLDownloadToFile* işlevini içeri aktarırsa, daha sonra yerel bir dosyada depoladığı içeriği indirmek için İnternet'e bağlandığını tahmin edebilirsiniz.

Tablo 2.6: En Sık Kullanılan Kitaplıklar

Kitaplık	Açıklama
Kernel32.dll	Çekirdek fonksiyonları barındıran en sık kullanılan kitaplık. Bellek, dosya ve donanım erişimi ve değiştirilmesi için kullanılan API'leri barındırır.
Advapi32.dll	Kayıt defteri ve hizmet yönetimi konusunda çalışan API'leri barındırır.
User32.dll	Kullanıcı arayüzü ile ilgili fonksiyonlar
Dgi32.dll	Grafik değiştirilmesi ve erişilmesiyle alakalı, ekran ve görüntü fonksiyonlarını barındırır
Ntdll.dll	Çekirdek ile arayüz kitaplığıdır. Sistem fonksiyonlarını barındırır.
Wsock32.dll	Ağ yönetimi ile ilgili kitaplık
Wininet.dll	Üst seviye ağ fonksiyonları, internet ile alakalı FTP,HTTP, NTP gibi protokollerle ilgili fonksiyonları barındırır.

2.3.1.1. Çözücü İşlemi

Çözücünün tam manası, genellikle talimatları çalıştırmadan bir ikili sistemden çıkarmayı içeren statik çözme olarak verilmektedir. Buna karşılık, daha yaygın

olarak yürütme izlemesi olarak bilinen dinamik sökme, çalıştırılan her komutu ikili çalıştırma olarak kaydeder. Her statik çözme cihazının amacı bir ikilideki tüm kodu bir insanın okuyabileceği veya bir makinenin işleyebileceği bir formata çevirmektir. Bu amaca ulaşmak için, statik parçalayıcıların aşağıdaki adımları gerçekleştirmeleri gerekir:

i) İkili dosyanın yüklenmesi
ii) Tüm ikili kodu ile yazılmış tüm makine komutlarının bulunması
iii)Komutların assembly dile dönüştürülmesi.

Çözücü işleminin zorlayıcı noktası 2. Adımdır. Çünkü makine kodlarını bulmak her zaman kolay değildir. Her ikili kod komut belirtmeyebilir, veri de olabilir. Bu yüzden çeşitli önerilmiş çözücüler vardır. Genel olarak komutların bulunması için 2 yaklaşım vardır; lineer çözücü, özyinelemeli çözücü.

2.3.1.2. Çözümleme Zorlukları

- Derleme işlemi kayıplıdır. Makine dili düzeyinde değişken veya işlev adı yoktur ve değişken türü bilgileri, açık tür bildirimleri yerine yalnızca verilerin nasıl kullanıldığıyla belirlenebilir. 32 bit veri aktarıldığını gözlemlediğinizde, bu 32 bitin bir tam sayı, 32 bit kayan nokta değeri veya 32 bit işaretçi temsil edip etmediğini belirlemek için bazı araştırmalar yapmanız gerekir.
- Derleme çoktan çoğa bir işlemdir. Bu, bir kaynak programın montaj diline birçok farklı şekilde tercüme edilebileceği ve makine dilinin de birçok farklı şekilde kaynağa geri çevrilebileceği anlamına gelir. Sonuç olarak, bir dosyanın derlenmesinin ve derhal çözülmesinin, girilen dosyadan çok farklı bir kaynak dosyası vermesi oldukça yaygındır.
- Çözücü çok dile ve kütüphaneye bağımlıdır. Delphi derleyicisi tarafından üretilen bir ikili kodun C kodu oluşturmak üzere tasarlanmış bir genişletici ile işlenmesi çok garip sonuçlar verebilir. Benzer şekilde, derlenmiş bir Windows ikili dosyasını Windows programlama API'sı hakkında bilgi sahibi olmayan bir Çözücü aracılığıyla beslemek yararlı bir şey getirmeyebilir.

- Bir ikiliyi doğru şekilde çözmek için neredeyse mükemmel bir çözme kabiliyeti gereklidir. Çözme aşamasındaki herhangi bir hata veya eksiklik neredeyse kesinlikle ayrıştırılan koda yayılacaktır.

İki tür çözme algoritması vardır: doğrusal ve akış yönelimli. Doğrusal çözme işlemlerinin uygulanması daha kolaydır, ancak aynı zamanda hataya daha açıktır.

2.3.1.3. Lineer Çözme

Kavramsal olarak nispeten daha basit yaklaşım doğrusal çözme olarak kabul edilir. Bir ikilideki tüm kod bölümleri boyunca sırayla tüm baytların kodunu çözer ve bunları bir talimatlar listesine ayrıştırır. Birçok basit çözücü bu yaklaşımı kullanmaktadır [12]. Doğrusal çözme kullanmanın riski, tüm baytların talimat olamayacağıdır. Örneğin, Visual Studio gibi bazı derleyiciler, tam olarak bu verilerin nerede olduğuna dair hiçbir ipucu bırakmadan, kodlarla atlama tabloları gibi verileri kesmektedir. Çözme cihazları yanlışlıkla bu satır içi verileri kod olarak ayrıştırırlarsa, geçersiz kodlarla karşılaşabilirler. Daha da kötüsü, veri baytları tesadüfen geçerli OpCode karşılık gelebilir ve bu da çözücülerin sahte talimatlar çıkarmasına neden olabilir. Bu özellikle, bayt değerlerinin çoğunun geçerli bir OpCode'u temsil ettiği x86 gibi yoğun ISA'larda olasıdır. Bu çalışmada kullanılan distorm3 bir lineer çözücüdür.

2.3.1.4. Özyinemeli Çözücü

Doğrusal sökme işleminden farklı olarak, özyinelemeli sökme kontrol akışına duyarlıdır. Bilinen giriş noktalarından ikiliye başlar (örneğin ana giriş noktası ve dışa aktarılan işlev sembolleri gibi) ve buradan kodu bulmak için tekrar tekrar kontrol akışını (atlar ve çağrılar gibi) izler [12]. Bu, özyinelemeli çözmenin bir avuç köşe vakası dışında hepsinde veri baytları etrafında çalışmasına izin verir. Bu yaklaşımın dezavantajı tüm kontrol akışının takip etmesinin o kadar kolay olmamasıdır. Örneğin, dolaylı atlamalar veya çağrıların olası hedeflerini statik olarak bulmak imkânsız olmasa da çoğu zaman zordur [19]. Sonuç olarak, sökme cihazı, özel (derleyiciye özel ve hataya eğilimli) sezgisel yöntemler kullanmıyorsa, dolaylı sıçramalar veya çağrılarla hedeflenen kod bloklarını (hatta Şekil 6-1'deki f1 ve f2

gibi tüm fonksiyonları) kaçırabilir ya da yanılabilir. Özyinelemeli çözme, zararlı yazılım analizi gibi birçok ters mühendislik uygulamasında fiili standarttır. IDA Pro en gelişmiş ve yaygın olarak kullanılan özyinelemeli çözücülerden biridir [22].

2.3.1.5. Kontrol Akış Grafiği

Assembly kodu fonksiyonlara bölmek bir şeydir, ancak bazı fonksiyonlar oldukça büyüktür, yani bir fonksiyonu bile analiz etmek karmaşık bir iş olabilir. Her işlevin içindekileri düzenlemek için, çözücüler ve ikili analiz çerçeveleri kontrol akış grafiği (CFG) adı verilen başka bir kod yapısı kullanır [57]. CFG'ler, otomatik analizin yanı sıra manuel analiz için de faydalıdır. Ayrıca, kod yapısının kullanışlı bir grafik gösterimini sunarlar; bu, bir fonksiyonun yapısı hakkında bir bakışta fikir sahibi olmayı kolaylaştırır. Literatürde graf benzerliklerini graf izomorfizm ile bulup tespit yapan çalışmalar vardır [58].

2.3.1.6. Fonksiyon Çağrı Grafiği

Çağrı grafikleri, CFG'lere benzer, ancak temel bloklardan ziyade çağrı siteleri ve fonksiyonlar arasındaki ilişkiyi gösterirler. Başka bir deyişle, CFG'ler kontrolün bir fonksiyon içinde nasıl aktığını gösterirken çağrı grafikleri hangi fonksiyonların birbirini çağırdıklarını gösterir [59]. CFG'lerde olduğu gibi, çağrı grafikleri de genellikle dolaylı çağrı kenarlarını atlar, çünkü hangi fonksiyonların belirli bir dolaylı çağrı sitesi tarafından çağrılabileceğini doğru bir şekilde bulmak mümkün değildir.

2.3.2. Dinamik Analiz

yazılımın güvenli bir ortamda çalıştırılıp, bilgisayar üstünde yarattığı etkiler ve yazılım davranışlarını izleyerek tetkik etmektir. Temel anlamda yapılan dinamik analizde API çağrımları, bilgisayara indirilen dosyalar veya ağ bilgileri gibi elde edilir. Daha gelişmiş dinamik analizde hata ayıklayıcı kullanılır. Gelişmiş dinamik analiz teknikleri, çalıştırılabilir bir bilgiden ayrıntılı bilgi elde etmenin başka bir yolunu sunar. Bu teknikler, diğer tekniklerle toplanması zor olan bilgileri edinmeye çalıştığınızda çok faydalıdır.

2.3.2.1. Dinamik renk tonu analizi (DTA)

Veri akışı izleme (DFT), renk tonu izleme veya basitçe renk tonu analizi olarak da adlandırılan Dinamik renk tonu analizi (DTA), seçilen bir program durumunun program durumunun diğer bölümleri üzerindeki etkisini belirlemenizi sağlayan bir program analiz tekniğidir [60]. Örneğin, bir programın ağdan aldığı herhangi bir veriyi etiketleyebilir, bu verileri izleyebilir ve program sayacını etkiliyorsa bir uyarı verebilir; bunun gibi bir etki kontrol akışı kaçırma saldırısını gösterebilir.

2.3.2.2. Hata Ayıklayıcılar

Çözücüler, ilk talimatın uygulanmasından hemen önce bir programın nasıl göründüğünün bir görüntüsünü sunar. Hata ayıklayıcıları, çalışırken bir programın dinamik bir görünümünü sağlar. Örneğin, hata ayıklayıcılar, bir programın yürütülmesi sırasında değiştikçe bellek adreslerinin değerlerini gösterebilir. Bir programın çalışmasını ölçme ve kontrol etme yeteneği, kötü amaçlı yazılım analizi sırasında kritik bir görüş sağlar. Hata ayıklayıcılar, her bir hafıza konumunun değerini görmenizi, her fonksiyonun kaydını ve argümanını görmenizi sağlar. Hata ayıklayıcılar ayrıca programın çalışması hakkında istediğiniz zaman herhangi bir şeyi değiştirmenize de izin verir. Hata ayıklayıcılar manuel olarak yapılan zararlı yazılım analizlerinde daha sıklıkla kullanılır. Otomatik analizlerde hata ayıklayıcılar sadece dinamik OpCode dizisini çıkarmak için faydalıdır [61].

2.3.2.3. Kum havuzu

Kötü amaçlı yazılımlar daha karmaşık hale geldikçe, sistemimizden ödün vermeden kötü amaçlı yazılımları kolayca analiz etmemizi sağlayacak daha fazla teknolojiye ihtiyacımız olmuştur. Kullanılabilecek böyle bir teknoloji, kumlamadır. Belirli terminolojide (bilgisayar güvenliği), sanal alan, güvenilmez programları ana ortamdan çalıştırmak için kullanılabilecek sınırlı uygulama ortamları sağlayarak bir programı (bu durumda kötü amaçlı yazılımları) yalıtmak için kullanılan bir tekniktir. Kum havuzu teknolojisi hakkında net bir açıklama yapmak için, çocuklar için bir

kum havuzu veya kum havuzu hayal edelim. Kum havuzu, çocukların oynaması için kum dolu bir kaptır. "Çukur" veya "havuzun" kendisi sadece çimleri depolamak için bir konteynırdır, böylece çimler veya diğer çevre yüzeylere dışarıya yayılmaz. Çocuklar, hala sanal alanda oldukları sürece, kumlardaki her şeyi yapabilirler. Aynı mantıkla bir sanal alan sağlayarak, kötü amaçlı uygulamaları yürütebilir ve kötü amaçlı yazılım etkinliklerini görebiliriz.

Ayrıca, kötü amaçlı yazılımları işlem sırasında meydana gelecek değişikliklerden endişe etmeden güvenli ve güvenli bir şekilde analiz edebiliriz. Çeşitli zararlı yazılım sanal analiz alanları vardır. Cuckoo, sanal alandaki bir kötü amaçlı yazılımı analiz etmek için doğru araçtır, çünkü Cuckoo tam bir özelliğe sahiptir, tamamen açık bir kaynaktır ve topluluğundan iyi bir destek almaktadır. Çok sayıda hepsi bir arada yazılım ürünü, temel dinamik analiz yapmak için kullanılabilir ve en popüler olanları sanal alan teknolojisini kullanır. Sanal alan, "gerçek" sistemlere zarar verme korkusu olmadan güvenilmeyen programları güvenli bir ortamda çalıştırmak için bir güvenlik mekanizmasıdır. Sanal alanlar, test edilen yazılımın veya kötü amaçlı yazılımın normal şekilde çalışmasını sağlamak için bir şekilde ağ hizmetlerini sıklıkla taklit eden sanallaştırılmış ortamlar içerir. Kötü amaçlı yazılım raporu, kötü amaçlı yazılım hakkında gerçekleştirdiği ağ etkinliği, oluşturduğu dosyalar, VirusTotal ile tarama sonuçları vb. Gibi çeşitli ayrıntıları içerir.

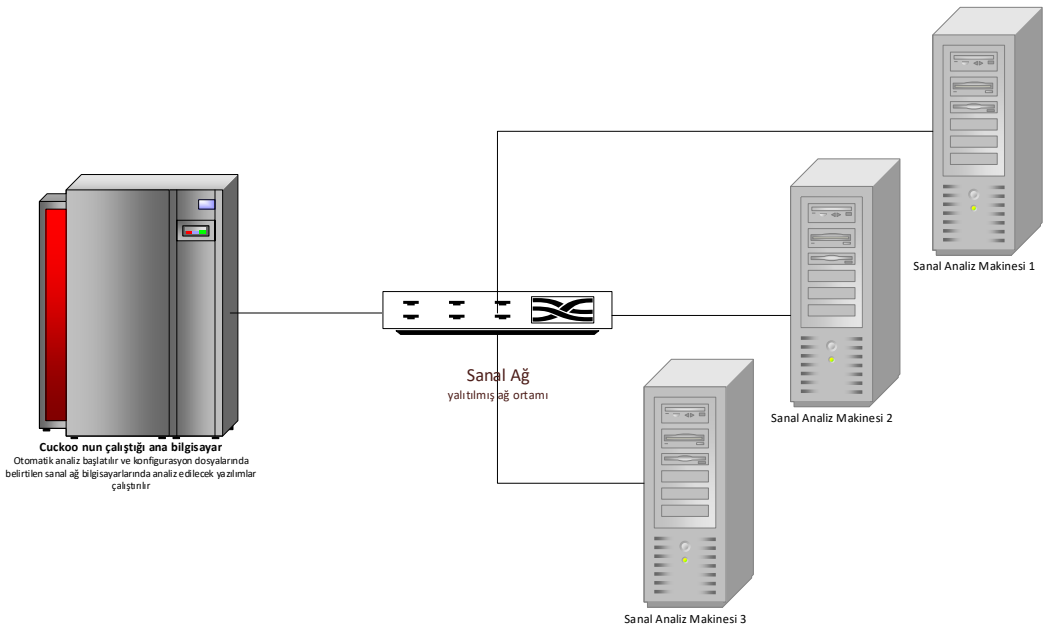
Tüm dinamik analizlerin ana dezavantajı, yalnızca dinamik sökme işlemi değil, kod kapsamı sorunudur: analiz yalnızca analiz çalışması sırasında yürütülen talimatları görür. Bu nedenle, herhangi bir önemli bilgi başka talimatlarda gizlenmişse, analiz bu konuda hiçbir zaman bilgi sahibi olmayacaktır. Örneğin, bir mantık bombası içeren bir programı dinamik olarak analiz ediyorsanız (örneğin, gelecekte belirli bir zamanda kötü niyetli davranışları tetikleme), çok geç olmadan asla öğrenemezsiniz. Buna karşılık, statik analiz kullanarak yapılan sıkı bir denetim bunu ortaya koymuş olabilir. Başka bir örnek olarak, yazılımları hatalara karşı dinamik olarak test ederken, testlerinizde ele almadığınız, nadiren çalıştırılan bazı kod yollarında bir hata olmadığından asla emin olamazsınız.

Cuckoo dinamik analiz yaklaşımının pratik uygulamalarına sahip bir zararlı yazılım sanal alanı aracıdır [62]. İkili dosyayı statik olarak analiz etmek yerine, gerçek zamanlı olarak yürütülür ve izlenir. Basit bir açıklama olarak, Cuckoo, sanal alandaki kötü amaçlı yazılımlar üzerinde analiz yapmanızı sağlayan açık kaynaklı bir

otomatik kötü amaçlı yazılım analiz sistemidir. Cuckoo Sandbox, Honeynet Projesi kapsamında 2010 yılında bir Google Yaz Kodu projesi olarak başladı. 2010 yazındaki ilk çalışmadan sonra, ilk beta sürümü 5 Şubat 2011'de Cuckoo'nun ilk kez halka duyurulduğu ve dağıtıldığı yayınlandı.

Mart 2012'de Cuckoo Kum havuzu, Rapid7 tarafından düzenlenen Magnificent7 programının ilk raundunu kazandı. Cuckoo, geliştiricilerin geleneksel ve mobil tabanlı kötü amaçlı yazılım analizlerine yenilikçi yaklaşımları nedeniyle ilk Magnificent7 sponsorluk turunda Rapid7 tarafından seçildi. Cuckoo, dosyaları otomatik olarak çalıştırmak ve analiz etmek ve izole edilmiş bir Windows işletim sistemi içinde çalışırken kötü amaçlı yazılımın ne yaptığını gösteren kapsamlı analiz sonuçları toplamak için kullanılır.

Cuckoo Kum havuzu, kötü amaçlı yazılım örnek işlemlerini ve analizlerini yapan merkezi bir yönetim yazılımından oluşur. Her analiz taze ve yalıtılmış bir sanal makinede başlatılır. Cuckoo'nun altyapısı bir ana makine (yönetim yazılımı) ve bir dizi konuk makine (analiz için sanal makineler) tarafından oluşturulmaktadır. Ev sahibi, tüm analiz sürecini yöneten sanal alanın çekirdek bileşenini çalıştırırken, konuklar kötü amaçlı yazılımın gerçekten güvenli bir şekilde yürütüldüğü ve analiz edildiği yalıtılmış ortamlardır. Aşağıdaki şekil Cuckoo mimarisini göstermektedir:



Şekil 2.7: Cuckoo Çalışma ortamının resmedilişi

2.4. Öğrenme Algoritmaları

2.4.1. Markov Zinciri

Markov zinciri, olasılık kurallarına göre bir durumdan diğerine geçişleri modelleyen matematiksel bir sistemdir. Markov zincirinin belirleyici özelliği, sürecin andaki durumuna nasıl gelirse gelsin, gelecekteki olası durumların sabit olmasıdır. Bu Markov zincirinin stokastik model olduğu anlamına gelmektedir. Diğer bir deyişle Markov zinciri "hafızasız" olmalıdır. Yani, gelecekteki eylemlerin olasılığı mevcut duruma yol açan adımlara bağlı değildir. Buna "Markov özelliği" denir. Koşullu olasılık ve rasgele değişkenlerin dilinde bir Markov zinciri, koşullu bağımsızlık kuralını karşılayan rasgele değişkenler bir dizisidir ve şöyle tanımlanır:

Tanım: Markov Özelliği

Herhangi bir n tam sayısı ve rastgele değişkenlerin muhtemel durumları $i_0, i_1, i_2, \ldots, i_n$, *için*

$$P(X_n=i_n/X_{n-1}=i_{n-1})=P(X_n=i_n/X_0=i_0,X_1=i_1, \ldots, X_{n-1}=i_{n-1}).$$

Markov zinciri, bir sistem ya da olayın "hafızasız" olasılık dağılımıdır. Bu dağılım modellenen sistemin ya da olayın durumları üstünden gerçekleşir. Bir durumdan ötekisine geçişlerin olasılıklarını çıkaran ve bu olasılıklar ile olayın *t* anında hangi durumda olduğunu tahmin etmek için kullanılan bir modeldir. Bu hesaplamalar geçiş matrisleri kullanılarak yapılmaktadır. Geçiş matrisleri, sistem ya da olayın durumlarının değişim olasılıklarını barındıran ve her satırdaki olasılık değerlerinin toplamı 1 olan stokastik bir matristir.

Tanım: Geçiş Matrisi

Durum kümesi S bazında satır ve sütunları sıralı olarak verilen stokastik matris P için,

Her i,j € S için,

$$(P_t)_{i,j}=P(X_{t+1}=j/X_t=i).$$

Zararlı yazılım tespitinde Markov zincirleri dizilerin analiz edilmesinde kullanılmaktadır [63]. Zararlı yazılımların ağ içindeki yayılma davranışlarını incelemek ve ağ içinde farklı bilgisayarlara bulaşan zararlı yazılımın tespiti için Markov zinciri tekniği kullanıldığı gözlemlenmiştir [64]. Bu kullanımda ağın zamandaki değişimi ele alınmış ve değişen zaman içerisinde ağın durumu, zararlı yazılım yayılması olduğu ağ durumundakine benzer ya da yaklaşık olma durumu tespit edilmeye çalışılmıştır. Bu benzerlik eğitilen Markov zinciri üstünden, ağ durumları olasılıklarının geçiş matrisine benzerliği ele alınmıştır. Bir diğer kullanım ise bu çalışmada olduğu gibi geçiş matrisleri üstünden öğrenim yapılmasıyla gerçekleştirilir.

2.4.2. Gizlenmiş Markov Model

HMM, durumları giriş verilerinin özelliklerini temsil eden bir durum makinesidir. Durumlar arasındaki geçişler, modelde kullanılacak özniteliklerin olasılığı olan istatistiksel özellikleri gösterir. Her durumun olası çıkış belirteçleri üzerinde bir olasılık dağılımı vardır, böylece bir dizi gözlemi bir etiket dizisine eşler. Gözlem sembolleri, gözlemlenen her türlü olayın bir ifadesidir. HMM, gözlemlenen olay dizileri kullanılarak her sınıf için eğitim gerçekleştirilir. HMM, her sınıfı dizi varyasyonlarına göre geçiş ve gözlem olasılıkları olarak tanımlamaktadır. Test aşamasında Verilen herhangi bir olay dizisi eğitimli HMM kullanılarak puanlanır. Puan eşiğin üstünde ise, analiz edilen dosya eğitimi gerçekleştirilen modelin sınıfına ait olarak etiketlenir. Bu yapı her zararlı yazılım ailesi için bir model oluşturarak, örüntü çıkarmayı temel alır. Böylece benzer örüntüye sahip yazılımlar o aileye mensup olarak sınıflandırılır.

2.4.3. Destek Vektör Makinesi

Diğer makine öğrenme tekniklerinin çoğunda olduğu gibi, SVM'leri basit bir şekilde çok yüksek bir perspektiften tarif etmek mümkündür. Ancak, önceki bölümlerde, bir makine öğrenme tekniğini gerçekten anlamak için ayrıntılara girmemiz gerektiğini ve bu anlamda SVM'nin istisna olmadığını gördük. Önceki

bölümlerde olduğu gibi, buradaki hedefimiz, okuyucunun SVM'de gerçekte neler olup bittiğini anlayabilmesi için yeterli bir ayrıntı sunmaktır. SVM'lerin arkasındaki büyük fikirler açık ve sezgisel. Bu fikirlere birazdan ulaşacağız, ancak önce kısaca denetimli ve denetimsiz öğrenmeyi tartışıyoruz. Denetimli bir öğrenme algoritması, etiketli bir eğitim verisi gerektirir. Yani, eğitim verileri önceden kategorize edilmelidir. Örneğin, kötü niyetli düzenlemede, eğitim verileri, belirli bir tipte (veya ailede) kötü amaçlı yazılım olduğu bilinen bir dizi örneğin yanı sıra, diğer iyi huylu örnekler grubundan oluşabilir. Bu tür bir etiketlenmiş veri SVM eğitim sürecinde gereklidir.

Bir SVM'yi eğitirken amaç, bir hiper düzlemin çalıştığımız alandan daha az bir boyutun alt alanı olarak tanımlandığı, ayırıcı bir hiper düzlem bulmaktır. Örneğin, eğer verilerimiz iki boyutlu uzayda yaşıyorsa, hiper düzlem sadece bir çizgidir. Ve "ayırmak" tam olarak ne demek istediğini, yani hiper uçağın iki sınıfı ayırdığını ifade eder. Ayırıcı bir hiper düzlem varsa, verilerin doğrusal olarak ayrılabilir olduğunu söyleriz. Eğer eğitim verilerimiz doğrusal olarak ayrılabilir hale gelirse, daha sonraki sınıflandırmanın temeli olarak herhangi bir ayırıcı hiper düzlem kullanılabilir.

2.4.4. K-En Yakın Komşu

KNN algoritması benzer şeylerin yakınlarda var olduğunu varsayar. Başka bir deyişle, benzer şeyler birbirine yakındır. KNN algoritmasının yararlı olması, bu varsayımın yeterince doğru olduğuna bağlıdır. Bu algoritma kendisine *k* kadar sayıda yakın olan örneğe benzer olduğunu söylemektedir. Bu bağlamda algoritmanın doğru çalışması, benzerlik fonksiyonunun gerçek sonuç vermesiyle doğrudan ilişkilidir. Zararlı yazılımların birbirine benzerliği önemli bir konudur. Benzerliğin hesaplanması, seçilecek özniteliklerle doğrudan ilişkilidir.

2.4.5. Karar Ağaçları

Karar ağacı, sınıflandırma ve tahmin için en güçlü ve popüler araçtır. Bir Karar ağacı, her iç düğümün bir öznitelik üzerinde bir testi temsil ettiği, her dalın testin bir sonucunu temsil ettiği ve her bir yaprak düğümünün (terminal düğümü) bir sınıf etiketi içerdiği, ağaç yapısı gibi bir akış şemasıdır.

Bir ağaç, öznitelik değeri testine göre ayarlanan kaynak alt kümelere bölünerek öğrenmektedir. Bu işlem, türetilmiş her alt kümede özyinelemeli bölümleme adı verilen özyinelemeli bir şekilde tekrarlanır. Bir düğümdeki altkümenin tümü hedef değişkenle aynı değere sahip olduğunda veya bölme artık tahminlere değer eklemediğinde özyineleme tamamlanır. Karar ağacı sınıflandırıcısının oluşturulması herhangi bir alan bilgisi veya parametre ayarı gerektirmez ve bu nedenle keşif bilgisi keşfi için uygundur. Karar ağaçları yüksek boyutlu verileri işleyebilir. Genel olarak, ağaç sınıflandırıcısının doğruluğu iyidir. Karar ağacı sınıflandırma hakkında bilgi edinmek için tipik bir tümevarım yaklaşımdır.

2.4.6. Rast gelelik Ormanı

Bünyesinde çok sayıda karar ağacını barındıran toplu öğrenme algoritmasıdır. Tahmini o karar ağaçların çoğunlukta verdiği karar olarak belirlemektedir. Rastgele ormanın ardındaki temel kavram, basit ama güçlü bir kavramdır - kalabalıkların bilgeliği. Veri biliminde, rastgele orman modelinin bu kadar iyi çalışmasının nedeni budur: Komite olarak faaliyet gösteren göreceli olarak ilişkisiz çok sayıda model (ağaç), tek tek kurucu modellerin herhangi birinden daha iyi performans gösterecektir.

2.4.7. Derin Öğrenme

Derin öğrenme özelleştirilmiş bir makine öğrenmesi alt alanıdır [65]. Bu yüzden derin öğrenme konusunu incelemeden önce makine öğrenmesini incelemek gerekir. Makine öğrenmesi şu soru ile ortaya çıkmıştır: Bir bilgisayar kendi başına belirli bir görevi nasıl gerçekleştireceğini öğrenebilir mi [66]? Diğer bir deyişle, klasik programların yaptığı belirli kurallar çerçevesinde algoritma ile yazılıp verilen veri ile cevap üretirken, makine öğrenmesi algoritması veri ve o verilerin cevapları ile kuralları üretir ve gelecek yeni verilerle ona göre cevap verir.

Derin öğrenme de makine öğrenmesinin alt alanı olarak aynı kaygıyı gütmektedir. Buradaki “derin” ifadesi, öğrenme işlemini her defasında daha anlamlı olacak şekilde katman katman yapılmasına denk gelir [66]. Önceki çizilen gösterimden açıklanacak olunursa; derin öğrenmenin ilk katmanında makine

öğrenmesinde olduğu gibi veri ve cevaplardan kurallar üretilir. Bir sonraki katmanında ise önceki katmanda üretilen kurallar ve ilk girdideki cevaplar kullanılarak yeni kurallar üretilir. “Derin” kavramı da bu katmanlı yapıda katman sayısının artması anlamına gelir ve anlaşılacağı gibi derin öğrenme işlemi öğrenilmiş kurallardan yeni kurallar üretme üstüne kurulmuştur. Son katmanda ise cevaplar uzayından bir elemanı karar olarak vermesi beklenir.

2.5. İlgili Çalışmalar

Günümüzün virüsten koruma programlarının çoğu, tespit teknikleri olarak imza tabanlı yöntemler kullanır. Kötü amaçlı yazılımın imzası, kötü amaçlı yazılımı kesinlikle algılayabilen benzersiz bir bayt dizisidir. İmzaya dayalı yöntemlerin uygulamada kabul edilebilir sonuçları vardır, ancak yeni türler veya bilinmeyen kötü amaçlı yazılımlarla başa çıkamazlar.

2.5.1. Kod Yapısı Analizi Tabanlı Yöntemler

Dinamik analizin hala bazı sınırlamaları vardır. Bazı teknikler, çalışma zamanı ortamının sanal ortam olup olmadığını test edebilir ve kötü amaçlı yazılımlar bu teknikleri kullanarak sanal ortamda normal davranış sergileyebilir ve bazı kötü amaçlı yazılımlar (arka kapı, Truva atı vb.) Özel talimatlar alınana kadar anormal davranış göstermez [67]. Bu durumda, kötü amaçlı yazılımların anormal davranışlarını toplamak zordur.

Diğer araştırmacılar, doğruluğu artırmak için kötü amaçlı yazılımları tespit etmek / sınıflandırmak için statik kötü amaçlı yazılım özelliklerini ayıklayan statik analizi kullanmayı tercih ederler. Literatürde ele alınan statik kötü amaçlı yazılım özellikleri arasında bayt dizisi [68, 69], kategorize etme [70, 71], işlem kodu [11] ve PE başlık alanları [72] bulunmaktadır. Bayt dizisi üstünden yapılan çalışmalarda çözücü aşamasına gerek duyulmadığı gibi, komutların anlamlandırılması da mümkün değildir. Bir örneği W. Li ve diğerleri tarafından [73] önerilen yöntem ile gösterilmiştir. Bu Çalışmada, ağ trafiği akışlarında veya yerel bir diskte dosya türünün geçerliliğini belirlemek için n-gram analizi ve verimli istatistiksel modelleme teknikleri kullanarak dosyaların ikili içeriğini analiz etmek için bir

yöntem önermektedir. İşlem kodu analizinde ise PE dosyasının çözücü aracı kullanılarak assembly koduna dönüştürülmesi gerekmektedir. Bu tersine mühendislik analizi ile gerçeklenen tespit sistemine örnek olarak [51] ve [74] çalışması gösterilebilir. Bu çalışmalarda çözücü ile elde edilen assembly dosyaları işlemden geçilerek OpCode dizileri elde edilmiş ve n-gram yöntemi ile örüntü oluşturulmuştur. Daha sonrasında bu örüntüler sırasıyla birinde makine öğrenmesi algoritmaları ve Hamming uzaklık hesaplama algoritması ile kullanılarak tespit modelleri oluşturulmuştur.

İşlem kodu dizisi, statik analiz için en önemli kötü amaçlı yazılım özelliklerinden biridir. OpCode dizileri kullanılarak kötü amaçlı yazılım tespit / sınıflandırması için daha önce yapılmış birkaç çalışma önerilmiştir. Santos ve diğ. [75] kötü amaçlı yürütülebilir dosyaları tespit etmek için kötü amaçlı yazılımların ve Bayes Ağı, Destek Vektör Makinesi, Karar Ağacı ve K-En Yakın komşu yöntemleri dahil olmak üzere farklı sınıflandırıcıların n uzunlukta Opcode dizilerini kullanmışlardır. Necmettin Çarkacı ve İbrahim Soğukpınar [76], B.,B. Rad ve diğerleri [77], Abhijit Yewale ve Maninder Singh [78] ise zararlı yazılımları opcode frekanslarına bakarak tespit etmiştir. Zolotukhin ve diğ. [79], iyi huylu yazılımın bir özelliği olarak opcode n-gramları çıkarmış ve iyi huylu yazılım ve kötü amaçlı yazılımları sınıflandırmak için Destek Vektör Makinesi'ni kullanmıştır. Ancak bu iyi huylu yazılımın opcode dizileri, iyi huylu yazılım ve kötü amaçlı yazılım arasındaki farkı önemli ölçüde gösterecek kadar temsili değildir. Cheng Wang ve diğ. [67] zararlı yazılımların opcode komutlarını kullanımında bir benzerlik bulmuştur. Zararlı yazılımlar arasındaki benzerliğe, her bir opcode sekansının metamorfik bilgi entropisi hesaplanarak çıkarılan benzerlik değerinin belirli bir eşik değeri altında olup olmamasına bakılarak karar verildi. kullandıkları makine öğrenmesi algoritması ile (Fast Density-Based Kümeleme) hızlı çalışan bir öznitelik seçme yöntemi (Information Entropy Based Öznitelik çıkarma yöntemi) önermişlerdir. Tao Gong ve diğerleri [80] ise zararlı yazılımları, dosyalardan çıkardığı OpCode dizilerini sıkıştırarak sınıflara ayırmıştır. Onların yöntemi parçalı eşleşme ile öngörü (PPM) yöntemine dayanmaktadır. Sıkıştırılıp PPM skorları çıkarılmış ve Markov Model kullanılarak dizilerin olasılık dağılımları hesaplanmıştır. Eşik değerine bağlı olarak tespit ve sınıflandırma yapılmıştır.

Kod yapısı kullanılarak davranış analizi yapmak da mümkündür. Yuxin ve diğerlerinin [81] önerdiği yöntem sistemin çalıştıracağı komutları statik olarak takip

edip opcode dizilerini oluşturuyor. Bu çalışmada Kötü amaçlı yazılım davranışlarını doğru bir şekilde tanımlamak için, bir programın dinamik yürütülmesini simüle ederek opcode çalışan ağacı oluşturulmuş ve bir çalıştırılabilir dosyanın özelliklerini temsil etmek için opcode n-gram'ları çıkarılmıştır. Jixin Zhang ve diğerleri [82] assembly dosyasını resim dosyasına dönüştürerek zararlı yazılım tespit problemini resim tanımlama problemine dönüştürmüştür. T. Wang ve N. Wu [83] da resim tanımlama problemine dönüştürerek k-en yakın komşu modeli eğitilerek tespit yapmışlardır. Onların çalışmasında farklı olan ise opcode dizilerini resme dönüştürme fonksiyonlarında baş göstermektedir. Bu çalışmada da dönüştürülen resimler kullanılarak eğitilen destek vektör makinesi modelleri kullanılarak tespit yapılmaktadır. Bu dönüşümü gerçekleştirerek derin öğrenme yöntemini kullanan çalışmalar da literatürde mevcuttur [84].

Operasyon komut dizileri ile tespit yapmanın başka bir yolu da Gizli Markov Model kullanmaktır [85, 86, 87, 88, 89]. Gizli Markov Model dizi verilerinden gizli manalar ve örüntü çıkarmak için kullanılan başarılı bir yöntemdir. Bu yöntemin zararlı yazılım tespitinde kullanılması daha çok zararlı yazılım sınıflarının her biri için o aileye mensup numunelerden benzer örüntü çıkarma amacını gütmektedir. Uygulanışı özetlemek gerekirse, Gizli durumlar matrisini hesaba katmadan 1. Kural ile eğitim yapılıp 3. Kural ile test ile puan hesaplar ve eşik değeri aşması ile tespit yapılmaktadır. Her sınıf için bir model oluşturulmuştur. Buradaki asıl ayrım HMM'nin öznitelik seçiminde öne çıkmaktadır. Hamid Divandari ve diğerleri [90] direk OpCode dizilerini öznitelik olarak almak yerine öznitelik seçim ve öznitelik çıkarım algoritmalarını kullanmışlardır. Entropi hesaplama ve Markov blanket yöntemleri kullanılarak dizi içinde alt diziler çıkarılmış, modeller birbirine daha yakın ve daha yüksek anlam içeren alt diziler ile eğitilmiştir. Wing Wong ve Mark Stamp [91] ise her bir metamorfik aile için bir model geliştirmiştir. Onlar zararlı yazılımlardan OpCode dizilerini çıkarmış ve modellerin eğitilmesi için dizileri kullanmışlardır. Stamp gizlenmiş Markov model yöntemini tespit için kullanan ilk kişidir. Metamorfik araçların tespiti için kullanmıştır.

2.5.2. Davranış Analizi Tabanlı Yöntemler

Bilgi sistemleri için kötü amaçlı yazılım tehdidinin çeşitliliği artarken, araştırmacılar alternatif kötü amaçlı yazılım tespit yöntemleri bulmaya çalışmışlardır.

Kötü amaçlı yazılımları tespit etmekten önce, ilk adım bilinmeyen yazılımın bir sistem için tehdit oluşturup oluşturmadığını belirlemektir. Tehditleri ortaya çıkarmanın etkin ve etkili yolu, yazılımın davranışını analiz etmektir. Davranış analizi, yazılım tarafından yürütülen tüm etkinliklerin gözlemlenmesini ve gösterilmesini kapsar, böylece kötü amaçlı olanları yakalamaya izin verir [92]. Çok sayıda çalışma, güvenli bir ortamda çalışarak programların davranışlarını izleyen dinamik analiz tekniklerini uygulamıştır. U. Bayer ve diğ. [93] çok sayıda kötü amaçlı yazılım örneği mutasyon dosyasını analiz ederek kötü amaçlı yazılım programlarının ortak davranışlarını yakalayan bir yöntem önermişlerdir. Jinrong Bai. ve diğerleri [94] API çağrılarının sıklığına dayalı dinamik bir kötü amaçlı yazılım tespit şeması önermişlerdir. Hansen ve diğerleri [95] API çağrılarını izleyerek API çağrı frekanslarını ve API çağrı sekanslarını ayıklayın ve kötü amaçlı yazılım tespiti için makine öğrenme yöntemlerini kullanın. Mohd Shaid. vd. [96], kötü amaçlı yazılım değişkenlerini tespit etmek için API çağrılarını renklerle eşleştirerek kötü amaçlı yazılım davranış görüntüleri oluşturan bir yöntem önermişlerdir. Pektas [97] API çağrısı dizi tabanlı algılama yöntemi önermiş ve çevrimiçi makine öğrenme algoritmalarını kullanmıştır. [98] 'te, işlev çağırma dizileri, kötü amaçlı yazılımı sınıflandırmak için Gizli Markov Modeli ile kullanılabilir. [99] 'te API çağırma dizileri, dizi hizalama algoritmaları ile kullanılır. Algoritmaların verimliliğini artırmak için kritik olmayan API işlevleri elimine edilir. [100.] 'da API çağrısı dizilerine metin madenciliği algoritmaları uygulanmıştır. Belirli bir konumda ve sırayla belirli bağımsız değişkenlerle işlemleri analiz etmek için, bir özellik kümesi her API çağrısı başına işlem, konum ve parametreler içerir. Udayakumar ve diğerleri [101] taşınabilir yürütülebilir dosyalar tarafından yüklenen güvenli olmayan dinamik bağlantılı kütüphanenin algılanmasına odaklanmaktadır. Bu yüklemeler sistemler için ciddi bir tehdide neden olabilir. [102] 'de statik ve dinamik analiz kullanarak anlamlı özellikler elde etmek için hibrit bir yaklaşım önerilmiştir. Yürütülebilir dosyaları analiz etmek için çalıştırılabilir dosyanın imzası, kullanılan DLL listesi ve işlev listesi adı verilen bazı statik ve dinamik özellikler kullanılır. Gandotra ve diğ. [103] ayrıca bir hibrit yöntem önerdi. Yöntemleri, taşınabilir yürütülebilir dosyalar tarafından etkinleştirilen sistem işlemlerinin sıklığını kullanır. [104] 'de bir davranış analizi tekniği sunulmuştur. Dinamik analizle çıkarılan öznitelikler olarak sistem işlemlerini kullanır.

İmza tabanlı yöntemlerin dezavantajlarının üstesinden gelmek için araştırmacılar kötü amaçlı yazılım davranışının analizine yöneldiler. Genellikle, kötü amaçlı yazılım sistem çağrılarını gerçekleştirerek kötü amaçlı davranışlar sergiler. Bu nedenle, sistem çağrıları kod davranışlarının özellikleri olarak görülebilir. Önemli sayıda sistem çağrısı tabanlı kötü amaçlı yazılım algılama yöntemi önerilmiştir [81].

Pektaş ve Acarman [5], API çağrı dizilerini kullanarak çalışma zamanı davranış analizine dayanan özgün bir zararlı yazılım sınıflandırma mekanizması geliştirdi. Zararlı yazılımların API çağrı dizisi içinde bir örüntü yakalamak üstüne çalışan Pektaş, bu belirteç örüntüyü n-gram ve oylama uzman algoritması (Voting expert Algorithm) yardımıyla buldu ve eşleştirme yöntemi ile tespit yaptı. Zhang ve diğerleri [65] OpCode n-gramlarını ve dinamik olarak ölçülen API-çağrım sıklıklarını öznitelik olarak kullanan melez bir zararlı yazılım tespit yöntemi geliştirdi. Yöntemin temelini Temel Bileşen Analizi (principal component analysis - PCA) yöntemi ve derin öğrenme anlayışı oluşturur. Vemparala ve diğerleri [6] önerdiği yöntem ise dinamik zararlı yazılım analizi yaparak çıkarılan API-çağrım dizilerini kullanır. Geliştirilen yöntem Hidden Markov Model yöntemi ile geliştirildi. Xiao ve diğerleri [105] ile Onwuzurike ve diğerleri [63] Android güvenliği için derin öğrenme teknikleri ve Markov zinciri tekniğini kullanarak çalışan birer yöntem önerdiler. Bu çalışmada Derin öğrenme ve Markov zinciri tekniklerinden faydalanılmıştır, fakat önerilen yöntem melez öznitelik ile çalışır ve Windows zararlı yazılımlarının tespiti için geliştirilmiştir. [11]'da gösterildiği gibi, melez modeller zararlı yazılımların atlatma ve saklanma tekniklerini alt etmek üzere önemli avantajlar sağlamaktadır. Başka bir çalışma da melez modeller ile anomali tespiti yaparak zararlı yazılımları üstüne önerilen çalışmadır [106]. Derin öğrenme tekniğini kullanan bir diğer çalışma Safa ve diğerleri tarafından [7] ile sunulmuştur. Onlar sınıflandırma modelini CNN/RNN ile geliştirmiştir Önerdikleri model melez analiz ile çıkarılan öznitelikleri kullanmaktadır. Derin öğrenmenin bir diğer kullanım yolu da Xiao ve diğerleri [38] tarafından sunulan özgün yöntemde IoT cihazlarını zararlı yazılımlardan önlemesi için kullanılmıştır. Katmanlı oto kodlayıcı (SAE) kullanarak geliştirilen yöntem, API-çağrı grafları ile çalışır ve eşleştirme yapılıp eşik değeri aşması ile tespit yapmıştır. [107] ve [84]'de de derin öğrenme yöntemleri kullanılarak tespit yapılmıştır. Zararlı yazılım tespitinde derin öğrenme teknikleri bilinçli ve akıllıca kullanılmalıdır. [13]'deki çalışmalar göstermiştir ki derin öğrenme temelli zararlı yazılım sınıflandırma yöntemleri kandırılabilir. Şu anlaşılmaktadır ki, derin öğrenme

teknikleri bilgisayar savunmacıları için önemli silahlardır fakat bunların uzman bakışı ve öznitelik mühendisliğine ihtiyaçları vardır. Kolosnjaji ve diğerleri [108] de derin öğrenme tekniklerinin zararlı yazılım tespitinde kullanımı ve tespit çalışmalarındaki etkisi üstüne çalışmıştır. Onun çalışmasına göre, derin öğrenme bazı kurtulma saldırıları ile kandırılabilir. Bu yüzden onun çalışmasına göre, Lee ve diğerleri [109]'ün yaptığı gibi ham data kullanan derin öğrenme temelli tespit sistemleri zafiyet göstermektedir. Kolosnjaji ve diğerleri [110] bir başka çalışmasında ise CNN/RNN katmanlarından oluşan yapay sinir ağı kurmuş ve sistem fonksiyon çağrımlarıyla çalışan bir model geliştirmiştir.

Garetto ve diğerleri [14] ve Chen&Ji [111] Zararlı yazılımın bulaşma ve yayılma davranışı üstüne çalışmışlar, bu davranışı tespit etmek için Markov zinciri ile temellendirilmiş birer model geliştirmişlerdir. Karyotis [64] yine Zararlı yazılım yayılma davranışı üstüne bir model geliştirmiş, fakat onun modeli stokastik bir Teknik olan Markov Rastgele Alanları(Markov Random Fields) yönteminden faydalanmıştır. Sartea ve Farinelli [9] de Markov zincirlerini kullanarak davranış tespit yöntemi öne sürmüştür. Bu çalışmada yine Markov zinciri yöntemi kullanılmıştır. Fakat bizim çalışmamızda Markov zinciri bir davranışın istatistiksel analizini yapması yerine, davranış ve kod yapısını belirten dizilim verisini istatistiksel olarak işlemesi için kullanılmıştır. Daha sonra bu işlenmiş veriden anlam çıkarması ve öğrenmesi için derin öğrenme tekniği kullanılmıştır.

3. ÖNERİLEN YÖNTEM

Silahlanma yarışının diğer tarafından bakıldığında, saldırganların ve zararlı yazılım yazarlarının analizden kaçınmak için kullandıkları yöntemler vardır. Durağan analizden kaçınmak için en popüler ve güçlü yöntem kendi kendini değiştirme yöntemidir. Zararlı yazılım kendi kodunda yapısal değişiklikler yaparak farklı bir imza ve kod yapısına sahip benzerlerini üretebilir [12]. Böylece durağan analiz yapan tespit sistemleri ve geleneksel imza tabanlı tespit yöntemleri atlatılabilir. Diğer taraftan, bazı yöntemler ile (hedefleme, tersine ayar testi, erteleme, tetikleme bazlı kodlama ve dosyasız (AVT) saldırı gibi [8]) dinamik analizden de kaçmak mümkündür. Analizden kaçınmak bir yana, Demetrio ve diğerleri [112] belirttiği gibi derin öğrenme teknikleri uygulanmış ileri düzey zararlı yazılım tespit sistemlerini aldatmak da mümkündür. Bu aldatma öznitelik mühendisliği yapılmadığı durumlarda gerçekleşir. İşlenmemiş ham veri ile eğitilmiş derin öğrenme modelleri, literatürde ortaya çıkarılan taktikler ile kandırılabileceği gösterilmiştir [14]. Bu yüzden zararlı yazılımlar ve aileleri dikkatlice incelenmelidir.

Bu çalışmanın amacı Windows ortamında çalışan zararlı yazılımların tespitini yapmaktır. Bunun için zararlı yazılımlar hem dinamik hem durağan analiz yöntemleriyle incelenmiştir. Böylece statik veya dinamik analiz sınırlamaları ortadan kalkmaktadır. Çalıştırılabilir dosyanın kod yapısı ve çalışma zamanı davranış bilgileri istatistiksel öznitelik olarak ele alınmaktadır. Sistem tüm resmi tam anlamıyla yakalayacak şekilde tasarlanmıştır. Yaklaşımdaki özgünlük, istatistiksel özellikleri ön plana çıkaran Markov zinciri geçiş matrisleri ile derin öğrenme tekniği olan evrişimsel sinir ağının (Convolutional Neural Network) birlikte kullanılması ve melez analiz yaparak hem davranış hem kod yapısı verilerinden faydalanılmasıdır. Dinamik analiz ile yazılım davranışlarını, diğer bir deyişle API-çağrım dizilimlerini, durağan analiz ile OpCode dizilimlerini çıkararak istatistiksel bir yöntem olan Markov zincirleri inşa edilmiş, bu Markov zincirleri birleştirilerek eğitme ve tespit yapılmaktadır. Sistem otomatik olarak Windows ortamında çalışacak şekilde geliştirilmiştir. Bu çalışmada önerilen tespit yönteminin temelini hem durağan hem dinamik analiz yaparak çıkarılan öznitelikler ile eğitilmiş derin öğrenme modeli oluşturmaktadır. Böylece derin öğrenme aldatmasından kaçınılmıştır.

Aynı aileye ait olan zararlı yazılımlar benzer davranışlar göstermekte ve kod yapılarında benzerliklere rastlanmaktadır [65]. Bu çalışmada istatistiksel veri kaynağı ve derin öğrenme teknikleri kullanılarak geliştirilmiş zararlı yazılım tespit yöntemi önerilmiştir. Çalışmanın hedefi zararlı davranışa sebep olabilecek yapı ve davranış örüntüsü bulmaktır. Bunun için, davranış analizi API-çağrım dizileri ile ve yapı analizi ise Operasyon kodu (OpCode) dizileri ile incelenmiştir. Bu incelemeler sonucunda istatistiksel bir yöntem olan Markov zinciri ile anlamlı veri elde edilebileceği görülmüştür. Her iki diziden çıkarılmış zincir çıktıları birleştirilerek, derin öğrenme yöntemi ile geliştirilen model eğitilmiştir. Eğitim modeli evrişimli sinir ağları (Convolutional neural network CNN), maksimum havuzlama (max-pooling) ve yoğunluk katmanları (dense-layer) içermektedir. Bu sayede, API-çağrı çiftinin olasılık değeri kombinasyonunu kullanıp iç değişkenleri optimize etmek için hesaplama yapan bir karar modeli eğitilir. Her modülün ve ilişkinin detaylı açıklaması verilmiştir.

Yöntem 4 adımdan oluşmaktadır. İlk olarak zararlı yazılım analizleri yapılmaktadır. Güvenli bir ortamda yapılan dinamik analiz ile yazılım numunelerinin davranışları görüntülenmekte ve json formatında raporlanmaktadır. Ayırıcı çözücü (disassembler) ile de numunelerin tümleşke (assembly) kodu çıkarılmıştır. Bu analiz raporlarından API-çağrım dizisi ve OpCode dizisi çıkarılmaktadır. Her iki diziden Markov zinciri oluşturulur ve geçiş matrisleri birleştirilir. Birleştirilmiş matrisler derin öğrenme modelini eğitmek için kullanılmaktadır. Test sürecinde ise yine zararlı yazılımın analizi yapılıp birleştirilmiş geçiş matrisi oluşturulmuştur. Geçiş matrisini girdi olarak alan model, zararlı yazılım olup olmadığına karar vermektedir. Her modülün ve ilişkinin detaylı açıklaması sonraki bölümlerde verilmektedir.

3.1. Analiz

Uygulama programlama arayüzü (API) işletim sistemi tarafından sunulan işlem kümesidir. API işlem çağrımları sayesinde ağ yönetimi, bellek yönetimi, kayıt defteri operasyonları, dosya giriş/çıkışları, proses ve izlekler ile alakalı işlemler yapılabilmektedir. Her bir çağrım, sistem için ayrı bir işlemi temsil etmektedir. O çağrım sonucunda, çağrının girdileri ile bilgisayarda bazı değişiklikler meydana gelir. Bu yüzden API-çağrım dizilimi mikro boyutta yazılım davranışı hakkında bilgi

vermektedir. Yazılım davranışının zararlı olması, bilgisayarda istenmeyen etkilere sebep olması anlamına gelmektedir [55]. Bunu ölçmek ve zararlı yazılımı tespit etmek için, API-çağrımları ile davranışı anlamlandırmanın doğru olacağı düşünülmüş ve bu düşünce çalışmanın temelini oluşturmuştur.

Tablo 3.1: Zararlı Yazılımların Yaptığı en uzun ortak alt diziler

Tür	Zararlı Davranış Dizisi
Solucan	GetSystemDirectoryA-NtCreateFile-GetFileSize-NtClose
Virus	NtOpenKey-LoadStringA-NtAllocateVirtualMemory-NtFreeVirtualMemory-LoadStringW-NtDuplicateObject-GetFileType-NtCreateMutant-CreateThread-NtAllocateVirtualMemory-NtOpenFile-NtClose-GetSystemWindowsDirectoryA-GetFileAttributesW-NtCreateFile-SetFilePointer-NtReadFile- NtClose
Kök gizleme aygıtı	NtFreeVirtualMemory-NtUnmapViewOfSection-NtClose-GetSystemMetrics-LdrUnloadDll-NtClose-LdrGetDllHandle-LdrGetProcedureAddress-NtClose
Arka Kapı	NtOpenKey-GetSystemTimeAsFileTime-LdrLoadDll-LdrGetProcedureAddress-NtClose-GetSystemMetrics
Trojan	NtOpenSection-NtMapViewOfSection-NtUnmapViewOfSection-NtClose-NtAllocateVirtualMemory-FindResourceExW-LoadResource-FindResourceExW-LoadResource-NtClose
Yığınlayıcı	RegOpenKeyExW-GlobalMemoryStatusEx-LdrLoadDll-NtDuplicateObject-GetComputerNameW-NtDuplicateObject-NtCreateFile-NtSetInformationFile

Sistemin ilk hedefi zararlı davranış örüntülerini bularak tespit yapmak olmasına rağmen, dinamik analiz ile ortaya çıkarılamayan bilgileri de gözden kaçırmamak için yardımcı olarak durağan analiz ile elde edilen işlem kodu (OpCode) dizileri de öznitelik olarak kullanılmıştır. Makine kodu CPU'nun yapması için komutlardan meydana gelmektedir. Her komut; işlem kodu, kaynak operand ve hedef operandan oluşmaktadır. İşlem kodları programın görevlerini yerine getirmek için CPU tarafından alınan talimatı belirtmektedir. Operandlar ise işlem kodlarının etkileyeni ve etkilenenidir. Önerilen sistem yazılımın ne yaptığı ile ilgilendiği için tüm komut satırı içinde sadece işlem kodlarını kullanmaktadır [113]. İkili kodları olan çalıştırılabilir dosyalar, tek tek ayırıcı (disassembler) ile çözümlenmiş ve

makine kodu (assembly) dosyaları elde edilmiştir. Bu dosyalar daha sonrasında yazdığımız program ile OpCode dizilerine çevrilmişlerdir. OpCode dizileri tablo3'de belirtilen işlem kodlarından oluşmaktadır**.** API-çağrımlarının yanında veri kaynağı olarak kullanılan işlem kodları, yazılım davranışlarını daha iyi analiz etmek için önemli ölçüde katkı vermektedir.

Alt dizileri kullanarak bir imza tabanlı tespit sistemi oluşturmanın sakıncası tablo x3.'de gözükmektedir. Akıllı olmayan bir örüntü bulma ve karşılaştırma yaparak geliştirilen tespit sistemini aldatmak ve atlatmak çok zor olmayacaktır. Keza, anlamsız şekilde yapılan bir API-çağrısı bile dizeyi bozabilir ve bu tip sistemi atlatabilir. Kaldı ki yeni zararlı yazılımlar oldukça gelişmiş atlatma yöntemleri kullanmaktadır. Bu çıkarım çalışmayı durumsuz (stateless) istatistiksel analiz yapmaya ve akıllı bir sistem geliştirmeye yöneltmiştir.

Tablo 3.2. Zararlı davranışlar ve o davranış için yapılan API çağrımları

Zararlı Davranış	API çağrım Dizisi
İndirme ve Çalıştırma	URLDownloadToFile, ShellExecute
TCP Port bağlantısı	WSAStartup, socket
Ağ Trafiği izleme	Socket, bind, WSAloctl, recvfrom
Kendini Silme	GetModuleFileName, ExitProcess, DeleteFile
Açıldığında Aktif Olma	RegOpenKeyExW, RegQueryValueExW, RegCloseKey

Önerilen tespit sisteminin ilk ve en önemli ayağını dinamik analiz oluşturmaktadır. Dinamik analiz kum havuzu yardımıyla yapılmaktadır. Bu çalışmada kullanılan veri kümesindeki tüm numuneler sırayla güvenilir ortamda çalıştırılır ve çalışan bilgisayar izlenir. Bu gözetleme işlemi sonunda bilgisayarda olan tüm değişiklikler ve yazılımın çalışma anı hareketleri raporlanır. Sistem bu raporlardan API-çağrım dizilerini çıkarmaktadır.

3.2. Markov Zinciri

Markov zincirleri stokastik süreçlerin temel bir parçasıdır. Bu, sistemin sonraki durumu önceki durumlarından bağımsız olarak ancak ve ancak o anki durumuna bağlı olmasıdır. Diyelim ki S= {s_1, s_2, s_3, ..., s_r} durumlar kümesi olsun. Öyleyse Markov zinciri şöyle tanımlanmaktadır: S uzayında, eğer bir stokastik süreç X = {X_n, n € $\mathbb{N}$},

$$\text{her bir } n \geq 0 \text{ için, } X_n \in S,$$

$$\text{her } n \geq 1 \text{ için ve her } i_0, i_1, \ldots, i_n \in S \text{ için}$$

$$\mathbb{P}(X_n = i_n \mid X_{n-1} = i_{n-1}, \ , X_0 = i_0) = \mathbb{P}(X_n = i_n \mid X_0 = i_0) \quad (1)$$

ise X Markov zinciridir.

Tablo 3.3: Markov Zincirinin Geçiş Matrisini Bulma algoritması

Algoritma 1. MZO- Markov Zinciri Oluştur

Girdi: DurumKumesi ≠ Ø, S ≠ Ø, N = |S|, B = |DurumKumesi|, M $\in R^{B \times B}$

1: Eğer N çift ise;
2: Eğer N>2 ise;
3: o← N/2 //Dizinin ortasındaki elemanın indisi bulunur
4: oSeq ← S[c: c+1] // Dizinin ortasındaki 2 eleman alınır
5: oSeq ve DurumKümesi ile M matrisini güncelle
6: rseq ← S[0:N/2]
7: lseq ← S[N/2:N]
8: MZO(rseq, DurumKumesi, M)
9: MZO(lseq, DurumKumesi, M)
10: Değilse;
11: S ve DurumKümesi ile M matrisini güncelle
12: Eğeri Bitir.
13: Değilse; //Eğer çift tek ise
14: Eğer N>3 ise;
15: o ← N/2 +1 //Dizinin ortasındaki elamanın indisi bulunur
16: oSeq ← S[o-1:o+1] //Dizinin ortasındaki 3 eleman oSeq'e alınır
17: oSeq ve DurumKumesi ile M matrisini güncelle
18: rSeq ← S[0:o-1]
19: lSeq ← S[o+1:N]
20: MZO(rseq, DurumKumesi, M)
21: MZO(lseq, DurumKumesi, M)
22: Değilse;
23: S ve DurumKümesi ile M matrisi güncelle
24: Eğeri Bitir.
25:Eğeri Bitir.

Görüleceği üzere Markov zinciri, dizinin içerdiği durumların değişimlerindeki olasılık dağılımından oluşmaktadır. Markov zinciri X için t anındaki geçiş matrisi P_t, durumlar arasındaki geçiş olasılıklarının tutulduğu bir matristir.

Geçiş matrisi için her bir gözün matematiksel tanımlaması şöyle yapılmaktadır:

$$(P_t)_{i,j} = \mathbb{P}(X_{t+1}=j \mid X_t = i), \quad i, j \in S. \qquad (2)$$

Bu, matrisin her satırının bir olasılık vektörü olduğu ve vektördeki elemanların toplamlarının 1 olduğu anlamına gelmektedir.

Bu makalede, Markov zinciri analizi ile yapılan durum analizinde zaman faktörü 2 değişkenlidir; şimdi t=0 ve sonra t=1. Matristeki her göz (entry) şimdiki ve sonraki durum geçiş olasılığını tutmaktadır. Şimdiki ve sonraki durum olasılığı bilgisi yazılımın zararlı davranışa sahip olduğu konusunda bir bilgi içermektedir. Geliştirilen sistem, bu bilgi örüntüsünü kullanarak tespit yapmaktır.

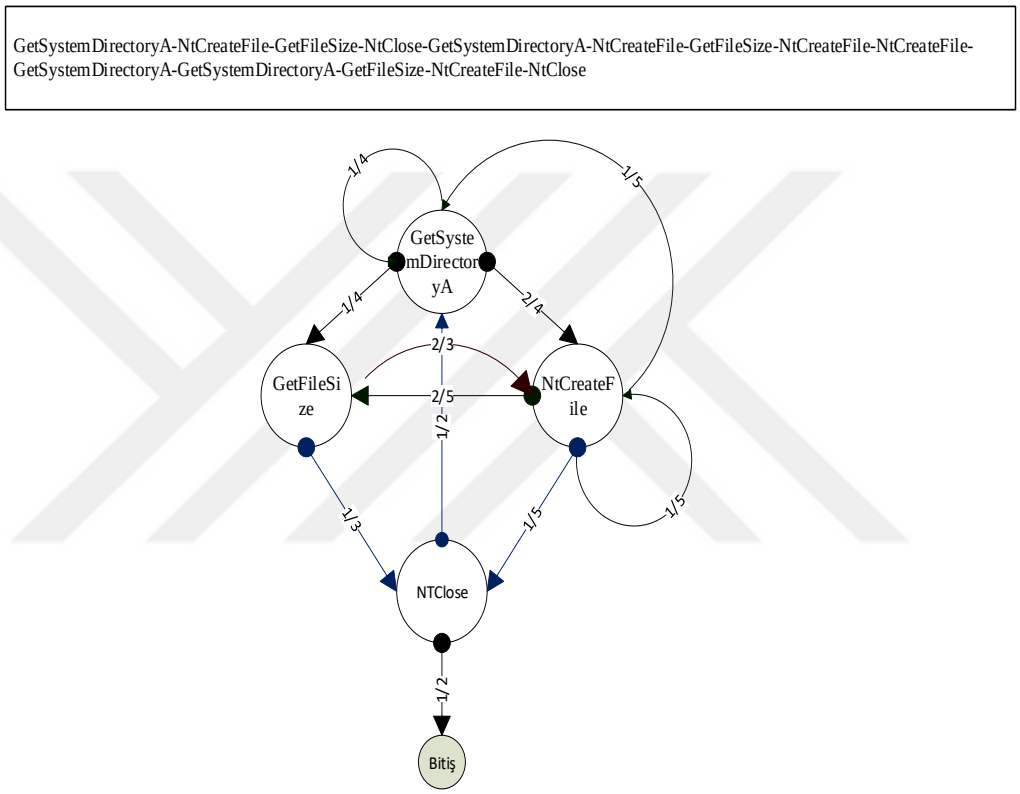
Geçiş matrisinin en önemli özelliği durumsuz sıralama olasılıklarını içermesidir. Bu olasılıklar dizi içinde geçen her eleman için bir sonrakinin gelme olasılığıdır. Böylece her iki eleman için bir olasılık değeri belirlenir. Geçiş matrisinde her bir durum için geçiş olasılıkları vardır. Öyle ki satır kısmına denk gelen durumdan sütun kısmına denk gelen duruma geçiş olasılığı o hücrede belirtilen olasılık değeridir.

Özet olarak söyleyecek olursak Markov zincirleri sistemin bir durumdan başka bir duruma geçişini modellemek için kullanılır. Bu çalışmada analiz edilen dizi içindeki her bir eleman dizinin o sıradaki durumu olarak ele alınmış ve geçiş matrisleri bu varsayım ile hesaplanmıştır. Durumlar arasındaki geçişler, yeni bir duruma geçme olasılığı veren koşullu bir olasılık dağılımına tabidir. Bu koşullu olasılık dağılımı geçiş matrisi ile sağlanmaktadır. Her bir dizi için bir Markov zinciri kurulur ve Markov zincirleri geçiş matrisleri ile ifade edilmektedir. Geçiş matrisi Algoritma 1.'de belirtilen yöntem ile oluşturulmaktadır.

Açıktır ki geçiş matrisi NxN boyutundadır, N sayısı durum kümesi eleman sayısına denk gelir. Durum matrisini çıkarmak için yinelemeli algoritma, Algoritma 1. de gösterildiği gibi yazılmıştır. Algoritmaya göre her adımda dizi 2'ye bölünerek her iki kısmı için tekrar çağrılır, ta ki 2 ya da 3 eleman kalana kadar. Eğer dizinin eleman sayısı tek ise ortadaki 3 sayı ile, eğer çift ise ortadaki 2 sayı ile M geçiş matrisi güncellenir. 2 sayılı dizi ile güncellemede bir, 3 sayılı dizi ile güncellemede

iki arttırma işlemi olur. Algoritma ilk çağrılışında M matrisinin tüm elemanları 0 olmalıdır. Karmaşıklık analizi, N dizinin eleman sayısı ve B da durum kümesinin eleman sayısı olduğu yerde, O(BlogN) olur.

Oluşturulan Markov zinciri Şekil 3.'de örnek üstünde gösterilmiştir. Markov zincirleri sistemimizde geçiş matrisleri ile ifade edilmektedir. Geçiş matrisleri her bir numune için 2 tane çıkarılmaktadır; API-call dizisi için ve OpCode dizisi için. Bu matrisler birleştirilir.



Şekil 3.1: API-çağrım dizisi ve Markov Zinciri Dönüşümü

Birleştirilmiş matriste, her hücrede iki eleman olacaktır, ilki API fonksiyon durumu geçiş olasılığı diğeri ise işlem kodu durumu geçiş olasılığıdır. Veri kümesinde var olan her bir numune bu birleştirilmiş matris ile ifade edilmektedir ve derin öğrenme modeline bu hali ile verilmektedir.

3.3. Derin Öğrenme Modeli

Derin öğrenme yöntemi en saf haliyle çok katmanlı algılayıcı (Multilayer perceptron) olarak tanımlanabilir. Genel olarak yapay sinir ağın yegâne amacı herhangi bir *f** fonksiyonuna yakınlaşmaktır. Yapay sinir ağı y = *f(x;θ)* gönderimini tanımlar ve fonksiyon f sonuçlarına en yakın $\boldsymbol{\theta}$ değerlerini hesaplar [66]. Eğitim aşamasında verilen y ve x değerleri ile $\boldsymbol{\theta}$ değişkenleri optimize edilir. Test aşamasında ise verilen x ve eğitim değişkenleri ile y tahmin edilir. Eğitim aşamasında yapılan işleme loss fonksiyonu denir ve bu deneyde kullandığımız loss fonksiyonu şu şekilde verilmektedir:

$$CE = -\sum_i^C f*(x_i) log\,(f(x_i;\theta)) \qquad (4)$$

CE fonksiyonu çapraz entropi (cross entropy) olarak isimlendirilmektedir. f fonksiyonu derin öğrenme modelinin aktivasyon fonksiyonunu ifade etmektedir. Bu fonksiyon her nöronda yapılan işlemdir ve her katman için farklı bir aktivasyon fonksiyonu belirlenebilmektedir. Deneylerde kullandığımız aktivasyon fonksiyonu ReLu (Recrified linear unit) fonksiyonudur ve şu şekilde ifade edilmektedir:

Tablo 3.4: Derin Öğrenme Model Özeti

Katman (tür)	Çıktı	Değişken
CNN	262, 262, 16	304
Maksimum Havuzlama	87, 87, 16	0
CNN	85, 85, 32	4640
Maksimum Havuzlama	28, 28, 32	0
CNN	26, 26, 64	18496
Maksimum Havuzlama	8, 8, 64	0
CNN	6, 6, 96	55392
Geçiş	3456	0
Yapay sinir ağı	2	6914
Eğitilen Toplam Değişken Sayısı:		85,746

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \qquad (5)$$

Her adımda f fonksiyonu gerçek değer olan *f** 'a yaklaşacak şekilde optimize edilir. Bu çalışmada Adam optimizer [114] yöntemi kullanılarak değişkenler optimize edilmiştir. Derin öğrenme modelimiz tablo 5'de gösterilen katmanlardan

oluşmaktadır. Bu katmanlar evrişimsel sinir ağı (CNN) ve maksimum havuzlama yöntemlerini içermektedir. Bu yöntemler [66]'de anlatıldığı gibi, çok katmanlı algılayıcı 'da yapılan çoklu işlemleri daha az değişkenle yapılmasını sağlamaktadır. Maksimum havuzlama ile de nöron sayısını azaltmakta ve değerler üstünde sadeleştirmeye gidilmektedir.

Zararlı yazılım tespiti için geliştirilen derin öğrenme modeli 4 adet CNN katmanı ve her katman arasına maksimum havuzlama katmanı içermektedir. Karar katmanından bir önceki katmanda ise tüm nöronların birbirine bağlandığı yoğunluk katmanı kullanılmıştır. Model girdi olarak 264x264x2 boyutlu bir matris almaktadır. Çıktı olarak da zararlı yazılım ya da iyicil yazılım olduğuna karar vermektedir. Toplam optimize edilen değişken sayısı 85,746'dir.

4. YÖNTEMİN UYGULANMASI

4.1. Veri Kümesi

Deneyde kullanılan veri kümesi zararlı yazılımlar Vxheaven'dan [115], iyicil yazılımlar ise Windows sistem dosyalarından ve Cygwin dosyalarından derlenmiştir. Bu çalışmada 2 veri kümesi kullanılmıştır. Bu veri kümelerinden A olarak isimlendirdiğimiz veri kümesi 2000 adet iyicil dosyadan, 2000 adet zararlı dosyadan oluşmaktadır. Bu veri kümesi çalışmanın deneylerinin yapıldığı ve diğer deneylerle karşılaştırma yapmak için kullanılan veri kümesidir. Sadece bu veri kümesi ile karşılaştırma yapılmasının sebebi, diğer çalışmaların büyük veri kümesi ile yavaş çalışması ve kaynak yetersizliğine sebebiyet vermesidir.

Tablo 4.1: Deneyin veri kümesinde bulunan zararlı ve iyicil yazılım sayıları

	Numuneler	Sayılar
A	İyicil	2000
	Zararlı	2000
B	İyicil	6857
	Zararlı	8936

B ismi ile adlandırılan veri kümesi ise daha kapsamlı ve daha fazla numuneyi içermektedir. Bu veri kümesinde iyicil olarak, zararlı yazılımlara benzerliği bir nebze daha yüksek olan Download.com sitesinden indirilen iyi huylu yazılımlar da kullanılmıştır. Bu veri kümesinin amacı, işleri zorlaştırarak çalışmanın doğruluğunu kanıtlamaktır. Veri kümelerinin numune sayıları tablo 6.'da içerikleri ise tablo 7.'de gösterilmiştir. Görüldüğü gibi büyük olan B kümesinde daha fazla sayı olmasının yanında zararlı yazılım çeşitliliği bakımından da bir fazlalık söz konusudur. A kümesi B kümesi içinden seçilen bir alt kümedir. 2 küme üstünde çalışılmasının sebebi referans yöntemlerle karşılaştırma ihtiyacıdır.

Tablo 4.2. Zararlı yazılım kümesi tür dağılımları

Türler	A	B
Arka Kapı	410	1711
Constructor	8	24
DoS	1	10
Exploit	9	60
Flooder	3	27
HackTool	6	22
Hoax	8	53
Packed	3	9
Kök gizleme aygıtı	25	104
SpamTool	-	2
Spoofer	-	2
Trojan	844	4016
Trojan-DDoS	1	3
Trojan-GameThief	209	952
Trojan-IM	5	18
Trojan-Notifier	1	2
Trojan-PSW	152	544
Trojan-Ransom	-	4
VirTool	5	18
Virus	209	945
Worm	100	410

4.2. Değerlendirme

Geliştirilen tespit sisteminin başarımı gerçek pozitif oranı ve yanlış pozitif oranı ile ölçülmüştür. Bu oranları incelemeden önce gerçek pozitif (TP), gerçek negatif (TN), yanlış pozitif(FP) ve yanlış negatif (FN) değerlerini anlamak gerekmektedir. Bu değerler genel olarak o numunenin gerçek sınıfı ve modelin tahmin ettiği sınıfını karşılaştırmak için kullanılan ölçüm değerleridir.

Bizim deney üstünden düşünülecek olursa, zararlı ve iyi huylu olarak iki sınıf vardır. Zararlı olarak 2000 ve iyi huylu olarak 2000 numune ile deney yapılmıştır. Zararlılar için değerlendirecek olursak; TP değeri, gerçekte zararlı olan ve deney sonucunda modelin zararlı olarak tespit ettiği numune sayısını göstermektedir. TN ise gerçekte iyi huylu olup deney sonucunda modelin iyicil olarak tespit ettiği numune sayısını belirtmektedir. FP değeri gerçek sınıfı iyicil olup modelin zararlı olarak tespit ettiği ve aslında yanlış tespit edilen numunelerin sayısını vermektedir. FN ise gerçek sınıfı zararlı olup modelin iyicil dediği ve aslında yanlış tespit edilen numune sayısını göstermektedir. TPR gerçek pozitif (TP) değerlerinin gerçekte pozitif olanlara (TP + FN) oranı ile bulunur. FPR ise FP değerlerinin gerçekte negatif olanlara (FP + TN) oranı ile hesaplanmaktadır.

4.3. Deney Sonuçları

Deneyin başlangıcı olarak yapılan işlem analiz sürecidir. Durağan analiz ve dinamik analiz olmak üzere 2 tip analiz yapılmıştır. Dinamik analiz sanal makine ile güvenli ortam oluşturularak yapılmıştır. Bu işlem için Cuckoo kum havuzu kullanılmıştır. Cuckoo'yu çoklu kullanıma hazırlamak için ve tüm dinamik analiz ortamını hazırlayıp otomatize etmek için çalışma zamanı gözetleme modülü kullanılmıştır. Cuckoo verilen numuneyi güvenli sanal makinede çalıştırmak, sanal makinede olup biteni izlemek ve raporlama açısından oldukça kullanışlı ve faydalı bir araçtır. Yazılımları sanal makinede çalıştırıp izleme aşamasının sonunda json formatında rapor dosyaları oluşturulmuştur. Modül bu dosyalardan sıralı API-çağrım dizilerini çıkartmaktadır. Durağan analiz için distorm3 kütüphanesi kullanarak yazdığımız ayırıcı modülü ile gerçekleştirilmiştir. Ayırıcı modülü veri kümesindeki numunelerin tek tek makine kodu dosyalarını (assembly file) oluşturur ve sıralı OpCode dizilerini çıkartmaktadır.

Tablo 4.3: Zararlı Yazılım tespit sistemi deney sonuçları

		Eğitim		Test	
		TPR	FPR	TPR	FPR
A	İyicil	0,9872	0,0726	0,9885	0,0879

	Zararlı	0,9274	0,0128	0,9121	0,0115
B	İyicil	0,9787	0,0259	0.9845	0.0303
	Zararlı	0,9741	0,0213	0.9697	0.0155

Her dizi için geçiş matrisleri oluşturmak amacıyla Markov Zinciri Oluşturma modülü yazılmıştır. Bu modül Algoritma 1. Temelli çalışmaktadır. Her dizi için Durum Kümeleri tipine göre değişmektedir. Durum Kümesi; API-çağrım için deneyde çıkarılan tüm API fonksiyonlarından, OpCode için ise en çok kullanılan OpCode'lardan oluşmaktadır. Bu modülün çıktısı olarak her zararlı yazılım için 2 adet 264x264'lik matris oluşmaktadır. Bu matrisler birleştirilerek numuneyi deneyde temsil eden birleştirilmiş matris oluşturulur. Matrislerin satır ve sütunları, en çok kullanılan durumlardan en az kullanılan durumlara göre sıralı olacak şekilde sıralanmıştır.

Derin öğrenme modelini gerçeklemek için keras [116] kütüphanesi kullanılmıştır. Python'nun 3.5 versiyonu kullanılmış ve bütün tespit sistemi bu dil ile gerçeklenmiştir. Test sonuçları tablo 2.'de verilmiştir.

Yapılan çalışmada test sonuçlarının yüksek çıkmasındaki ana pay dinamik analiz ile elde edilen davranış özniteliklerindedir. Davranış analizi ile zararlı yazılım ve iyicil yazılım farkı ortaya konmuş, durağan analiz ile de bu fark daha da belirgin hale getirilmiş ve destekleyici olmuştur.

Tablo 4.4: Diğer Yöntemler ile Karşılaştırma

Yöntem	TPR	FPR	ACC
MZO-DÖ	0.950	0.049	0.968
Rastgele Orman	0.943	0.062	0.959
J48	0.925	0.088	0.935
Destek Vektör Makinesi	0.718	0.685	0.727
İkili Ham Veri -DÖ	0.649	0.327*	0.729*

Tablo 4.4.'da referans yöntemler ile bu çalışmada sunulan yöntemin başarı karşılaştırması verilmiştir. Karşılaştırma ortamını hazırlamak için, A veri kümesi kullanılarak melez öznitelikler çıkarılmış ve dizilerin histogram analiz verisi oluşturulmuştur. Histogram dosyaları birleştirilip weka [117] aracı kullanılarak

rastgele orman, J48 ve destek vektör makinesi algoritmaları ile test edilmiştir. Diğer karşılaştırma ise referans olarak aldığımız derin öğrenme yöntemi kullanılarak yapılmıştır. Bu karşılaştırma deneyinde ise ikili dosyalar işlenmeden [109]'de belirtilen yöntem ile resim dosyasına çevrilmiş ve her bir numune için 1024x1024'lük resimler oluşturulmuştur. Bu Resim dosyaları ile bu çalışmada kullandığımız derin öğrenme modeli ile testler yapılmıştır. Deneylerde A veri kümesi kullanılmıştır. Bunun sebebi ise veri kümesinin büyük olması, bu deneylerin çalışma zamanı ve kaynak kullanımını büyük oranda arttırmasıdır. Deneylerin başarım sonuçları Tablo 8.'de belirtilmektedir. Bu çalışmalar ile önerilen yöntem karşılaştırıldığında, başarım oranı daha yüksek olduğu net bir şekilde görülmektedir.

Tablo 4.5: Derin Öğrenme ile Tespit yapan diğer Araştırmalar ile Karşılaştırma

REF	Öznitelik	Model	Amaç	Doğr.
[17]	API & Opcode	PCA + NN	Tespit	95.1
[20]	API çağrı Graf	SAE + DT	Tespit	99.1
[24]	İkili Ham Dosya-Resim	CNN +LSTM	Tespit	97.1
[26]	System-Call	CNN +LSTM	Snf.	89.4
MZO-DÖ	API & Opcode	CNN	Tespit	97.8

Tablo 4.5'de ise derin öğrenme yöntemi kullanılarak yapılan diğer çalışmalar ile karşılaştırmalar gösterilmektedir. Bu araştırmada ortaya çıkarılan yöntem, sadece evrişimsel sinir ağı kullanmasına rağmen oldukça yüksek başarım sağladığı görülmüştür. Fakat önerilen yöntem dinamik bir çalışma ile tespit yaptığı için, hızlı karar vermesi uygulanabilirliği açısından önemli bir parametredir. Daha basit bir derin öğrenme modelinin tercih edilmesi ve öne sürülen modelin farklı yöntemlerle karmaşıklaştırılmamasının sebebi, bu çalışmanın yapısında var olan statik analiz ile nitelik sadeleştirme ile kazanılan kaynak kullanım miktarındaki azalma ve test süresini kısaltma özelliklerinden vazgeçmek istenmemesidir.

5. SONUÇLAR ve GELECEK ÇALIŞMALAR

Önerilen tespit sisteminde, zararlı yazılımların davranışları ve bu davranışlara sebep olan kod yapılarını tecrübe ederek öğrenen bir model temel alınmıştır. Bu sebepledir ki zararlı yazılımlara ait olan davranış bilgileri ayrıntılı olarak incelenmiş, iyicil olarak etiketlenen yazılımlarda bu davranışlara nadiren rastlanmasından dolayı tespit sistemi yüksek başarım vermiştir.

Bir başka konu ise zararlı yazılımların sınıflara ayrılmasıdır. Bu sınıflara ayırma 2 türlü olabilir; ya davranışı baz alınarak virüs, solucan, arka kapı gibi isimlerle ayırmak, ya da uzmanlar veya anti virüsler tarafından verilen zararlı yazılım yazarını, yazılım cinsini, kod ailesini (birbirine benzer kod gruplarını kullanan yazılımlar) belirtmek için koyulan isimlerle ayırmak. Her iki türlü sınıflandırmanın kendi içinde avantaj ve dezavantajları vardır. Fakat dezavantajların altında yatan ana soyut sebep, zararlı yazılım yazarlarının kendi içinde bir ayrıma veya kurala bağlı kalmadan geliştirme yapmasıdır. Bu yüzden herhangi bir sınıflandırma yapıldığında istisnai bir deneğe rast gelme ihtimali oldukça yüksektir.

Zararlı yazılımların davranışlarına göre sınıflandırmanın tespit sistemlerindeki gibi yüksek sonuçlar vermediği literatürde var olan çalışmalarda görülmektedir [67, 107, 110]. Bu sonucu bu çalışmada yaptığımız araştırmada da görmekteyiz. Bunun başlıca sebebi, zararlı yazılım numunesinin kesinlikle bir davranışa uyma zorunluluğu olmamasıdır [67]. Özellikle yeni internete sürülen yazılımlarda daha komplike bir yapıda olduğu gözlemlenmektedir. Mesela bir zararlı yazılım hem solucan özelliği hem de Truva atı özelliği taşıyabilir. Yani asıl davranışı etiketlendiği davranış olsa bile, zararlı yazılım diğer etiketteki yazılımlarla aynı davranışı da göstermeye eğilimli olabilmektedir. Bu durumda o yazılımı diğer etiketlere ait olarak tespit etmek, yanlış bir sınıflandırma olmamasına rağmen tek tip etiketlenmiş veri kümesi için hatalı sonuç olarak gözükmektedir. Bunun yanında çoğu zararlı yazılım benzer gizlenme ve saklanma taktiklerini kullanmaktadır. Bu durumlar da başarımı

düşürmektedir. Sınıflandırma sistemlerindeki bu bulanıklık, öne sürülen sistemlerin başarım oranlarını da tartışmaya açmaktadır.

Gelecek çalışmalarda, bu araştırmadan edindiğimiz tecrübe ve birikimlerle zararlı yazılımların davranışlarını sınıflandırma ya da kümeleme yöntemi geliştirilecektir. Bu yöntem, davranışları modellemeyi ve modellenen davranışların birden fazla zararlıda var olup olmadığını test etmeyi temel alacaktır. Maalesef ki günümüzde zararlı davranışların etiketlenmiş veri kümesi bulunmamaktadır. Çalışmamızda gördüğümüz gibi böyle bir etiketli veri oluşturulması da oldukça zordur. Farklı API çağrım ya da komut dizsi kombinasyonları, aynı davranışa sebep olabilmektedir [118]. Bu yüzden zararlı davranışlar için bir örüntü modeli geliştirilmeli ve bu örüntülerin birbirine benzerliğini kontrol edecek uzaklık (distance) hesaplama fonksiyonu önerilmelidir. Daha sonrasında kümeleme yöntemleri kullanılarak davranışlar benzerlik ölçüsünde farklı kümelere dağılacaktır. Bildiğimiz "asıl" davranış etiketi de bu kümelerin ne olduğunu tahmin etmek için faydalı bilgiler verecektir. Bu tezde önerilen çalışma, zararlı davranış küme analizinin başlangıcı, o çalışma da zararlı davranış etiketlerinin oluşturulması için faydalı olarak davranış tespitinin ön ayağını oluşturacaktır.

KAYNAKLAR

[1] Chen, P., Huygens, C., Desmet, L., & Joosen, W., (2016), "Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware", IFIP International Conference on ICT Systems Security and Privacy Protection, 323-336, 30 Mayıs – 1 Haziran.

[2] MalwareBytes-Labs, 2019 State of Malware, (2019), https://blog.malwarebytes.com/malwarebytes-news/ctnt-report (Erişim Tarihi: 15/04/2019).

[3] AV-Test-Institute. 2019 New Malware, (2019), https://https://www.av-test.org/en/statistics/malware/ (Erişim Tarihi: 15/04/2019).

[4] Rajeswaran, D., DiTroia, F., Austin, T. H., &Stamp, M., (2018), "Function call graphs versus machine learning for malware detection", Guide to Vulnerability Analysis for Computer Networks and Systems, 259-279, Springer.

[5] Vemparala, S., DiTroia, F., Corrado, V. A., Austin, T. H., Stamo, M., (2016), "Malware detection using dynamic birthmarks", Proceedings of the 2016 ACM on International Workshop on Security and Privacy Analytics.

[6] Pektaş, A., &Acarman, T., (2017), "Malware classification based on API calls and behaviour analysis", IET Information Security, 12(2), 107-117.

[7] Safa, H., Nassar, M., & Al Orabi, W. A. R., (2019), "Benchmarking Convolutional and Recurrent Neural Networks for Malware Classification", 15th International Wireless Communications & Mobile Computing Conference (IWCMC), 561-566, 24-28 Haziran.

[8] Afianian, A., Niksefat, S., Sadeghiyan, B., &Baptiste, D., (2018), Malware Dynamic Analysis Evasion Techniques: A Survey", arXivpreprint arXiv.

[9] Sartea, R., &Farinelli, A., (2018), "Detection of Intelligent Agent Behaviors Using Markov Chains", 17th International Conference on Autonomous Agents and Multi Agent Systems, (2064-2066).

[10] Choi, C., Esposito, C., Lee, M., & Choi, J., (2019), "Metamorphic malicious code behavior detection using probabilistic inference methods", Cognitive Systems Research, 56, 142-150.

[11] Anderson, B., Storlie, C., &Lane, T., (2012), "Improving malware classification: bridging the static/dynamic gap", 5th ACM workshop on Security and artificial intelligence.

[12] Nar, M., Kakisim, A. G., Yavuz, M. N., & Soğukpinar, İ., (2019), "Analysis and Comparison of Disassemblers for OpCode Based Malware Analysis", 2019 4th International Conference on Computer Science and Engineering (UBMK), 17-22, IEEE.

[13] Demetrio, L., Biggio, B., Lagorio, G., Roli, F., & Armando, A., (2019), "Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries", arXiv preprint arXiv:1901.03583.

[14] Biggio, B., Rieck, K., Ariu, D., Wressnegger, C., Corona, I., Giacinto, G., Roli, F., (2014), "Poisoning behavioral malware clustering", 2014 workshop on artificial intelligent and security workshop, 27-36, ACM.

[15] Wartell, R., Zhou, Y., Hamlen, K. W., Kantarcioglu, M., & Thuraisingham, B. (2011), "Differentiating code from data in x86 binaries", In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, (522-536), Springer, Berlin, Heidelberg.

[16] Hyde, R. (2003), "The art of assembly language", No Starch Press.

[17] Brookshear, J. G. (2008), "Computer science: an overview", Addison-Wesley Publishing Company.

[18] Marak, V. (2015), "Windows Malware Analysis Essentials", Packt Publishing Ltd.

[19] Sikorski, M., & Honig, A., (2012), "Practical malware analysis: the hands-on guide to dissecting malicious software", no starch press.

[20] Aho, A. V., Sethi, R., & Ullman, J. D., (1986)., "Compilers, principles, techniques", Addison wesley, 7(8), 9.

[21] Reynolds, J. C., (1972), "Definitional interpreters for higher-order programming languages", ACM annual conference-Volume 2, 717-740, 15-17 Ağustos.

[22] Eagle, C., (2011), "The IDA pro book", No Starch Press.

[23] Dandamudi, S. P., (2013), "Introduction to assembly language programming: from 8086 to Pentium processors", Springer Science & Business Media.

[24] Silberschatz, A., Gagne, G., & Galvin, P. B., (2018), "Operating system concepts", Wiley.

[25] Eilam, E., (2011), "Reversing: secrets of reverse engineering", John Wiley & Sons.

[26] Shanley, T., (2010), “x86 Instruction Set Architecture”, MindShare press.

[27] Kusswurm, D., (2014), “Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX. Apress.

[28] Irvine, K. R., (2011), “Assembly language for x86 processors”, Upper Saddle River, NJ: Prentice Hall.

[29] Carpenter, T., (2011), “Microsoft Windows Operating System Essentials”, John Wiley & Sons.

[30] Okolica, J., & Peterson, G. L., (2010), “Windows operating systems agnostic memory analysis. Digital investigation”, 7, S48-S56.

[31] Caelli, W., & Longley, D., (1989), “Information security for managers”, Springer.

[32] Malin, C. H., Casey, E., Aquilina, J. M., (2008), “Malware forensics: investigating and analyzing malicious code”, Syngress.

[33] Monnappa, K., (2018), “Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware”, Packt Publishing.

[34] Skoudis, E., & Zeltser, L., (2004), “Malware: Fighting malicious code”, Prentice Hall Professional.

[35] Bott, E., Siechert, C., & Stinson, C, (2009), “Windows 7 inside out”, Pearson Education.

[36] Matrosov, A., Rodionov, E., & Bratus, S., (2019), “Rootkits and bootkits: reversing modern malware and next generation threats”, No Starch Press.

[37] Carlin, D., Cowan, A., O’kane, P., & Sezer, S., (2017), “The effects of traditional anti-virus labels on malware detection using dynamic runtime opcodes”, IEEE Access, 5, 17742-17752.

[38] Xiao, F., Lin, Z., Sun, Y., & Ma, Y., (2019), “Malware Detection Based on Deep Learning of Behavior Graphs”, Mathematical Problems in Engineering, 2019.

[39] Aycock, J., deGraaf, R., & Jacobson, M., (2006), “Anti-disassembly using cryptographic hash functions”, Journal in Computer Virology, 2(1), 79-85.

[40] Jämthagen, C., Lantz, P., & Hell, M., (2013), “A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries”, In 2013 Workshop on Anti-malware Testing Research, (1-9), 10-13 Ekim.

[41] Ehteshamifar, S., Barresi, A., Gross, T. R., & Pradel, M., (2019), "Easy to Fool? Testing the Anti-evasion Capabilities of PDF Malware Scanners", arXiv preprint arXiv:1901.05674.

[42] Karl-Bridge-Microsoft. (12AD, May 12). Is Debugger Present function (debugapi.h) - Win32 apps. https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent, (Erişim Tarihi: 18/11/2019).

[43] Chen, X., Andersen, J., Mao, Z. M., Bailey, M., & Nazario, J., (2008), "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware", IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), (177-186), 17-19 Haziran.

[44] Ferrie, P., (2011), "The ultimate anti-debugging reference".

[45] Liston, T., (2006), "On the cutting edge: Thwarting virtual machine detection", http://handlers.sans. org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf.

[46] Tang, Y., Chen, S., (2007), "An automated signature-based approach against polymorphic internet worms", IEEE Transactions on Parallel and Distributed Systems, 18(7), 879-892.

[47] Xiao, Y., Cao, S., Cao, Z., Wang, F., Lin, F., Wu, J., Bi, H., (2016), "Matching Similar Functions in Different Versions of a Malware", In 2016 IEEE Trustcom/BigDataSE/ISPA, (252-259), 23-26 Ağustos .

[48] Alam, S., Traore, I., & Sogukpinar, I. (2014). "Current trends and the future of metamorphic malware detection", In Proceedings of the 7th International Conference on Security of Information and Networks, (411), 9-11 Eylül.

[49] Wang, L., Xu, D., Ming, J., Fu, Y., & Wu, D., (2019), "MetaHunt: Towards Taming Malware Mutation via Studying the Evolution of Metamorphic Virus", 3rd ACM Workshop on Software Protection, (15-26), 15 Kasım.

[50] Alam, S., Horspool, R. N., Traore, I., & Sogukpinar, I., (2015), "A framework for metamorphic malware analysis and real-time detection", computers & security, 48, 212-233.

[51] Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., & Elovici, Y., (2012), "Detecting unknown malicious code by applying classification techniques on opcode patterns", Security Informatics, 1(1), 1.

[52] Nguyen, M. H., Nguyen, T. B., Quan, T. T., Ogawa, M., (2013), "A hybrid approach for control flow graph construction from binary code", In 2013 20th Asia-Pacific Software Engineering Conference (APSEC), (Vol. 2, 159-164), 5 Aralık.

[53] Islam, R., Tian, R., Batten, L., & Versteeg, S., (2010), "Classification of malware based on string and function feature selection", In 2010 Second Cybercrime and Trustworthy Computing Workshop, (9-17), 19 Temmuz.

[54] Iwamoto, K., & Wasaki, K., (2012), "Malware classification based on extracted api sequences using static analysis", In Proceedings of the Asian Internet Engineering Conference, (31-38), 14-16 Kasım.

[55] Kakisim, A. G., Nar, M., Carkaci, N., Sogukpinar, I., (2018), "Analysis and Evaluation of Dynamic Feature-Based Malware Detection Methods", In International Conference on Security for Information Technology and Communications (247-258), Springer, Cham, 8-9 Kasım.

[56] Bonfante, G., Kaczmarek, M., Marion, J. Y., (2007), "Control flow graphs as malware signatures".

[57] Eskandari, M., & Hashemi, S., (2011), "Metamorphic malware detection using control flow graph mining", Int. J. Comput. Sci. Network Secur, 11(12), 1-6.

[58] Alam, S., Traore, I., & Sogukpinar, I., (2015), "Annotated control flow graph for metamorphic malware detection", The Computer Journal, 58(10), 2608-2621.

[59] Iwamoto, K., & Wasaki, K., (2012), "Malware classification based on extracted api sequences using static analysis", In Proceedings of the Asian Internet Engineeering Conference, (31-38). ACM, 14-16 Kasım.

[60] Kang, M. G., McCamant, S., Poosankam, P., & Song, D., (2011), "Dta++: dynamic taint analysis with targeted control-flow propagation", In NDSS, 6-9 Şubat.

[61] Carlin, D., O'Kane, P., Sezer, S., (2019), "A cost analysis of machine learning using dynamic runtime opcodes for malware detection", Computers & Security, 85, 138-155.

[62] Oktavianto, D., & Muhardianto, I., (2013), "Cuckoo malware analysis", Packt Publishing Ltd.

[63] Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G., Stringhini, G., (2019, "MaMaDroid: Detecting android malware by building markov chains of behavioral models (extended version)", ACM Transactions on Privacy and Security (TOPS), 22(2), 14.

[64] Karyotis, V., (2010), "Markov random fields for malware propagation: the case of chain networks", IEEE Communications Letters, 14(9), 875-877.

[65] Zhang, J., Qin, Z., Yin, H., Ou, L., Zhang, K., (2019). "A feature-hybrid malware variants detection using CNN based opcode embedding and BPNN based API embedding", Computers & Security, 84, 376-392.

[66] Goodfellow, I., Bengio, Y., Courville, A., (2016), "Deep learning", MIT press.

[67] Wang, C., Qin, Z., Zhang, J., Yin, H., (2016), "A malware variants detection methodology with an opcode based feature method and a fast density based clustering algorithm", In 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), (481-487), 13-15 Ağustos.

[68] Kolter, J. Z., Maloof, M. A., (2006), Learning to detect and classify malicious executables in the wild. Journal of Machine Learning Research, 7 (Dec), 2721-2744.

[69] Perdisci, R., Lanzi, A., & Lee, W., (2008), "McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables", In 2008 Annual Computer Security Applications Conference (ACSAC), (301-310), 8-12 Aralık.

[70] Azhikoden, A., & Vinod, P., (2015), "Meta opcode space for morphed malware detection", In 2015 11th International Conference on Innovations in Information Technology (IIT), (284-289), IEEE, 1-3 Kasım.

[71] Kang, B., Yerima, S. Y., McLaughlin, K., & Sezer, S., (2016), "N-opcode analysis for android malware classification and categorization", In 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security), (1-7), IEEE, 13-14 Haziran.

[72] Raman, K. (2012), "Selecting features to classify malware", InfoSec Southwest, 2012.

[73] Li, W. J., Wang, K., Stolfo, S. J., Herzog, B., (2005), "Fileprints: Identifying file types by n-gram analysis", In Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop (64-71), IEEE, 15-17 Haziran.

[74] George, N., Vinod, P., (2015), "Opcode position aware metamorphic malware detection: Signature vs histogram approach", In 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), (1011-1017), IEEE, 11 Mart.

[75] Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P. G., (2013), "Opcode sequences as representation of executables for data-mining-based unknown malware detection", Information Sciences, 231, 64-82.

[76] Rad, B. B., Masrom, M., Ibrahim, S., (2012), "Opcodes histogram for classifying metamorphic portable executables malware", In 2012 International Conference on e-Learning and e-Technologies in Education (ICEEE), (209-213), IEEE, 24-26 Eylül.

[77] Çarkacı, N., & Soğukpınar, İ., (2016), "Frequency based metamorphic malware detection", In 2016 24th Signal Processing and Communication Application Conference (SIU), (421-424), IEEE, 16-19 Mayıs.

[78] Yewale, A., Singh, M., (2016), "Malware detection based on opcode frequency", In 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), (646-649), IEEE, 8 Mayıs.

[79] Zolotukhin, M., Hämäläinen, T., (2014), "Detection of zero-day malware based on the analysis of opcode sequences", In 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC), (386-391), IEEE, 10-13 Ocak.

[80] Gong, T., Tan, X., & Zhu, M., (2009), "Malware detection via classifying with compression", In 2009 First International Conference on Information Science and Engineering, (1765-1768), IEEE.

[81] Yuxin, D., Wei, D., Yibin, Z., Chenglong, X., (2014), "Malicious code detection using opcode running tree representation", In 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (616-621), IEEE, 8 Kasım.

[82] Zhang, J., Qin, Z., Yin, H., Ou, L., Hu, Y., (2016), "IRMD: malware variant detection using opcode image recognition", In 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), (1175-1180), IEEE, 13-16 Aralık.

[83] Wang, T., & Xu, N., (2017), "Malware variants detection based on opcode image recognition in small training set", In 2017 IEEE 2nd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), (328-332), IEEE, 10 Mart.

[84] Sun, G., Qian, Q., (2018), "Deep learning and visualization for identifying malware families", IEEE Transactions on Dependable and Secure Computing.

[85] Alqurashi, S., Batarfi, O., (2016), "A comparison of malware detection techniques based on hidden markov model", Journal of Information Security, 7(03), 215.

[86] Annachhatre, C., Austin, T. H., Stamp, M., (2015), "Hidden Markov models for malware classification", Journal of Computer Virology and Hacking Techniques, 11(2), 59-73.

[87] Raghavan, A., Di Troia, F., Stamp, M., "Hidden Markov Models with Random Restarts vs Boosting for Malware Detection".

[88] Lin, D., Stamp, M., (2011), "Hunting for undetectable metamorphic viruses", Journal in computer virology, 7(3), 201-214.

[89] Tajoddin, A., Jalili, S., (2018), "HM3alD: Polymorphic Malware Detection Using Program Behavior-Aware Hidden Markov Model", Applied Sciences, 8(7), 1044.

[90] Divandari, H., Pechaz, B., Jahan, M. V., (2015), "Malware detection using Markov Blanket based on opcode sequences", In 2015 International Congress on Technology, Communication and Knowledge (ICTCK), (564-569), IEEE, 11 Kasım.

[91] Wong, W., Stamp, M., (2006), "Hunting for metamorphic engines", Journal in Computer Virology, 2(3), 211-229.

[92] Egele, M., Scholte, T., Kirda, E., Kruegel, C., (2012), "A survey on automated dynamic malware-analysis techniques and tools", ACM computing surveys (CSUR) 44(2), 6.

[93] Bayer, U., Kirda, E., Kruegel, C., (2010), "Improving the effciency of dynamic malware analysis", In: Proceedings of the 2010 ACM Symposium on Applied Computing, 1871-1878, ACM, 22 Mart.

[94] Bai J., An Z., Zou Z., Mu S., (2014), "A Dynamic Malware Detection Approach by Mining the Frequency of API Calls", Vols. 519-520, 309-312, Applied Mechanics and Materials.

[95] Hansen, S. S., Larsen, T. M. T., Stevanovic, M., Pedersen, J. M., (2016), "An approach for detection and family classification of malware based on behavioral analysis", In 2016 International Conference on Computing, Networking and Communications (ICNC), (1-5), IEEE, 15-18 Şubat.

[96] Shaid, S. Z. M., Maarof, M. A., (2014), "Malware behavior image for malware variant identification", In 2014 International Symposium on Biometrics and Security Technologies (ISBAST), 238-243, IEEE, 26 Ağustos.

[97] Pektaş, A., (2015), "Behavior based malware classification using online machine learning", Doctoral Tezi, Grenoble Alpes.

[98] Imran, M., Afzal, M. T., Qadir, M. A., (2015), "Using hidden markov model for dynamic malware analysis: First impressions", In 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), (816-821), IEEE, 15 Ağustos.

[99] Ki, Y., Kim, E., Kim, H. K., (2015), "A novel approach to detect malware based on API call sequence analysis", International Journal of Distributed Sensor Networks, 11(6), 659101.

[100] Choudhary, S. P., Vidyarthi, M. D., (2015), "A simple method for detection of metamorphic malware using dynamic analysis and text mining", Procedia Computer Science, 54, 265-270.

[101] Udayakumar, N., Anandaselvi, S., Subbulakshmi, T., (2017), "Dynamic malware analysis using machine learning algorithm", In 2017 International Conference on Intelligent Sustainable Systems (ICISS), (795-800), IEEE, 7 Aralık.

[102] Choi, Y. H., Han, B. J., Bae, B. C., Oh, H. G., & Sohn, K. W. (2012, August). Toward extracting malware features for classification using static and dynamic analysis. In 2012 8th International Conference on Computing and Networking Technology (INC, ICCIS and ICMIC) (pp. 126-129). IEEE.

[103] Gandotra, E., Bansal, D., Sofat, S.: Zero-day malware detection. In: Embedded Compu-ting and System Design (ISED), 2016 Sixth International Symposium on pp. 171-175. IEEE (2016).

[104] Firdausi, I., Erwin, A., & Nugroho, A. S. (2010, December). Analysis of machine learning techniques used in behavior-based malware detection. In 2010 second international conference on advances in computing, control, and telecommunication technologies (pp. 201-203). IEEE.

[105] Xiao, X., Wang, Z., Li, Q., Xia, S., & Jiang, Y. (2016). Back-propagation neural network on Markov chains from system call sequences: a new approach for detecting Android malware with system call sequences. IET Information Security, 11(1), 8-15.

[106] Kaur, R., & Singh, M. (2015). A Hybrid real-time zero-day attack detection and analysis system. IJ Computer Network and Information Security, 9, 19-31.

[107] Alsulami, B., Mancoridis, S., (2018), "Behavioral Malware Classification using Convolutional Recurrent Neural Networks", In 2018 13th International Conference on Malicious and Unwanted Software (MALWARE), (103-111), IEEE, 22-24 Ekim.

[108] Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., (2018), "Adversarial malware binaries: Evading deep learning for malware detection in executables", In 2018 26th European Signal Processing Conference (EUSIPCO), (533-537), IEEE, 3 Eylül.

[109] Le, Q., Boydell, O., Mac Namee, B., & Scanlon, M., (2018), "Deep learning at the shallow end: Malware classification for non-domain experts", Digital Investigation, 26, S118-S126.

[110] Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C., (2016), "Deep learning for classification of malware system call sequences", In Australasian Joint Conference on Artificial Intelligence, (137-149), Springer, Cham, 5-8 Aralık.

[111] Chen, Z., Ji, C., (2005), "Spatial-temporal modeling of malware propagation in networks", IEEE Transactions on Neural networks, 16(5), 1291-1303.

[112] Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., (2019), "Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries", arXiv preprint arXiv:1901.03583.

[113] Nar, M., Kakisim, A. G., Çarkaci, N., Yavuz, M. N., Sogukpinar, I., (2018), "Analysis and Comparison of Opcode-based Malware Detection Approaches",

In 2018 3rd International Conference on Computer Science and Engineering (UBMK), (498-503), IEEE, 20 Eylül.

[114] Kingma, D. P., Ba, J., (2014), "Adam: A method for stochastic optimization", arXiv preprint arXiv:1412.6980.

[115] VxHeaven, Computer virus collection, 2014, http://83.133.184.251/virensimulation.org/, (Erişim Tarihi: 15/04/2019).

[116] Chollet, F. Keras, (2015), GitHub repository. https://github.com/fchollet/keras, (Erişim Tarihi: 27/09/2019).

[117] Witten, I. H., Frank, E., Hall, M. A., Pal, C. J., (2016), "The WEKA workbench", In Online Appendix for Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann.

[118] Choi, C., Esposito, C., Lee, M., Choi, J., (2019), "Metamorphic malicious code behavior detection using probabilistic inference methods", Cognitive Systems Research, 56, 142-150.

ÖZGEÇMİŞ

Mert NAR 1991 yılında Ankara'da doğdu. 2009 yılında başladığı Atılım Üniversitesi Mühendislik Fakültesi Yazılım Mühendisliği Bölümünü 2014 yılında, çift ana dal programı ile 2012 yılında başladığı Fen-Edebiyat Fakültesi Matematik bölümünü 2015 yılında başarıyla bitirdi. Daha sonra 2 yıl özel sektörde yazılım Mühendisliği tecrübesinin ardından 2017 yılında yüksek lisans eğitimine Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalında başladı.