

**T.C.
PAMUKKALE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**

**JAYA OPTİMİZASYON ALGORİTMASI TABANLI
METAMORFİK KÖTÜCÜL KOD TESPİTİ**

YÜKSEK LİSANS TEZİ

KÜBRA NUR ATASEVER

DENİZLİ, OCAK - 2019

**T.C.
PAMUKKALE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**



**JAYA OPTİMİZASYON ALGORİTMASI TABANLI
METAMORFİK KÖTÜCÜL KOD TESPİTİ**

YÜKSEK LİSANS TEZİ

KÜBRA NUR ATASEVER

DENİZLİ, OCAK - 2019

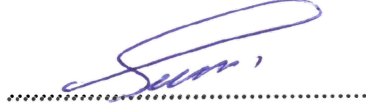
KABUL VE ONAY SAYFASI

KÜBRA NUR ATASEVER tarafından hazırlanan “**Jaya Optimizasyon Algoritması Tabanlı Metamorfik Kötücül Kod Tespiti**” adlı tez çalışmasının savunma sınavı **20.01.2019** tarihinde yapılmış olup aşağıda verilen jüri tarafından oy birliği / oy çokluğu ile Pamukkale Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı Yüksek Lisans Tezi olarak kabul edilmiştir.

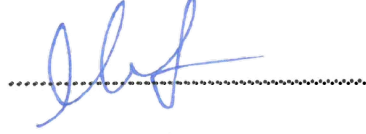
Jüri Üyeleri

İmza

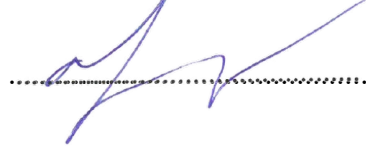
Danışman
Prof. Dr. Sezai TOKAT



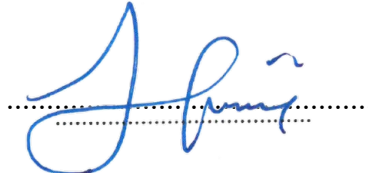
Üye
Dr.Öğr. Üyesi Mahmut SİNECEN
Adnan Menderes Üniversitesi



Üye
Dr. Öğr. Alper Uğur
Pamukkale Üniversitesi



Pamukkale Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun
..... tarih ve sayılı kararıyla onaylanmıştır.



Prof. Dr. Uğur YÜCEL

Fen Bilimleri Enstitüsü Müdürü

Bu tezin tasarımı, hazırlanması, yürütülmesi, arařtırmalarının yapılması ve bulgularının analizlerinde bilimsel etięe ve akademik kurallara özenle riayet edildiđini; bu alıřmanın dođrudan birincil ürünü olmayan bulguların, verilerin ve materyallerin bilimsel etięe uygun olarak kaynak gösterildiđini ve alıntı yapılan alıřmalara atfedildiđini beyan ederim.

KÜBRA NUR ATASEVER

ÖZET

**JAYA OPTİMİZASYON ALGORİTMASI TABANLI METAMORFİK
KÖTÜCÜL KOD TESPİTİ
YÜKSEK LİSANS TEZİ
KÜBRA NUR ATASEVER
PAMUKKALE ÜNİVERSİTESİ FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
(TEZ DANIŞMANI: PROF. DR. SEZAI TOKAT)**

DENİZLİ, OCAK - 2019

İnternet kullanımının yaygınlaşması ve bilginin daha kolay ulaşılabilir hale gelmesi ile birlikte zararlı kodlar ve bu kodları otomatik üreten araçlar her seviyeden virüs üreticisinin rahatlıkla ulaşabileceği hale gelmiştir. Günümüzde az bir bilgiyle, etkili ve güçlü kötücül kodlar kolaylıkla kullanılabilir ve üretilebilir durumdadır. Kötücül yazılımların, günümüzde ekonomiye, insan emeğine, zamana, bilgi güvenliğine zararı olduğu bilinmektedir. Bu nedenle, kötücül yazılımların tespit edilmesi bu zararların ortadan kalkmasına veya azalmasına yardımcı olmada oldukça önemlidir.

Metamorfik virüsler, son zamanların tespiti zor olan zararlı yazılımlarındandır. Bu durum önemli bir siber güvenlik problemi oluşturmaktadır. Metamorfik virüsleri tespit etmek için genellikle sezgisel tespit yöntemleri kullanılmaktadır.

Bu çalışmada metamorfik virüsleri tespit edebilmek için yöntem olarak, çalıştırılabilir dosyalarından elde edilen, assembly dosyalarından çıkarılan yerel ve harici fonksiyonları kullanılmıştır. Bu fonksiyonlardan oluşturulan grafların benzerlik oranı ve bu fonksiyonlara ait opcode dizilerinin benzerliği en uzun ortak küme (longest common subsequence) algoritması kullanılarak ölçülmüştür. Opcode benzerlik oranında en uygun çözümü bulmak için Jaya optimizasyon algoritması kullanılmıştır. Bir metamorfik virüsün iki farklı varyantı arasındaki benzerliği bu yöntem ile ölçülerek metamorfik virüs tespiti için yüzdeler şüphe oranı bulunmuştur.

Bu çalışmanın ve çalışma sonucu ortaya çıkan uygulamanın metamorfik kötücül yazılımlar alanında çalışma yapan bilim insanlarına ve uygulayıcılara faydalı olması ve daha farklı çalışmaların önünü açması beklenmektedir.

ANAHTAR KELİMELEER: Kötücül Kod, Kötücül Kod Tespiti, Metamorfik Kötücül Kod, Jaya Algoritması

ABSTRACT

METAMORPHIC MALWARE DETECTION BASED ON JAYA OPTIMIZATION ALGORITHM

MSC THESIS

KÜBRA NUR ATASEVER

PAMUKKALE UNIVERSITY INSTITUTE OF SCIENCE

COMPUTER ENGINEERING

(SUPERVISOR: PROF . DR. SEZAI TOKAT)

DENİZLİ, JANUARY 2019

With the widespread use of the Internet and making information more easily accessible, malicious codes and the tools that produce these codes automatically become easily accessible to virus producers of all levels. Today, with untutored information, effective and powerful malicious codes can be easily used and produced. Metamorphic viruses are malicious software that is difficult to detect recently, which is an important cyber security problem. Heuristic methods are often used to detect metamorphic viruses.

Malware is known to be harmful to today's economy, human labor, time, and information security. Therefore, it can be stated that the detection of malware is very important in helping to eliminate or reduce these damages.

In this study, local and external functions extracted from the assembly files obtained from executable files were used as a method for detecting metamorphic viruses. The similarity ratio of the graphs generated from these functions and the similarity of the opcode sequences of these functions were measured using the longest common subsequence algorithm. Jaya optimization algorithm was used to find the most appropriate solution in the opcode similarity ratio. The similarity between two different variants of a metamorphic virus was measured by this method.

It is expected that this study and the resultant application will be useful to researchers and practitioners working in the field of cyber security and malware, and paving the way for different studies.

KEYWORDS: Malware, Malware Detection, Metamorphic Malware Detection, Jaya Algorithm

İÇİNDEKİLER

İçindekiler

ÖZET	i
ABSTRACT	ii
İÇİNDEKİLER	iii
ŞEKİL LİSTESİ	iv
TABLO LİSTESİ	v
ÖNSÖZ	vi
1. GİRİŞ	1
2. KÖTÜCÜL KODLAR	3
2.1 Kötücül Kod Çeşitleri.....	4
2.1.1 Truva Atı.....	5
2.1.2 Cusus Yazılım.....	5
2.1.3 Solucanlar.....	6
2.1.4 Kök Kullanıcı Takımları.....	7
2.1.5 Virüsler.....	7
2.2 Kötücül Kod Tespit Yöntemleri.....	14
2.2.1 İmza Tabanlı Yöntemler.....	15
2.2.2 Davranış Tabanlı Yöntemler.....	16
2.2.3 Sezgisel Yöntemler.....	17
3. JAYA ALGORİTMASI TABANLI METAMORFİK KÖTÜCÜL KOD TESPİTİ	19
3.1 Jaya Algoritması.....	19
3.2 Fonksiyon Çağrı Grafları.....	22
3.3 Fonksiyon Çağrı Graflarının Oluşturulması.....	23
3.4 Fonksiyon Çağrı Grafları Benzerliği Ölçümü.....	26
3.4.1 Harici Fonksiyon Benzerliği.....	27
3.4.2 Aynı Harici Fonksiyonları Çağırarak Yerel Fonksiyonları Bulma.....	28
3.4.3 Komşu Düğümlere Göre Yerel Fonksiyon Benzerliği.....	28
3.4.4 Fonksiyon Çağrı Grafları Arasındaki Benzerlik Ölçümü.....	30
3.5 Jaya Algoritması ile Opcode Dizisi Benzerliği Ölçümü.....	30
3.5.1 Opcode Kategorilerinin Sınırlarının Belirlenmesi.....	30
3.5.2 İşaretlenmiş Fonksiyon Dizilerinin Karşılaştırılması.....	33
3.5.3 Jaya Algoritması ile Opcode Benzerlik Oranı Bulma.....	35
3.6 Opcode ve Vertex Benzerliği İçin Katsayı Belirleme.....	45
4. SONUÇLAR VE ÖNERİLER	46
5. KAYNAKLAR	47
6. EKLER	52
EK A: Opcode Dizisi Alt Bölütlemesi.....	52
EK B : 4 ile Bölütlenmiş Opcode Alt Bölütleme Örneği.....	54
EK C : Türetilmiş Kısa Kod Üzerinden Opcode Benzerliği Yönteminin İncelenmesi.....	56
ÖZGEÇMİŞ	59

ŞEKİL LİSTESİ

Sayfa

Şekil 2.1: Av-Test enstitüsü'ne göre 2009-2018 yılları arasında üretilmiş toplam zararlı yazılım sayısı grafiği (Web 1).....	4
Şekil 3.2: Jaya algoritması akış şeması (Pandey, 2016).....	21
Şekil 3.3: Mwor virüsü fonksiyon çağrı grafi parçası (Xu, 2010).	23
Şekil 3.4: Benzer köşelerin komşularının benzerliği (Deshpande, 2013).....	29
Şekil 3.5: Önerilen Benzerlik ölçüm şeması.....	36
Şekil 3.6: Opcode örnekleri için nihai sonuç grafiği	44

TABLO LİSTESİ

Sayfa

Tablo 2.1: Çöp kod ekleme örneği	11
Tablo 2.2: Değişken isimlerinin değiştirilmesi örneği	12
Tablo 2.3: Komutların sırasının değiştirilmesi örneği	13
Tablo 2.4: Fonksiyon sırasının değiştirilmesi örneği	13
Tablo 2.5: Komutların değiştirilmesi örneği (Kaushal, 2012).	14
Tablo 3.6: Jaya algoritması sözde kodu	22
Tablo 3.7: NGVCK virüsü örnek fonksiyonu ve fonksiyon grafi tablosu	24
Tablo 3.8: Fonksiyon çağrı graflarının oluşturulması algoritması (Deshpande, 2013).....	26
Tablo 3.9: Harici fonksiyon eşleştirme algoritması (Deshpande, 2013).....	27
Tablo 3.10: Harici fonksiyon tabanlı yerel fonksiyon benzerliğini bulma algoritması (Deshpande, 2013)	28
Tablo 3.11: Eşleşmiş düğümlerin komşu düğümleri için eşleşme (Ming, 2013).....	29
Tablo 3.12: Opcode sınırlarının belirlenmesi algoritması.....	31
Tablo 3.13: İşaretlenmiş fonksiyon dizisi oluşturma algoritması	32
Tablo 3.14: Bölütlenmemiş opcode dizisine göre işaretlenmiş fonksiyon örneği.....	32
Tablo 3.15: Dörtlü bölütlenmiş opcode dizisine göre işaretlenmiş fonksiyon örneği.....	32
Tablo 3.16: LCS algoritması sözde kodu	33
Tablo 3.17: Önerilen opcode dizisi benzerliği bulma algoritması	34
Tablo 3.18: Jaya algoritması ile opcode benzerliği ölçümü.....	38
Tablo 3.19: Mwor virüsü örneği	40
Tablo 3.20: Mwor virüsü çağrı komutu azaltılmış örneği.....	41
Tablo 3.21: Mwor virüsü metamorfoz geçiren fonksiyon sayısı artırılmış örneği.....	43
Tablo 3.22: Nihai sonuç ölçüm tablosu	45
Tablo A.23: Opcode Dizisi Alt Bölütlemesi	52
Tablo A.24: Opcode Dizisi Alt Bölütlemesi (Devamı).....	53
Tablo B.25: 4 ile Bölütlenmiş Opcode Alt Bölütleme Örneği.....	54
Tablo B.26: 4 ile Bölütlenmiş Opcode Alt Bölütleme Örneği (Devamı)	55
Tablo C.27: Orijinal ve Türetilmiş Kod Karşılaştırma Örneği	56
Tablo C.28: Türetilmiş kod için program çıktısı tablosu	58

ÖNSÖZ

Bilimin tanım ve özellikleri arasında sürekli gelişme halinde olma özelliği mevcuttur. Buradan hareketle, bilgisayar dünyasının en önemli problemlerinden olan kötücül kodların ekonomiye, zamana, bilim dünyasına, insan emeğine olan zararlarını önlemek veya azaltmak adına, bu tez çalışmasında, en tehlikeli virüslerden olan metamorfik kötü amaçlı yazılımların tespit edilmesi için yeni bir yöntem geliştirilmeye çalışılmıştır. Çalışmanın en önemli katkısı, metamorfik kötücül kodların tespiti için opcode ve graf benzerliği yöntemleri ile birlikte Jaya algoritmasının da kullanılmış olmasıdır. Bu çalışma Pamukkale Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı bünyesinde gerçekleştirilmiştir.

Çalışma dört bölümden oluşmaktadır. Birinci bölüm giriş bölümü olarak ayrılmıştır. İkinci bölümünde kötücül kod çeşitleri, truva atı, casus yazılım, solucanlar, kök kullanıcı takımları ve virüsler tanımlararak temel kavramlardan ve kötücül kodların tespit yöntemleri; imza tabanlı sistemler, davranış tabanlı sistemler ve genellikle metamorfik virüsler için kullanılan sezgisel yöntemlerden bahsedilmiştir. Üçüncü bölümde, Jaya algoritması tabanlı metamorfik virüs tespiti uygulaması analiz edilmiş ve açıklanmıştır. Son bölüm olan dördüncü bölümde sonuçlar ve öneriler yer almaktadır.

Bu çalışma boyunca bilgi ve tecrübeleriyle yol gösteren, desteklerini esirgemeyen danışmanım Prof. Dr. Sezai Tokat'a, tezimi defalarca okuyarak akış ve içerik olarak önemli katkılarda bulunan, bilgi güvenliği ve virüsler konusundaki değerli bilgi ve tecrübelerini benimle paylaşan Dr. Öğr. Üyesi Alper Uğur'a, yetişmemde katkısı olan tüm hocalarıma ve beni her zaman destekleyen aileme teşekkürü bir borç bilirim.

1. GİRİŞ

Hızlı iletişim altyapıları, akıllı telefonlar ve gelişen yazılım ve donanım teknolojileri ile İnternet yaygınlaşmakta ve bilgi kolay ulaşılabilir hale gelmektedir. Bu durum zararlı kodlara ve bu kodları otomatik üreten araçlara her seviyeden üreticinin rahatlıkla ulaşabilmesini de sağlamaktadır. Kötücül kod (malware), sahibinin bilgilendirilmiş rızası olmadan bir bilgisayar sistemine sızmak veya zarar vermek için tasarlanmış bir yazılımdır (Lee vd., 2010). Düşük seviyede yazılım bilgisi ile etkili ve güçlü kötücül kodlar kolaylıkla kullanılabilir ve üretilebilir hale gelmiştir. Bu etmenler kötücül kod sayısını artırmaktadır. (Çarkacı ve Soğukpınar, 2016).

Kötücül kodların gizlenmeye ve şifrenmeye başlaması 1970’li yıllardan sonra başlamıştır. Kötücül kod geliştiricileri, yazılımları tespit edildikçe yeni gizlenme yöntemleri denenmiş ve zaman içerisinde sırasıyla şifreli virüsler, oligomorfik, polimorfik ve son olarak metamorfik virüsler günümüzde bir çok sistemi etkilemeye başlamıştır (Rad ve diğ. 2012).

Oligomorfik virüsler, imza tabanlı virüs tespit yazılımlarından kaçınmak için sabit şifre çözücü yerine birden fazla şifre çözücü anahtar kümesi üreten fonksiyon bulundurarak her nesilde yapısında bulundurduğu yüzlerce farklı şifre çözücü anahtardan birisini rastgele olarak kullanan virüs çeşitleridir (Konstantinou, 2008).

Polimorfik virüsler, bulaştıkları sistemde, bir şifre ve milyonlarca farklı şifre çözücü kullanarak farklı formlarda gizlenen virüs çeşitleridir (Szor, 2005). Polimorfik virüsler her iterasyonda kendi kodlarının içinde bulunan şifre çözücü ile zararlı kod kısmının şifresini çözerek bulaştığı sisteme zarar verir ve daha sonra tekrar şifreleme algoritması kullanarak kodu şifreler. Gelişmiş örneklerinde milyonlarca şifre ve şifre çözücü yapısına rastlamak mümkündür. Kod yapısı her seferinde değiştiği ve imzası sabit kalmadığı için klasik imza tabanlı virüs tespit edici programlara yakalanmaları çok zordur (Nachenberg, 1997).

Metamorfik virüsler, oligomorfik ve polimorfik virüs nesillerinden farklı olarak, şifreli bir bölüm bulundurmazlar. Bu nedenle, şifre çözücü algoritmalara ihtiyaç duymazlar, fakat polimorfik virüs ile ortak yön olarak, bir mutasyon motoru

kullanırlar. Polimorfik virüslerden farklı olarak ise sadece şifre çözücü kısmı değiştirmek yerine, tüm gövdeyi değiştirirler (Rad ve diğ. 2012).

Geleneksel kötücül kod tespit yazılımları, kötücül yazılımların sabit ve bilindik imzalara sahip olmasını bekler ancak oligomorfik, polimorfik ve metamorfik virüsler sürekli olarak kod yapılarında değişim gösterip imzalarının sabit kalmamalarına neden oldukları için geleneksel kötücül kod karşıtı yazılımlar bu kötücül yazılımlara etki etmemektedir, metamorfik virüsler de bu sayede kolayca anti-virüs yazılımlarını atlatabilmekte ve siber güvenlik, bilgi güvenliği açısından büyük problemlere yol açmaktadır (Christodorescu ve S. Jha, 2004). Kötü amaçlı yazılımlar; sistem kaynaklarının aşırı tüketimi, hizmetlerin aksaması ve verilere yetkisiz erişim gibi bir çok kötü sonuca neden olabilir (Bayoğlu, Soğukpınar, 2012).

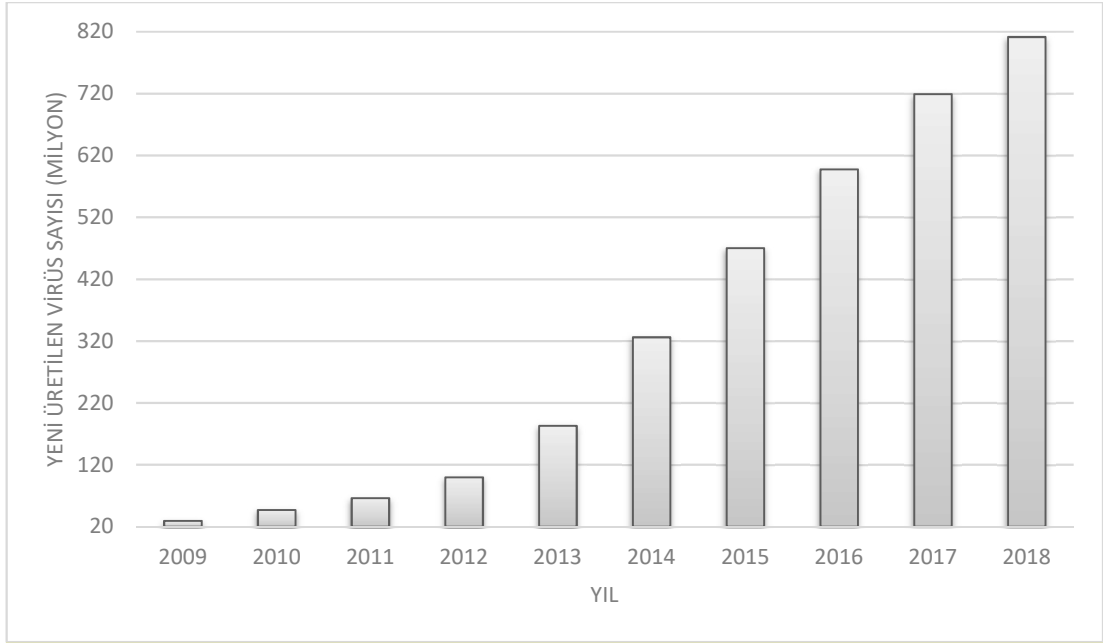
Bu tez çalışmasında en tehlikeli virüslerden olan metamorfik kötü amaçlı yazılımlar konusu üzerinde durulmuştur. Metamorfik kötü amaçlı yazılımlar, kontrol akışı aktarımı, fonksiyon ve değişken isimleri, yer değiştirme, çöp kod ekleme gibi çeşitli gizlenme teknikleri kullanarak geleneksel yöntemleri atlatabilir (Szor, 2005). Metamorfik virüsler her iterasyonda kodlarını değiştirirler, böylece statik imza tabanlı virüs tarayıcıları tarafından tespit edilmekten kurtulmaya çalışırlar ve istatistiksel olarak neredeyse tespit edilemeyen kötü amaçlı programlara yol açma potansiyeline sahiptirler (Walenstein ve Lakhota, 2006). Bu virüsler aynı zamanda daha derin statik analize meydan okumak için kod gizleme tekniklerini kullanırlar ve ayrıca kontrollü bir ortam altında yürütüldüklerini saptadıklarında davranışlarını değiştirerek, emülatörler gibi dinamik analizcileri de yenebilirler (Lakhota ve Kapoor, 2004).

2. KÖTÜCÜL KODLAR

Kötü amaçlı yazılımlar, kötücül kodlar, çoğunlukla zararlı ya da istenmeyen davranışlar gerçekleştirmek için tasarlanmış programlara verilen genel bir isimdir (Darrel, 2003). Kötücül kod (malware) çok çeşitli zararlı yazılımları kapsayan bir terimdir. Bu terime solucan, truva atı, virüsler gibi bir çok zararlı yazılım dâhil edilebilir (Zhang, 2007). Zaman içerisinde her zararlı yazılıma karşı bir tespit yöntemi geliştirildikçe yenileri ortaya çıkmıştır. Zararlı yazılımlar bu tespit yöntemlerini atlatmak için sürekli bir gelişim içindedirler.

İnternet kullanımının yaygınlaşması ve bilginin daha kolay ulaşılabilir hale gelmesi ile birlikte zararlı kodlar ve bu kodları otomatik üreten araçlar her seviyeden kötücül kod üreticisinin rahatlıkla ulaşabileceği hale gelmiştir. Günümüzde az bilgiyle etkili ve güçlü kötücül kodlar kolaylıkla kullanılabilir ve üretilebilir durumdadır. Bu durum kötücül kod sayısını artıran önemli faktörlerden biridir (Çarkacı ve Soğukpınar, 2016).

Zararlı yazılımları kötü amaçlar için kullanan kitlenin artması ve zararlı yazılımların siber saldırı gibi özel amaçlar için profesyonel gruplar tarafından kullanılması kötücül kodların çeşitliliğini, karmaşıklığını ve sayısını arttırdığı için bu kodların tespitinde ideal bir yöntem henüz mevcut değildir. Av-Test enstitüsünün Şekil 2.1’de verilen 2018 raporuna göre sadece 2018 yılında 800 milyon adet yeni kötücül kod ortaya çıkmıştır (web 1). Grafiğe göre 2009 yılından 2018 yılına kadar kötücül yazılımların düzenli bir şekilde arttığı görülebilir. Buna sebep olarak internet kullanımının artması düşünülebilir. Bu noktada kötücül kodlarla mücadele için anti-virüs yazılımlarının da aynı oranda artması gerekmektedir.



Şekil 2.1: Av-Test enstitüsü'ne göre 2009-2018 yılları arasında üretilmiş toplam zararlı yazılım sayısı grafiği (Web 1).

2.1 Kötücül Kod Çeşitleri

İlk olarak 1949 yılında kendi kendini çoğaltabilen programlarla hayatımıza giren virüs kavramı bugün milyonlarca farklı çeşit ve amaçta virüslere öncülük etmektedir. İlk ortaya çıktığında genellikle eğlence amacı taşıyan virüsler günümüzde çok daha farklı bir anlam kazanmıştır. Özellikle 2000'li yıllardan sonra internet bankacılığı, e-ticaret, sanal oyunlar vb. kullanım alanların yaygınlaşması kötücül yazılımların amaçlarını da değiştirmiş ve maddi gelir elde etme yöntemi olarak kullanımı yaygınlaşmıştır.

Anti-virüsler gibi kötü amaçlı yazılım saptama araçları, kişisel bilgisayarlarda kötü amaçlı yazılım saldırılarına karşı en büyük savunma olmuştur. Bu araçların virüs tespitine katkıları büyüktür ancak özellikle bilinmeyen kötü amaçlı yazılım örnekleriyle veya bilinen kötü amaçlı yazılımların varyantlarıyla baş etmede aşırı derecede basit olmaktan dolayı eleştirilmektedir (Gordon, 2004). Bu saptama araçları (dedektörler) imzaları kullanır ve sadece toplanan kötü amaçlı yazılım örneklerinden türetilen karakteristik komut dizilerinin tanımlanmasına odaklanır.

2.1.1 Truva Atı

Truva atı, iyi niyetli bazı amaçlara sahip gibi görünen bir programdır, ancak bazı gizli, kötü niyetli işlevleri gizler (Lenny, 2003). İlk bakışta, Truva atı normal veya kullanışlı bir program olarak görünecek, ancak cihaz üzerinde kurulduktan veya çalıştırıldıktan sonra aslında zararlı olacaktır. Truva atı kullanan kullanıcılar, Truva atının meşru bir yazılım veya meşru bir kaynaktan gelen lezyonlar gibi görüldüğü için açılmaya veya çalıştırmaya yöneltilir. Örneğin, bir saldırgan bir sistemdeki kötü amaçlı kod adını değiştirebilir, böylece Truva atı o makineye ait görünecektir. Truva atları kendini normal bir süreç olarak maskeleyebildiklerinden, bunları kullanıcılar tarafından tespit etmek zordur (Dezfouli, 2013).

Truva atı zararlı yazılımı, kendisini çalıştırabilir ve birçok istenmeyen etkilere veya ek zararlı yazılımların yüklenmesine yol açabilir. Truva atları en çok pazarlama için kullanılmaktadır. Günümüzün gelişmiş truva atları, web tarayıcısının tam kontrolünü ele geçirme ve bir bilgisayarın kayıt dosyasını değiştirme yeteneğine sahiptir (Pektaş, 2012).

Truva atı, kullanıcının izni olmadan ekran hareketlerini izleme, sistemi uzaktan kontrol etme, kameradan kayıt alma ve kişisel bilgilere erişme gibi bir çok zararlı uygulama olabilir. Ancak truva atları virüsler ve solucanlardan farklı olarak kendini çoğaltmaz ve yayılmazlar (Aycok, 2006).

2.1.2 Casus Yazılım

Bir casus yazılım, kullanıcı davranışını izleyen veya kullanıcı hakkında bilgi sahibi olmayan, kullanıcı hakkında bilgi toplayan herhangi bir yazılım olarak tanımlanabilir (Boldt, 2006). Ayrıca finansal kayba neden olabilen casus yazılımlar genellikle önceden yüklenmiş bir yazılımın parçası olarak görünür, ancak bazen bir ek veya köprü olarak e-posta yoluyla gelir (Davarcı, 2012). Casus yazılım kötücül yazılımları bir çok kötücül kod çeşidini de içinde barındırır, bu sebeple kötücül kod dünyasında diğer zararlı yazılımlara oranla daha çok ön plana çıkar (Canbek, Sağıroğlu, 2007).

Casus yazılım, kişisel bilgileri depolayabilir, tuş vuruşlarını yakalayabilir (ve saklayabilir) veya ekran görüntüsü alabilir. Genel olarak finansal kayba neden olan casus yazılım yazılımları, genellikle önceden yüklenmiş yazılımın bir parçası olarak görünür. Nadiren de bir e-posta yoluyla kurban sisteme ulaşır (Pektaş, 2012).

Casus yazılımlar internet kullanıcılarının güvenliği için giderek artan bir hızla daha büyük tehditlerden biri haline gelmektedir (Sarioiu, Gribble ve Levy., 2004). Virüsler ve solucanlar gibi diğer kötü amaçlı yazılım türlerinden farklı olarak, casus yazılım yazılımların hedefi genellikle hasara neden olmaz veya diğer sistemlere yayılmaz. Bunun yerine, casus yazılım programları, kullanıcıların davranışlarını izler ve tuş vuruşları ve tarama kalıpları gibi özel bilgileri çalar. Bu bilgi daha sonra casus yazılım dağıtıcılarına geri gönderilir ve hedefli reklam (ör. Pop-up reklamlar) veya pazarlama analizi için bir temel olarak kullanılır. Casus yazılım programları ayrıca bir kullanıcının tarayıcısını "kandırabilir" ve söz konusu olmayan kullanıcıyı casus yazılım yazılımının seçtiği web sitelerine yönlendirebilir. Son olarak, kullanıcıların gizliliğinin ihlaline ek olarak, casus yazılım programları, sistem performansının bozulmasından da sorumludur (Kirda, 2006).

2.1.3 Solucanlar

Solucanlar, herhangi bir kullanıcı müdahalesine ihtiyaç duymadan yayılabilen zararlı yazılımlardır (Lenny, 2003). Genellikle, virüs bulaşan belge bir makineden diğerine geçerken ağdaki diğer makineleri atlamak ve bunlara bulaşmak için ana makinenin ağını kullanır.

Bir solucan kötücül yazılımı, küçük bir Word belgesi boyutunda olabilir. Bu da yayılmasını ve gizlenmesini kolaylaştırır. Solucanlar, tüm ağı yalnızca bant genişliğini kullanarak ve herhangi bir kötü amaçlı etkinlik gerçekleştirilmeden daraltabilir. Geliştiriciler, solucanları ayrıca arka kapılar (Backdoors) oluşturmak için de kullanırlar.

Solucan, virüse oldukça benzer olsa da, kendisini mevcut bir programa kopyalamaya gerek duymaz. Bu nedenle, kendi kendini çoğaltma işlemi, yürütülebilir kod bağımlı olmadan bağımsız olarak devam edilebilir. Dahası, virüsler genellikle bir aygıttaki lezyonları bozarken veya değiştirirken, solucanlar ağa her zaman zarar verir.

Solucan, kullanıcı etkileşimi olmadan kendini kopyalayabildiği için, kendi başına milyonlarca kopya gönderebilir ve büyük yıkıcı etkilere neden olabilir (Davarcı, 2012).

2.1.4 Kök Kullanıcı Takımları

Kök kullanıcı takımları, kendilerini işletim sistemine çekirdek modülleri gibi kurarak sistemi bozan kötü amaçlı yazılımlardır. Bu tür yazılımlar sadece işletim sistemi çalışır durumda olduğu sürece tek seferlik zarar verici yapıda oldukları için sistem yeniden başlatıldığında zararlı özelliklerini kaybederler. Ancak, çoğu zaman sistemin açılıp kapanması günler, hatta aylar sürdüğü durumlarda çok uzun süreli zarar vermeleri de olasıdır (Bickford, 2010).

Bir kez bulaştığında, bir kök kullanıcı takımı gelecekteki birçok saldırı için basamak taşı olarak hizmet edebilir. Örneğin, parolalar ve kredi kartı numaraları gibi hassas kullanıcı verilerini ve tuş vuruşlarını sessizce kaydederek çalan keylogger'ları gizlemek için yaygın olarak kullanılır. Ayrıca sistemde arka kapı programlarını da kurabilirler; bu da uzaktan bir saldırganın gelecekte sisteme girmesine izin verir. Kök kullanıcı takımları, ayrıca, güvenlik duvarı / anti-virüs araçlarını devre dışı bırakmak veya sistemin sözde rasgele sayı üreticinin çıktı kalitesini etkilemek gibi diğer gizli etkinlikleri de gerçekleştirebilir, böylece zayıf şifreleme anahtarlarının üretilmesine neden olur (Baliga, Kamat ve Iftode, 2007).

Kök kullanıcı takımı faaliyetlerinin hiçbiri doğrudan kullanıcı tarafından görülemez çünkü kök kullanıcı takımı yazılımı kendi varlığını gizler uzun süre fark edilmeden kalmasını sağlar. Dolayısıyla enfekte sistemler üzerinde uzun süreli kontrolü sağlar (Bickford, 2010).

2.1.5 Virüsler

Virüs terimi ilk olarak 1984'te Fred Cohen tarafından hazırlanan 'Experiments with Computer Viruses' adlı tez çalışmasında kullanılmış ve çalıştırılabilir (exe file) bir dosyaya gizlenen, program çalıştırıldığında, kendi kopyalarını başka bir programa ekleyerek çoğaltan kötü amaçlı bir yazılımlar olarak tanımlanmıştır (Cohen, 1984).

Virüslerin kendi kendilerine hareket edebilmeleri, sürekli olarak yeni sistemlere bulaşmasına imkan tanır. Bu eylem kendi kendini çoğaltma olarak adlandırılır ve kötü amaçlı yazılımın ana özelliklerinden biridir. Özel bilgileri ve sabit disk alanını çalmak, verileri bozmak, kullanıcının girişini kaydetmek, ekranda politik mesajlar görüntülemek virüslerin gerçekleştirdiği zararlı işlemlerin örnekleridir. Virüsler genellikle bilinmeyen bir gönderici tarafından gönderilen veya bilinmeyen kaynaktan yüklenen bağlantıyı açmak gibi kullanıcı etkileşimli tetikleyiciler ile kendilerini çoğaltırlar (Aycock, 2006).

Bilgisayar virüsleri, en bilinen kötü amaçlı yazılım türüdür ve genellikle e-posta yoluyla iletilir, ancak bir flash sürücü gibi çeşitli depolama ortamları aracılığıyla da dağıtılabilir. Kurulduktan sonra, kendini çalıştırır, sistem dosyalarına bulaşır ve diğer sistemlere yayılmaya çalışır. Bir virüsün etkisi, yavaş sistem performansından, dosyaların kaybolmasından, sistemde istemsiz çalışan programlardan anlaşılabilir (Pektaş, 2012).

Virüsler 1970'li yıllara gelene kadar bilinen bir zararlı yazılım olsa da bu yıllarda geliştirilen şifreli virüsler, geleneksel virüs tespiti kavramına bakışı değiştirmiştir. Şifreli virüs tespitinin klasik yöntemlere göre çok daha fazla bilgi ve zaman gerektirmesi virüs üreticilerini bu alanda çalışmaya itmiştir. Şifrelenmiş virüs iki temel bölümden oluşur: şifre çözme algoritması ve ana gövde. Şifre çözücü algoritma, ana gövdenin kodunu şifrelemek ve şifresini çözmekle sorumlu olan kısa bir kod parçasıdır. Ana gövde, şifrelenmiş kötü amaçlı yazılımın gerçek kodudur ve şifre çözme döngüsü tarafından şifrelenmeden önce anlamlı değildir. Virüs bir sistem üzerinde çalışmaya başladığında, ilk olarak şifre çözücü algoritma çalıştırılır ve ana gövdeyi anlamlı veri haline dönüştürüp çalıştırılabilir makine kodu elde edilir (Rad ve diğ. 2012).

Şifrelenmiş virüslerin mantığı oldukça basittir. Örneğin XOR şifreleme konusunda oldukça pratiktir, çünkü şifrelenmiş koda tekrar XOR işlemi yapıldığında orijinal kodu verecektir. Böylece hem şifreleme hem de şifre çözme için aynı rutini kullanabilecektir. Sonuç olarak, şifre çözücü kısım her seferinde aynı kalacağı için şifre çözücünün modelini algılamak ve bunu kullanarak tespit etmek mümkündür. Virüs gövdesini çoğu zaman tanıyamasa bile şifre çözücü modülden şifrelenmiş virüsü takip etmek kolaydır (Wong, 2006)

Kötü amaçlı yazılım gizleme teknikleri zaman içerisinde gelişim göstererek şifreli virüslerden sonra oligomorfik virüslerin ortaya çıkmasını sağlamıştır. Oligomorfik virüs ayrıca semi-polimorfik olarak da adlandırılır (Aycock, 2006). Oligomorfik virüsler imza tabanlı virüs tespit yazılımlarından kaçınmak için sabit şifre çözücü yerine birden fazla şifre çözücü anahtar kümesi üreten fonksiyon bulundurarak her nesilde yapısında bulundurduğu yüzlerce farklı şifre çözücü anahtardan birisini rastgele olarak kullanır. “Whale virus” olarak adlandırılan virus ilk oligomorfik virüs olarak bilinir. Whale virus, bir düzine farklı şifre çözücü içinden bir tanesini rastgele olarak seçip kullanabilme özelliğine sahiptir (Szor, 2005). Örn. W95/memorial adlı oligomorfik kötücül yazılımı, 96 farklı şifre çözücüye sahiptir (Konstantinou, 2008).

Her ne kadar bu şifreleme kriptografik açıdan zayıf görülmesine de, ilkel anti-virüs programları sadece arama dizelerini kullanarak şifre çözücüye tespit edebilir. Bu yöntemle ilgili problem, sadece kendi şifre çözücüsüne dayanan bir virüsü saptayarak, algoritmanın varyantını tanımlayamamasıdır, çünkü farklı işlevselliğe sahip çeşitli virüsler aynı şifre çözücüye uygulayabilir. Bu yöntemi kullanan anti-virüs programları, bulaşmış lezyonu onaramaz ve aynı zamanda yanlış pozitifler de üretebilir (Szor, 2005).

Oligomorfik virüsler için tespit yöntemleri geliştikçe virüs geliştiricileri yeni yollar denemişlerdir. Bunun sonucunda 1990’da bilişim dünyası polimorfik virüsler ile tanışmıştır (Rad ve diğ. 2012). Polimorfik virüsler, oligomorfik virüslerin gelişmiş biçimi olarak düşünülebilir. Polimorfik virüsler de tıpkı oligomorfik virüsler gibi şifre ve şifre çözücüler kullanırlar ancak şifre çözücüye oluşturma yöntemleri farklıdır; polimorfik virüsler bir programa bulaştıklarında yeni bir şifre çözme rutini oluşturmak için mutasyon motorlarını kullanırlar. Yeni şifre çözme rutini, tam olarak aynı işlevselliğe sahip olacaktır, ancak talimatların sırası tamamen farklı olabilir. Mutasyon motoru aynı zamanda yeni bir dosyaya bulaşmadan önce virüsün statik kodunu şifrelemek için bir şifreleme rutini üretir. Ardından virüs, şifrelenmiş virüs gövdesiyle birlikte yeni şifre çözme rutinini hedeflenen dosya üzerine ekler. Virüs gövdesi şifrelenmiş ve şifre çözme rutini her bulaşma için farklı olduğundan, virüsten koruma tarayıcıları virüsleri arama dizelerini kullanarak algılayamaz. Mutasyon motorları, beraberindeki virüslerden genellikle çok daha karmaşık olan çok karmaşık programlardır. Daha sofistike mutasyon motorlarından bazıları, milyarlarca farklı şifre çözme rutini üretebilir (Nachenberg, 1997).

Kötücül kod gizleme tekniklerinde gelişim polimorfik virüslerden sonra metamorfik virüsler ile devam etmiştir. Metamorfik virüsler her iterasyonda kodlarını değiştirerek yayılan kötücül yazılımlar olarak tanımlanır (Chouchane, Lakhotia, 2006). Kodlarını sürekli olarak değiştirdikleri için statik, imza tabanlı virüs tarayıcıları ve anti-virüsler tarafından tespit edilmekten bu yöntemle kaçınırlar ve istatistiksel olarak neredeyse tespit edilemeyen kötü amaçlı programlara yol açma potansiyeline sahiptirler (Konstantinou, 2008). Bu virüsler aynı zamanda daha derin statik analize meydan okumak için kod gizleme tekniklerini kullanırlar ve ayrıca kontrollü bir ortam altında (sanal makine vb.) yürütüldüklerini saptadıklarında davranışlarını değiştirerek, emülatörler gibi dinamik analizcileri de yenebilirler (Zhihong, 2005).

Metamorfizmanın temel amacı, işlevselliğini korurken virüsün görünümünü değiştirmektir. Bunu başarmak için, metamorfik virüsler, parametre değişimi, kod permütasyonu, kod genişletme, kod küçültme ve çöp kod ekleme gibi birçok metamorfik dönüşüm kullanırlar (Konstantinou, 2008). Bilinen ilk metamorfik virüs win32.Apparition virüsüdür. Bu virüs basit şekilde kaynak kodunu içinde bulundurur ve çalıştığı makede bir derleyici bulduğunda, çöp kodunu (gereksiz ve işlevsiz kod) kaynak koduna ekleyip kaldırarak kendini yeniden derler. Bu, virüsün yeni nesillerinin öncekilerden tamamen farklı görünmesini sağlar (Szor, 2005).

Metamorfik virüsler, geleneksel imza tabanlı tespit yapan anti-virüsleri kolayca atlatabilir. Geleneksel imza tabanlı anti-virüsler daha önce karşılaşılmış olan virüslerin, her birinin kendine özgü olan imzalarını bir veri tabanında tutarak var olan virüsleri tespit etmek için bu veri tabanından faydalanırlar. Ancak metamorfik virüsler imzalarını çalışma anında sürekli olarak değiştirdikleri için geleneksel yöntemlerle yakalanmaları mümkün olmaz.

Metamorfik virüsler iki bölüme ayrılmış kod sistemi ile çalışırlar. İlk bölüm %20'lik kısmı oluşturan şifreleme bölümüdür. İkinci bölüm ise %80'i oluşturan kod dönüştürme ve üretme kısmıdır. Bu bölümler sayesinde metamorfik yapıdaki virüs, her iterasyonda kodun yapısını değiştirerek belirli bir imzaya sahip olmasını engeller ve imza tabanlı anti-virüslerden kaçabilir. İterasyonlar sırasında kodun metamorfoz geçiren kısmı yapısal olarak bir çok farklı duruma geçer ancak kodun yapısının değişmesi kodun işlevini ve davranışını değiştirmez. Geçirilen metamorfoz sadece tespit ediciyi aldatmaya yönelik işlemlerdir (Kaushal ve diğ. 2012). Metamorfik

virüsler için kodun yapısını değiştirmeye yönelik bir çok yöntem bulunur. Bunlardan bir kaçı aşağıda açıklanmıştır.

İlk örnek olarak işlevsiz döngü veya komut eklenmesi (junk code) verilebilir. Bu gizleme yöntemi, kod içerisinde imzanın değişmesine sebep olacak ancak davranışın aynı kalmasını sağlayacak kod eklenmesidir. Kısaca, çöp veya işlevsiz kod olarak adlandırılan kodlar, programın bir parçası olan, ancak mantıksal olmayan programlama yönergeleridir. Programın sonucunu etkilemez. “NOP”, ”MOV ax, ax”, ”SUB ax, 0” vb. gibi talimatlar virüsün farklı görünmesini ve imzasının değişmesini sağlar, bu nedenle sezgisel analizden kaçınılabılır (Kaushal, 2012).

Tablo 2.1’de gösterildiği üzere, kod parçacıkları, kodun bütününde rastgele şekilde yerleştirilen ve işlevsiz ölü kodlardır. Sıklıkla ölü kod olarak NOP kullanılır. Değişkenleri sıfır ile toplama (SUB eax, 0; ADD eax, 0), 1 ile çarpma gibi örnekler verilebilir. Kodun işlevinde herhangi bir değişiklik olmazken kodun akışında ve imzasında değişikliğe yol açar.

Tablo 2.1: Çöp kod ekleme örneği

Orijinal kod	Metamorfoz sonrası kod
PUSH ecx	PUSH ecx NOP
PUSH eax	PUSH eax
POP ebx	POP ebx
PUSH eax	PUSH eax
POP ebx	SUB eax, 0
MOV ebp, [ebx]	NOP
POP ebx	POP ebx nop MOV ebp, [ebx] POP ebx

Diğer bir gizleme yöntemi, değişken isimlerinin değiştirilmesidir. Bu yöntem, komutların eş değeri ile değiştirilmesi, kodun boyutunu artıran, imzasını değiştiren ama davranışını değiştirmeyen bir diğer yöntemdir (Zhang, 2007). Bu teknik en çok kullanılan ve en basit metamorfoz yöntemlerinden biridir. Bu yöntem, virüs yazarı Vecna tarafından yaratılan ve ilk olarak 1998’de piyasaya sürülen Win95 / Regswap

virüsü tarafından kullanılmaya başlandı. Virüsün farklı nesilleri işlev olarak aynı kodu kullanır fakat farklı komut isimleri verir (Konstantinou, 2008).

Tablo 2.2 örneğinde gösterildiği üzere, kod içinde kullanılan değişken isimleri (edi, esi, eax, ebx vs.) rastgele olarak kendi içlerinde yer değiştirerek kullanılmıştır. Kodun yapısındaki bu değişiklik imzanın da değişmesine yol açar.

Tablo 2.2: Değişken isimlerinin değiştirilmesi örneği

Orijinal kod	Metamorfoz sonrası kod
POP edx	POP eax
MOV edi, 0004h	MOV ebx ,0004h
MOV esi, ebp	MOV edx , ebp
MOV eax, 000Ch	MOV edi , 000Ch
ADD edx, 0088h	ADD eax , 0088h
MOV ebx, [edx]	MOV esi , [eax]
MOV [esi+eax*4], ebx	MOV [edx+edi*4], esi

Bir diğer örnek, komutların sırasının değiştirilmesidir. Bu yöntemde, kod bloklarının yerleri değiştirilir ve böylece kodun yapısı ve akışı değiştirilmiş olur. En etkili metamorfik değişim yöntemlerindedir. Değişim belli bir kod bloğunda ya da belli komutlarda yapılabilir (Çarkacı, Soğukpınar, 2016).

Sırası değiştirildiğinde kodun işlevinin bozulmadığı durumlarda metamorfoz için, değiştirilebilecek kod parçacıkları rastgele olarak yer değiştirir. Kodun boyutunda veya işlevinde farklılık meydana getirmeyen bu yöntem imzasının değişmesine ve yakalanma riskinin azalmasına neden olur (Szor, 2011).

Tablo 2.3 örneğinde gösterildiği üzere, “MOV esi, ebp” ve “MOV edi, 0004h” komutları işlev olarak öncelik sırasına tabi olmadığı için sırasını değiştirerek kodu metamorfoza uğratmak mümkündür ancak bu yöntem komut sırasının önemli olduğu noktalarda işlevsiz kalmaktadır. Bu sebeple büyük çaplı değişimlerde kullanılamaz.

Tablo 2.3: Komutların sırasının değiştirilmesi örneği

Orijinal kod	Metamorfoz sonrası kod
POP edx	POP edx
MOV edi,0004h	MOV esi, ebp
MOV esi,ebp	MOV edi, 0004h
MOV eax,000Ch	MOV eax,000Ch
ADD edx,0088h	ADD edx,0088h
MOV ebx,[edx]	MOV ebx,[edx]
MOV [esi+eax*4],ebx	MOV [esi+eax*4],ebx

Bir diğer yöntem, fonksiyon sırasının değiştirilmesidir. Bu yöntem, sırası birbirini etkilemeyen fonksiyonların yerlerinin değiştirilmesi olarak bilinir. Programın işlevselliğini ve kodun yürütülmesini etkilemez, modüllerin sırası değiştiği için sadece kod yapısı değişmiş olur (Kaushal, 2012). Mantık olarak komut sırasının değiştirilmesi ile aynı amacı güder.

Tablo 2.4 örneğinde gösterildiği üzere, “FUNCTION1”, “FUNCTION2” ve “FUNCTION3” fonksiyonlarının rastgele olarak sırası değiştirilir. Bu işlem sırasında fonksiyonlar öncelik sırasında işlev olarak birbirini etkilemediği için kodun yapısında değişiklik olur ancak işlevinde bir değişiklik meydana gelmez.

Tablo 2.4: Fonksiyon sırasının değiştirilmesi örneği

Orijinal kod	Metamorfoz sonrası kod
FUNCTION1:	FUNCTION3:
ADD eax, ebx	MOV eax, [x]
MOV [x], eax	FUNCTION1:
FUNCTION2:	ADD eax, ebx
MOV ebx, [y]	MOV [x], eax
FUNCTION3:	FUNCTION2:
MOV eax, [x]	MOV ebx, [y]

Bir diğer yöntem, komutların değiştirilmesidir. Bu yöntem, metamorfik virüslerin, komutlarının bir kısmını eşdeğer komutlarla değiştirmesidir. Örneğin, virüs

“XOR eax, eax” komutunu “SUB eax, eax” talimatıyla deęiřtirebilir. Her iki talimat da aynı iřlevi yerine getirir (eax kaydının ierięini sıfırlama) ancak ikisi de farklı bir opcode sahiptir ve virüs imzasını deęiřtirir. Bu teknik iin ayrıca komut eřdeęer listesini ieren bir veri tabanına ihtiya vardır bu da yontemi hantallařtıran bir etkidir.

Tablo 2.5’te Kaushal tarafından verilen rnekte kuyruk yapısında alıřan bir kod iin “ebp” deęiřkeni ncelikte kuyruęa push ediliyor daha sonra “MOV ebp, esp” komutu ile ebp deęiřkenine esp kopyalanıyor. Bu kod paracacıęı komutların deęiřtirilmesi yontemi ile metamorfoz geirdięinde, “MOV ebp, esp” komutunun yerini “PUSH esp” ve “PUSH ebp” alır.

Tablo 2.5: Komutların deęiřtirilmesi rneęi (Kaushal, 2012).

Orijinal kod	Metamorfoz sonrası kod
PUSH ebp	PUSH ebp
MOV ebp, esp	PUSH esp
MOV esi, ptr [ebp + 08]	POP ebp
TEST esi, esi	MOV esi, ptr [ebp + 08]
MOV edi, ptr [ebp + 0c]	OR esi, esi
OR edi, edi	MOV edi, ptr [ebp + 0c]
XOR edx, edx	TEST edi, edi
	SUB edx, edx

2.2 Ktcl Kod Tespit Yontemleri

Ktcl kodlardan kaynaklanan saldırılarla mcadele etmek iin, genellikle kt amalı yazılım yapısının kayda deęer bir řekilde deęiřmedięi varsayımına dayanan yazılımlar geliřtirilmiřtir. Fakat ikinci nesil ktcl kodlar her iterasyonda kendini deęiřtirdięi veya řifreledięi iin birbirinden ok farklıdır, dolayısıyla bu tr yazılımlardan gelen saldırılar her geen gn artmaktadır. Bu nedenle, hem akademik alıřmalar hem de anti-vir geliřtiricileri ktcl kodların evriminden kaynaklanan zararı nlemek iin srekli olarak vir eřitlerine dayalı olarak kendilerini yenilemesi gerekmektedir. Bu blmde ktcl kodların tespitinde kullanılan eřitli teknikler tartiřılmaktadır.

Kötücül kod tespit yöntemleri için üç temel kategori vardır. Bunlar imza tabanlı, davranış tabanlı ve sezgisel tabanlı yaklaşımlardır. Farklı tipte yazılımlar için farklı çözüm yöntemleri sunulmaktadır. Basit ve ilkel yapıları virüsler imza tabanlı yaklaşımlar ile daha hızlı yakalanırken; metamorfik ve polimorfik gibi imzası sabit olmayan, sürekli değişim içerisinde olan virüsler ise ancak davranış tabanlı ve sezgisel tabanlı yaklaşımlarla tespit edilebilmektedirler.

2.2.1 İmza Tabanlı Yöntemler

İmza tespiti, bilinen kötücül kodları tespit etmenin en basit ve etkili yollarından biridir (Griffin ve diğ. 2009). Geleneksel imza tabanlı yöntemler ilk olarak virüslerin ortaya çıktığı 1980'li yıllardan beri kullanılan yöntemlerdir. Tespit için öncelikle kötü amaçlı yazılım tanımlandıktan sonra, benzersiz bayt dizileri çıkarılır. İmza, her dosya için, bir exe dosyasının parmak izi gibi benzersiz bir özelliktir (Gutmann, 2007). İmza temelli yöntemler, bunları tanımlamak için çeşitli kötücül kodlardan çıkarılan modelleri kullanır. Bu diziler kötü amaçlı yazılımın imzasını temsil eder. Bu imzalar, belirli bir kötü amaçlı yazılımı, diğer iyi huylu bir programla ilgili olarak karakterize etmek için yeterince uzun seçilmektedir.

Bu teknik, sistemde bulunan ve tanımlanmış kötü amaçlı yazılım imzasını bulmak için virüs imzaları veritabanını tarar, eğer kötü amaçlı yazılım bulunduğu dair bir uyarı bulunursa, kötücül kod olarak işaretler. Ancak virüs imzasının birebir eşleşmediği durumlarda taramadan kaçması olasıdır. Yani küçük bir farklılık bile imza tabanlı tarayıcılardan kaçmasına yetecektir (Tran, 2013).

Her gün binlerce yeni virüs üretildiği düşünüldüğünde, tüm bu virüslerin imzalarını veri tabanında tutmak, etkili bir süre içerisinde tarama yaparak virüsü tespit etmek gittikçe zorlaşmakta ve maliyetli bir hale gelmektedir. Virüs imzalarının da virüslerin gösterdiği gelişime paralel olarak daha büyük boyutlarda olması anti-malware üreticilerini oldukça zorlamaktadır.

İmza tabanlı yaklaşımlar, sürekli kendilerini güncellemek zorunda oldukları için, aktif bir veri tabanına ihtiyaç duyar ve yeni üretilen virüslere karşı duyarsızdır (Sharma, Sahay, 2014). Bu yöntemler, bilinmeyen kötü amaçlı yazılım varyantlarını algılayamaz ve benzersiz virüs imzalarını bir veritabanında toplamak yüksek miktarda

insan gücü, zaman ve para gerektirir. Bunlar, bu yöntemlerin temel olumsuzluklarıdır. Ayrıca, polimorfik ve metamorfik gibi her iterasyonda kendi kodlarını değiştiren kötücül kodlara karşı koyamamak, başka bir dezavantajdır. Bu zorlukların üstesinden gelmek için araştırma kurumları davranış tabanlı veya sezgisel yöntemler gibi tamamen yeni bir kötü amaçlı yazılım tespit ailesi önermektedir (Bazrafshan, 2013).

2.2.2 Davranış Tabanlı Yöntemler

Davranış tabanlı yöntemler, kötü amaçlı yazılım tespiti sırasında zararlı yazılımın davranışlarını inceleyen yöntemlerdir. Sistem üzerinde çalışan her program çalışma zamanında (runtime) sisteme belli istekler gönderir. Bu isteklerin takip edilmesi ile zararlı yazılım tespiti yapmak mümkün olur.

Davranış temelli yöntemlerde, hem zararsız yazılım dosyalarının hem de kötü amaçlı yazılım dosyalarının davranışı, genellikle eğitim veya öğrenim aşaması olarak adlandırılan ilk aşamada incelenir ve daha sonra izleme aşamasında, belirli bir yürütülebilir dosyayı sınıflandırmak için eğitim aşamasında toplanan bilgiler kullanılır (Jacob ve diğ. 2008).

Davranış tabanlı yöntemler, zararlı yazılımların yürütülebilir (executable) dosyalarının yapmış oldukları davranışları izleyerek, daha önceden var olan bir dizi zararlı davranış ile karşılaştırır ve benzerlikleri saptar. Kötü amaçlı yazılımlar da diğer tüm programlar gibi işlemciye, belleğe, programlara ve diğer işletim kaynaklarına girdi göndermek adına ana bilgisayar sistemini kullanır (Jacob, Debar, Filiol, 2008). Bu sebeple, imzaları bilinmese dahi her bir grup malware için belli davranış kalıpları içinden zararlı yazılımları tespit etmek mümkündür. Davranış tabanlı yöntemler şifrelenmiş, gizlenmiş, bilinmeyen zararlı yazılımların tespitinde etkilidir. Davranış tabanlı bir algılama yaklaşımının bir örneği, Symantec tarafından patentli olan histogram tabanlı kötü amaçlı kod algılama teknolojisidir (web 2). Symantec tarafından üretilen bu uygulamanın olumlu yönü polimorfik virüslerde etkili olmasıdır. Olumsuz yönü ise tarama zamanı yüksekliğinin fazlalığıdır.

Zero-day atakları yazılımdaki herhangi bir zayıflığın üreticiler tarafından fark edilmesine rağmen önlem alınmadan saldırıya uğramasıdır. Davranış tabanlı

yöntemler, anormalliğe dayalı bir karşılaştırma ile çalıştıkları için zero-day saldırıları için etkili bir çözümdür.

Bir davranış tabanlı yöntem kısaca şu bölümlerden oluşur (Bazrafshan, 2013):

Veri toplayıcı (Data collector): bu bileşen analizi yapılmak istenen exe dosyası hakkında statik ve dinamik bilgi toplar.

Yorumlayıcı (Interpreter): bu bileşen data collector tarafından toplanan bilgiyi ara temsil (intermediate represented) haline dönüştürür.

Eşleştirici (Matcher): davranış sinyalleri ile yorumlayıcı modülden elde edilen verileri karşılaştırır ve karar verir.

Davranış tabanlı yöntemlerin iki temel sınırlaması, yüksek yanlış alarm ve eğitim aşamasında hangi özelliklerin öğrenilmesi gerektiğini belirleyen karmaşıklığıdır. Bu, anomali tabanlı tespit teknikleriyle ilişkili yüksek yanlış pozitif orana katkıda bulunur. Bu problemi ele almak için, denetime tabi tutulan bir programın hatalı davranıp davranmadığına karar vermek için, normal davranışın ne olduğu ile ilgili bazı belirtileri veya kural kümelerini kaldırarak, belirtme dayalı algılama sistemi yaklaşık olarak tahmin etmek için ortaya çıkar. Bununla birlikte, incelenen bir sistemin veya programın tam davranışını belirtmek zordur; bu, normal programın yanlış bir şekilde kötü amaçlı yazılım olarak kabul edildiği pozitif (yanlış pozitif) ile ilgili bir problem olarak ortaya çıkar (Ling, 2017).

2.2.3 Sezgisel Yöntemler

Metamorfik kötücül kod tespitine yönelik en büyük sorun, kötü amaçlı yazılımın yayıldığı zaman mutasyon geçirme yeteneğidir. Bu sorunun üstesinden gelmek için sezgisel yöntemler kullanılır (Nair, Vinod P. ve diğ., 2010).

Sezgisel yöntemler tüm sanal ortamın sisteme kurulmasını gerektirdiği için yanlış pozitif oranı yüksektir. Araştırmacılar tespit tekniklerinin yanlış alarmı azaltmak için başka bir tespit tekniği ile birleştirir (Govindaraju, 2010).

Sezgisel temelli yaklaşımlar, imza tabanlı ve davranış tabanlı tespit yöntemlerinin eksikliklerinin üstesinden gelme amaçlı geliştirilmiştir. Geleneksel

tespit yöntemleri yeni nesil bilgisayar virüsleri için etkili bir yakalama sunamamaktadır. Özellikle polimorfik ve metamorfik virüsler, çalışma zamanında yeni şifrelemeler yaparak veya metamorfoz geçirerek imzalarının sabit olmasını engeller ve imza tabanlı anti-virüslerin ve davranış tabanlı anti-virüslerin büyük bir çoğunluğuna yakalanmazlar.

Sezgisel yöntemler virüs tespiti için veri madenciliği, makine öğrenmesi gibi yöntemleri kullanırlar. Bu sebeple geleneksel yöntemlerle tespit edilemeyen polimorfik ve metamorfik virüslerin tespitinde oldukça etkilidir.

Bu çalışmada metamorfik kötücül kodlar, Jaya algoritması kullanılarak sezgisel yöntemler ile tespit edilmeye çalışılmıştır.

3. JAYA ALGORİTMASI TABANLI METAMORFİK KÖTÜCÜL KOD TESPİTİ

Bu bölümde exe dosyasından çıkarılan assembly kodu üzerinden fonksiyon çağrı grafiklerinin oluşturulması, grafiklerin benzerliğinin ölçülmesi, opcode benzerliğinin ölçülmesi ve Jaya optimizasyon algoritması ile en uygun benzerlik ölçüm yüzdesinin bulunarak metamorfik kötücül kod tespitinin yapılması anlatılmıştır.

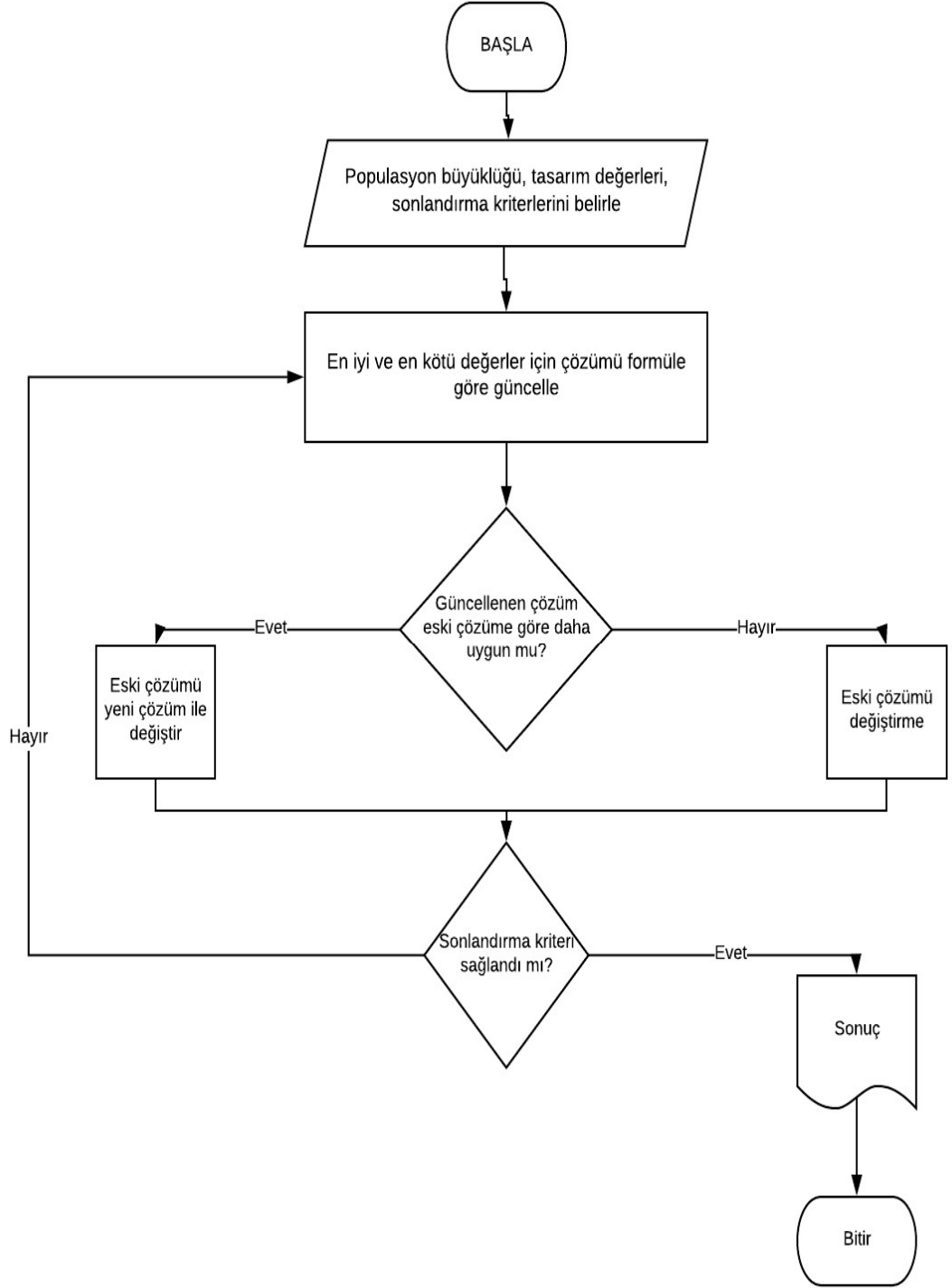
3.1 Jaya Algoritması

Tüm evrimsel ve sürü tabanlı algoritmalar, olasılık tabanlı algoritmalar ve popülasyon boyutu, nesil sayısı, elit strateji boyutu gibi ortak kontrol parametrelerini gerektirir. Ortak kontrol parametrelerinin yanı sıra her algoritma kendi yapısına özgü kontrol parametrelerini gerektirir. Örneğin, genetik algoritmalar, mutasyon olasılığı, çapraz olasılık, seçim operatörü; parçacık sürü optimizasyonu ağırlık, sosyal ve bilişsel parametreler; yapay arı kolonisi ise gözlemci arı, işçi arı parametrelerini kullanır.

Benzer şekilde, ES, EP, DE, SFL, ACO, FF, CSO, AIA, GSA, BBO, FPA, ALO, IWO gibi diğer algoritmalar ilgili algoritmaya özgü parametrelerin ayarlanmasına ihtiyaç duyar. Algoritmaya özgü parametrelerin uygun şekilde ayarlanması yukarıda belirtilen algoritmaların performansını etkileyen çok önemli bir faktördür. Algoritmaya özgü parametrelerin yanlış ayarlanması ya hesaplama çabasını arttırır ya da yerel en uygun çözümü verir. Bu gerçeği göz önüne alarak, herhangi bir algoritmaya özgü parametre gerektirmeyen öğretilme-öğrenme tabanlı optimizasyon (TLBO) algoritmasını tanıttı (Rao, 2011).

TLBO algoritması, sadece popülasyon büyüklüğü ve güncel jenerasyon sayısı gibi ortak kontrol parametlerine ihtiyaç duyar ve optimizasyon araştırmacıları arasında geniş bir kabul görmüştür. TLBO algoritmasının başarısı ile birlikte, 2016 yılında Rao tarafından başka bir özel parametresiz algoritma önerilmiştir. Ancak, TLBO algoritmasının iki aşamasından farklı olarak (öğretmen aşaması ve öğrenen aşaması), önerilen algoritma yalnızca bir safhadır ve uygulanması nispeten daha basittir. Önerilen algoritmanın çalışması TLBO algoritmasından çok farklıdır (Pandey, 2016).

Jaya, basit ve güçlü bir küresel optimizasyon algoritmasıdır. Sınırlı ve sınırsız problemlerin çözümüne başarıyla uygulanmıştır. Bununla birlikte, TLBO gibi parametresiz algoritma olsa da, öğrenen fazı gerektirmediği için farklıdır, yani TLBO eylemini iki aşamada gerçekleştirirken yalnızca bir faz öğretme fazı kullanmaktadır. Belli bir problemin çözümü en iyi çözüme doğru ilerletilebileceği, en kötü çözümü önleyebileceği gerçeğine dayanmaktadır. Bu algoritmanın olumlu yanı, sadece popülasyon boyutu gibi az sayıda kontrol parametresine ihtiyaç duymasındadır. Algoritmaya özgü parametrelerin kontrolü, görüldüğü kadar kolay değildir. Ayrıca, her yinelemede parametrelerin denetlenmesi çoğu zaman zor ve zaman alıcıdır. Bütün bu süreç Jaya algoritmasında yoktur. Böylece algoritmayı ciddi derecede hafifletir. Şekil 3.2, Jaya algoritmasının akış diyagramını göstermektedir.



Şekil 3.2: Jaya algoritması akış şeması (Pandey, 2016)

Tablo 3.6’da gösterildiği gibi, Jaya algoritması, sadece popülasyon büyüklüğü, tasarım değişkenleri (design variables), maksimum nesil (generation) sayısı parametreleri alınarak başlar. Daha sonra en iyi ve en kötü çözümü temel olarak S4 adımıdaki formül uygulanır, yeni çözüm ile eski çözüm karşılaştırılır. Daha uygun bir çözüm bulundu ise algoritma sonlandırılır ve en uygun çözümü sonuç olarak verir; daha uygun çözüm bulamadığı durumda S3 ve S4 adımları sonuca ulaşana dek tekrarlanır.

Tablo 3.6: Jaya algoritması sözde kodu

<p>S1: Initialize</p> <p> PS Population_size</p> <p> NDV Number_of_Design_Variables</p> <p> TER_COD Termination Condition</p> <p>S2: Until the termination condition, not satisfied Repeat S3 to S5</p> <p>S3: Evaluate the best and worst solution</p> <p> Set best Best_Solution_Population</p> <p> Set worst Worst_Solution_Population</p> <p>S4: Modify the solution</p> $X_{J,K,i}' = X_{J,K,i} + r_{2,i,j}(X_{j,best,i} - X_{j,best,i}) - r_{1,i,j}(X_{j,worst,i} - X_{j,worst,i})$ <p>S5: If ($X_{J,K,i}' > X_{J,K,i}$) Then</p> <p> Update the previous solution</p> <p>Else</p> <p> No update in the previous solution</p> <p>S6: Display the optimum result</p>
--

3.2 Fonksiyon Çağrı Grafları

Bir fonksiyon çağrı grafiği $G = (V; E)$ şeklinde ifade edilir ve köşeler V, kenarlar E'den oluşur (Ming, 2013). Bir executable dosya içerisinde fonksiyonları iki başlıkta harici fonksiyonlar ve yerel fonksiyonlar olarak tanımlamak mümkündür. Yerel fonksiyonlar, her programın kendine özgü olarak işlem yaptığı fonksiyonlardır. Harici fonksiyonlar ise sistem ve kütüphane fonksiyonlarıdır ve her zaman isimleri sabit kalır. Yerel fonksiyonlar “sub_xxxx proc near” anahtar kelimeleri ile başlar ve

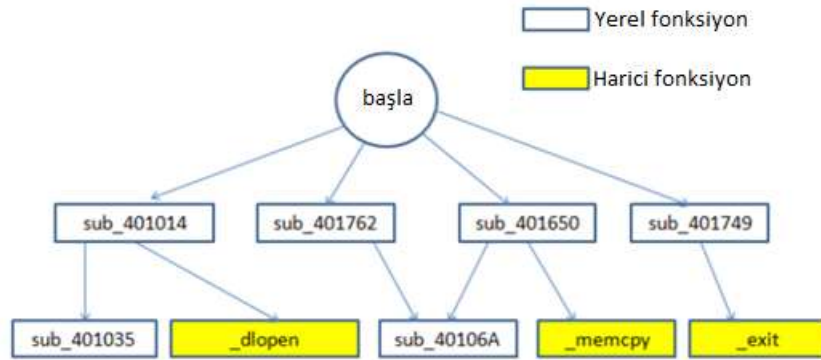
“xxxx” kısmı her fonksiyona özgü adı temsil eder. Yerel fonksiyonlar bitiminde “ends” veya “sub_xxxx endp” ile işaretlenir.

İşlevleri aynı olsa dahi her programda yerel fonksiyonların ismi farklıdır. Ancak harici fonksiyonlar tüm programlarda aynı adla çağırılır.

3.3 Fonksiyon Çağrı Graflarının Oluşturulması

İlk adım olarak, bir kötücül yazılım örneğinin exe dosyasından IDA Pro programı kullanılarak assembly kodu elde edilir. Daha sonra bu kod kullanılarak, tüm fonksiyon isimleri ve fonksiyon özellikleri sınıflandırılır.

Tablo 3.7, disassembly işleminden sonra NGVCK virüsündeki bir fonksiyonu ve bu fonksiyona ait özellikleri temsil eder. Assembly kod içerisindeki her bir fonksiyon vertex olarak ve fonksiyonların çağırdıkları fonksiyonlar edge olarak eklenerek fonksiyon çağrı grafikleri oluşturulur. Şekil 3.3 Mwor virüsün fonksiyon çağrı grafiğinden bir parçadır. “_dlopen”, “_memcpy”, “_exit” birer harici fonksiyondur. Yerel fonksiyonlar, harici fonksiyonları çağırabilirken; harici fonksiyonlar yerel fonksiyonları çağıramazlar.



Şekil 3.3: Mwor virüsü fonksiyon çağrı grafi parçası (Xu, 2010).

Tablo 3.7: NGVCK virüsü örnek fonksiyonu ve fonksiyon grafi tablosu

NGVCK Virüsü Örnek Fonksiyon Assembly Kodu	Fonksiyon Kodundan Çıkarılmış Graf Özellikleri
<pre> Sub_40106A proc near MOV ecx, ss:dword_4015B6[ebp] MOV esi, [ecx+3Ch] ADD esi, ecx MOV edx, [esi+3Ch] MOV ecx, ss:dword_40141C[ebp] ADD ecx, 795h MOV ss:dword_4015D8[ebp], ecx MOV ss:dword_4015BA[ebp], edx CALL sub_401762 MOV ecx, ss:dword_4015D8[ebp] MOV ss:dword_4015FE[ebp], ecx PUSHA CALL sub_401749 POPA MOV ss:dword_40141C [ebp], ecx CALL sub_4016E7 PUSH ss:dword_4015B6[ebp] POP esi MOV edx, [esi+3Ch] Sub_40106A endp </pre>	<pre> Function_name: Sub_40106A Function_Type: Local subroutine Function indegree: 2 Function outdegree: 3 Function's callees: sub_401762, sub_401749, sub_4016E7 function's callers: sub_401650, sub_4016BE Opcode sequence: Mov, mov, add, mov... .. push, pop, mov </pre>

Breadth First Search (BFS) ve Depth First Search (DFS) teknikleri, gereksinimlere bağlı olarak graf yapımında kullanılır. BFS tekniğinde, birinci seviye düğümlerden (kök düğümleri) başlar ve daha sonra ikinci seviye düğümlerden ilerler. Bu teknik ile graf oluşumunda her bir fonksiyon düğümünü dolaşmak mümkün olur. Bu sebeple BFS algoritması kullanmak caller-callee (çağırır-çağırılan) ilişkisi ile fonksiyonlar boyunca graf oluşturulur (Shang, 2010). Graf oluşumunda fonksiyonlardaki “call” çağrı komutları kullanılır. Çağırır fonksiyon ata, çağırılan fonksiyon halef olarak kabul edilir.

Graf oluşturma algoritması, başlangıç noktasından başlar ve her bir fonksiyonda dolaşarak özellikleri arasında “call” çağrısı ile çağırılan fonksiyonları tutar. Tüm fonksiyonlar işleminden geçtikten sonra, fonksiyon çağrı grafi oluşturulur. Tablo 3.8, fonksiyon çağrı grafini oluşturmak için kullanılan bir algoritmayı tarif etmektedir. Tüm bunlardan önce, algoritma executable dosyasından IDA PRO

programını ile çıkarılmış olan assembly kodunu okuyarak her bir fonksiyon için sınırlarını ve özelliklerini belirler. Grafta uğranılmamış tüm fonksiyonlar Functionset'te saklanır ve grafta uğranılmış tüm fonksiyonlar EntryFunctionSet'te saklanır. FunctionQueue fonksiyonlarla birlikte başlatılır ve queue tamamen boşalana kadar algoritma devam eder. Caller (çağırıcı) fonksiyonları tailVertex'leri ifade eder ve vertex olarak tailVertex içine depolanır. Ve daha sonra, tailVertex graf'a eklenir, enqueueFlag, aynı vertex'in tekrar aynı değere sahip olmadığından emin olmak için true değerine ayarlanır. Daha sonra, algoritma tailVertex'in opcode dizisini geçerek callee (çağırılan) setini belirler.

Callee seti elde edildikten sonra tek tek headVertex içine kaydedilir. Her bir headVertex için grafın tailVertex'ten headVertex'e kenarının olup olmadığı araştırılır. Eğer bir kenar bulunursa, caller derecesi ve callee derecesi bir artırır. Bulunamadığı durumda, headVertex ve onun ilişkili olduğu kenar tailVertex ile birlikte grafa eklenir. Sonunda, headVertex'in enqueueFlag değeri true değilse, true değerine ayarlanır ve headVertex, sıranın sonuna eklenir. Bu algoritmanın zaman karmaşıklığı, $O(|V| * |E|)$ şeklindedir. V toplam köşe sayısı ve E toplam kenar sayısını ifade eder (Shang, 2010).

Tablo 3.8: Fonksiyon çağrı graflarının oluşturulması algoritması (Deshpande, 2013)

```
// Input: Assembly file M, Output: Function call graph GM
// Initializations
GM.V =  $\varnothing$  and GM.E =  $\varnothing$ 
EntryFunctionSet =  $\varnothing$ , FunctionSet =  $\varnothing$ , FunctionQueue =  $\varnothing$ , VertexSet =  $\varnothing$ 
FunctionSet = ExtractFunction(M)
EntryFunctionSet = ExtractEntryFunction(M)
FunctionQueue = InitQueue(EntryFunctionSet)
while(FunctionQueue is not empty)
    tailVertex = Dequeue(FunctionQueue)
    Insert tailVertex in GM
    tailVertex.enqueueFlag = true
    VertexSet = getCallee(tailVertex)
    for each vertex in VertexSet
        if(vertex is not in FunctionSet)
            Continue
        Endif
    headVertex = vertex
    // Construct an edge between tailVertex and headVertex
    if(e  $\in$  GM.E)
        tailVertex.outdegree++
        headVertex.indegree++
    Else
        Insert headVertex in GM
        Insert edge e in GM
    Endif
    if(headVertex.enqueueFlag == false)
        Enqueue headVertex in FunctionQueue
        headVertex.enqueueFlag = true
    Endif
    next vertex
end while
return GM
End
```

3.4 Fonksiyon Çağrı Grafları Benzerliği Ölçümü

Assembly dosyasından elde edilen, bir virüsün iki farklı varyantı için oluşturulan fonksiyon çağrı grafikleri arasındaki benzerlik ne kadar fazla ise şüpheli dosyanın metamorfik virüs olma olasılığı o kadar fazladır. Graflar, fonksiyonların çağırıldıkları ve çağırıldıkları diğer fonksiyonlarla ilişkisinden oluşturulur. Bu graflar arasındaki benzerliği bulmak için farklı teknikler kullanılabilir. Harici fonksiyonların benzerliğini bulmak, yerel fonksiyonların benzerliğini bulmaya kıyasla daha kolaydır. Çünkü harici fonksiyonlar her programda aynı isimle adlandırılan ve aynı görevi yapan

fonksiyonlardır. Eşleştirme yapmak için de sadece isimlerinin benzerliğini bulmak yeterlidir. Fakat aynı metamorfik virüsün iki farklı varyantı arasında aynı işlevi yapan yerel fonksiyonların isimleri farklıdır (Ming, 2013), bu sayede yakalanmaktan kaçınırlar. Yöntem olarak yerel fonksiyon benzerliklerini ölçmek için opcode dizisi karşılaştırmasında LCS algoritması kullanılmıştır.

3.4.1 Harici Fonksiyon Benzerliği

Harici fonksiyonlar işletim sistemi tarafından oluşturulan ve oluşturulduğu programa göre farklılık göstermeyen, aynı isimle aynı işlevi yapan sistem veya kütüphane fonksiyonlarıdır. Ayrıca atomic fonksiyonlar olarak da bilinir (Deshpande, 2013). Graf teorisine göre harici fonksiyonlar, yaprak (leaf node) düğümlerdir. Bu sebeple in-degree değeri 1, out-degree değeri 0'dır (Carrera, 2004).

Harici fonksiyonlar sembolik isimlerine göre eşleştirilebilir. Örneğin, bir programdaki "GetVersion" fonksiyonu diğer programlarda aynı işlevle eşleştirilebilir. Tablo 3.9, harici fonksiyonları eşleştirmek için kullanılan bir algoritmayı göstermektedir (Ming, 2013).

Tablo 3.9: Harici fonksiyon eşleştirme algoritması (Deshpande, 2013).

```
// Input: Function call graph G1 and G2 Output: common vertex (G1, G2)
ExternalfuncSet1 ← External function from G1
ExternalfuncSet2 ← External function from G2
Copy vertices from G1 into Us
Copy vertices from G2 into Vs
foreach vertex Usi ExternalfuncSet1 do
    foreach vertex Vsj ExternalfuncSet2 do
        if(Usi.name = Vsj.name)
            common vertex(G1, G2) ← common vertex(G1, G2) U (Usi, Vsj )
            Remove Usi from Us
            Remove Vsj from Vs
        End
    End
End
End
End
```

Harici fonksiyonlar assembly dosyası içerisinde elde edilen G1 ve G2 graflarından çıkarılır ve sırasıyla ExternalfuncSet1 ve ExternalfuncSet2'ye kopyalanır. Daha sonra her iki set içinde aynı isimleri bulmak için çaprazlanır. Ad eşleşmesi varsa, bu vertex, ortak vertex çiftine kopyalanır.

3.4.2 Aynı Harici Fonksiyonları Çağırarak Yerel Fonksiyonları Bulma

İki yerel işlev, iki veya daha fazla benzer dış işlev çağırıyorsa eşleşiyor olarak kabul edilir (Carrera, 2004). Tablo 3.10, eşleşen fonksiyon çiftini bulmak için kullanılan bir algoritmayı göstermektedir (Ming, 2013).

Tablo 3.10: Harici fonksiyon tabanlı yerel fonksiyon benzerliğini bulma algoritması (Deshpande, 2013)

```
// Input: Function call graph G1 and G2,Us,Vs,common vertex(G1, G2)
//Output: common vertex(G1, G2)
foreach vertex Ui Us do
    foreach vertex Vj Vs do
        if(ExternalFunction(Ui) ∩ ExternalFunction(Vj) ≥ 2) then
            common vertex(G1, G2) ← common vertex(G1, G2) ∪ (Ui,Vj)
            Remove Ui from Us
            Remove Vj from Vs
        End
    End
End
End
```

3.4.3 Komşu Düğümlere Göre Yerel Fonksiyon Benzerliği

İki köşe eşleşiyor ise komşularının da eşleşmesi daha olasıdır (Ming, 2013). Bir grafikte, köşe komşuları onun ardılları ve öncülleridir. Şekil 3.4'te gösterildiği gibi, bir grafiğin köşe noktası A, diğerinin köşe noktası B ile eşleştirilmiştir. Daha sonra, bir grafikteki U, V ve W noktalarının X, Y ve Z köşe noktaları ile eşleşebilmeleri, diğer komşulardan daha önce eşleşmiş olan köşe noktalarına doğrudan

erişebilmeleridir. Şekil 3.4'te aynı rengin köşe noktaları benzer şekilde muamele edilir.



Şekil 3.4: Benzer köşelerin komşularının benzerliği (Deshpande, 2013)

Tablo 3.11 komşu düğümlerin eşleşme algoritmasını göstermektedir. Algoritma komşu düğümlerin benzerliğini ölçmek için her bir düğümün diğer düğümlerle komşularını tek tek karşılaştırma yapar. Kuyruk (queue), ortak düğüm ile başlatılır. Ardından, algoritma sıra boşalana kadar çalışır, her iterasyonda ilk sıradaki silinir. Komşu düğümler tek tek ortak düğümü dolaşır.

Tablo 3.11: Eşleşmiş düğümlerin komşu düğümleri için eşleşme (Ming, 2013)

```

// Input: Function call graph G1 and G2,Us,Vs,common vertex(G1, G2)
//Output: common vertex(G1, G2)
vertexQueue ← InitvertexQueue(common vertex(G1, G2))
while (vertexQueue is not empty)
  (u,v) ← vertexQueue.dequeue()
  foreach vertex Usi successor(u) ∩ Us do
    foreach vertex Vsj successor(v) ∩ Vs do
      If(color relaxed sim(Usi,Vsj) ≥ δ) then
        common vertex(G1, G2) ← common vertex(G1, G2) ∪ (Usi,Vsj)
        Remove Usi from Us
        Remove Vsj from Vs
        vertexQueue.enqueue(Usi,Vsj)
      End
    End
  End
End
return common vertex(G1, G2)
  
```

3.4.4 Fonksiyon Çağrı Grafları Arasındaki Benzerlik Ölçümü

İki çağrı grafi arasındaki ortak düğümler bulunduktan sonra benzerlik oranı hesaplanır. Benzerlik oranı aşağıdaki denklemde verilmiştir (Ming ve diğ. 2013);

$$sim(G1, G2) = \frac{2|common_edge(G1, G2)|}{|E(G1)| + |E(G2)| * 100} \quad (3.1)$$

Denklem (3.1)'deki eşitlikte `common_edge(G1, G2)` olarak ifade edilen kısım, graf 1 ve graf 2 için ortak olan kenarların sayısal ifadesidir. `|E(G1)| + |E(G2)|` kısmı ise graflardaki toplam kenar sayısını ifade eder.

3.5 Jaya Algoritması ile Opcode Dizisi Benzerliği Ölçümü

Bu çalışmada bir şüpheli dosyanın iki varyantı arasındaki opcode'ların benzerliğini ölçmek için LCS (longest common subsequence) algoritmasından yararlanılmıştır. Ancak bu problemin çözümü için önemli olan nokta opcode'lar için baz alınan kategorilerin sınırlarının belirlenmesidir. En uygun opcode kategorisini bulabilmek için bu kategorilerden popülasyon oluşturularak Jaya optimizasyon algoritmasında kullanılmış ve güvenilir bir sonuç elde edilmiştir.

3.5.1 Opcode Kategorilerinin Sınırlarının Belirlenmesi

Bir executable dosyanın disassembler yardımı ile çıkarılan assembly kodlarından opcode dizisi elde edildikten sonra opcode kategorisi için Tablo A.23'de görülen sınıflandırma baz alınmıştır. Bu sınıflandırma opcode işlevleri göz önüne alınarak hazırlanmıştır. Uygulama içerisinde, bu sınıflandırma, fonksiyon opcode'larının kategorisini belirlemek ve bu kategorizasyon işlemi ile en uygun çözümü bulmak için Jaya algoritmasında kullanılmıştır.

Tablo 3.12'de algoritması verildiği üzere opcode sınırlarını belirleme, birden otuz dört'e kadar rastgele olarak seçilen bir bölüt değeri belirleyerek başlanır.

Belirlenen sayı her bir kategoriyi sınıflandırmak ve bir opcode kategorisi içerisinde bulunacak opcode sayısını ifade etmek için kullanılır.

Tablo B.25, örnek olarak, her bir kategorisi dörde bölütlenmiş opcode sınıflandırmasını göstermektedir. Kategori içerisinde opcode sayısı belirlenen bölüt değeri ile tam bölünmüyorsa kalan sayı da aynı şekilde yeni gruplandırmaya dahil olur. Bu örnekten yola çıkarak opcode dizisini dörtlü bölütlere ayırdığımızda Tablo A.23’de 0 olarak belirtilen kategorinin yedi farklı sınıfa bölündüğü, 1 olarak belirtilen kategorinin ise opcode sayısı dört olduğu için tek bir adet sınıfı temsil ettiği görülmektedir. Opcode alt bölütlerini elde etmek için sıfırdan başlayarak her bir grup rakam ile işaretlenir. On beş kategorinin tamamı işaretlendiğinde yeni oluşan liste opcode alt bölütleri olarak adlandırılır.

Tablo 3.12: Opcode sınırlarının belirlenmesi algoritması

```
// Input: opcode_list
// Output: Split_opcode_list
Count <- number of segments
Foreach (count on opcode_list(count))
    ArrayList(count)<- opcode_list(count)
    Split ArrayList(count)
    Foreach (ArrayList(count) size)
        HashMap(count, ArrayList)
    End
End
```

Fonksiyonlar içerisindeki her bir komut, oluşturulan opcode dizisi kategorisinde komutun karşılığı olan sınıflandırma grubu ile birlikte kaydedilir. Bu haliyle her fonksiyonun kendine özgü bir katar değeri oluşturulmuş olur. Tablo 3.15 örneğinde ise bir katar dizisinin, dörtlü kategorize edilmiş opcode dizisi (Tablo B.25) içindeki karşılıkları yer almaktadır. Bu fonksiyon parçası için karşılaştırma yapılacak olan katar, Tablo 3.13 algoritması kullanılarak “mov-mov-call-sub-push-pop-jz-xor-add-proc” dizisi için “0-0-34-13-7-7-34-26-13-45” olarak işaretlenir.

Tablo 3.13: İşaretlenmiş fonksiyon dizisi oluşturma algoritması

```
//Input: Variant 1 Functions, Variant 2 Functions, split_output_list
//Output: Marked Function List1, Marked Function List2
Init Marked_list
While(Function1 and Function2 is not empty)
    Marked_list(split_output_list) <- opcode in function
End
```

Tablo 3.14: Bölütlenmemiş opcode dizisine göre işaretlenmiş fonksiyon örneği

MOV	OR	CALL	SUB	PUSH	POP	JZ	XOR	ADD	PROC
0	6	8	5	1	1	9	6	5	13

Tablo 3.14, 4 ile bölütlenmiş opcode dizisine göre işaretlenmiş fonksiyon örneğidir. Komut listesinde yer alan her komut bir rakam ile işaretlenir.

Tablo 3.15: Dörtlü bölütlenmiş opcode dizisine göre işaretlenmiş fonksiyon örneği

MOV	OR	CALL	SUB	PUSH	POP	JZ	XOR	ADD	PROC
0	21	34	13	7	7	35	26	13	45

Tablo 3.14'te, Tablo A.23'deki bölütlenmemiş opcode dizisine göre işaretlenmiş fonksiyon örneği verilmiştir. Tablo 3.15'te ise dörtlü bölütlenmiş opcode dizisine göre işaretlenmiş fonksiyon örneği verilmiştir. Tablo 3.14'te hem OR hem de XOR komutu aynı bölütte ifade edilir. Ancak Tablo 3.15'te bölütleme işlemi ile ölçümün hassasiyeti artırılmış ve farklı bölütleri ifade eder hale getirilmiştir.

Bölüt değeri artması her bir kategori için ayrı işlem yapılması bakımından hızın azalmasına sebep olur ancak kategori sayısını artıracığı için hassasiyeti artırır daha doğru bir tespit yapmaya yaklaştırır. Bu sebeple bölüt değerine karar vermek ve en uygun çözümü bulmak için bu çalışmada Jaya optimizasyon algoritması kullanılmıştır.

3.5.2 İşaretlenmiş Fonksiyon Dizilerinin Karşılaştırılması

LCS algoritması dinamik programlama yöntemidir ve verilen iki katar arasındaki en uzun ortak katarı bulmaya çalışır. Tablo 3.16’da X ve Y olarak sembolik isimlendirilen iki katar arasındaki en uzun ortak katarı bulmak için, algoritma X ve Y katarlerinin uzunluklarını sırasıyla m ve n değerlerine atayarak başlar, daha sonra bir matris oluşturularak her bir katar parçası diğer katar içinde aranır. Algoritma benzer katar parçasına rastladığında eski benzer parça ile kıyaslar ve daha uzun olanı tutar.

Tablo 3.16: LCS algoritması sözde kodu

```
m ← length[X]
n ← length[Y]

for i ← 1 to m
  c[i,0] ← 0
for j ← 1 to n
  c[0,j] ← 0

for i ← 1 to m
  for j ← 1 to n
    if (x_i == y_j) {
      c[i,j] ← c[i-1,j-1] + 1
      b[i,j] ← NW
    }
    else if (c[i-1,j] >= c[i,j-1]) {
      c[i,j] ← c[i-1,j]
      b[i,j] ← N
    }
    else {
      c[i,j] ← c[i,j-1]
      b[i,j] ← W
    }
  }
```

İlk varyantta yer alan birinci fonksiyon ikinci varyantın ilk fonksiyonundan başlar ve son fonksiyonuna kadar karşılaştırma yapar. Daha sonra ilk varyantın ikinci fonksiyonuna geçilir, bu sıra takip edilerek tüm fonksiyonlar karşılaştırılır. Benzer parça bulunduğu zaman parçanın uzunluğu “Lcs_rate_temp” değeri olarak kaydedilir. Eğer yeni bulunan LCS rate değeri eski değerden daha büyük ise değer güncellenir ve opcode dizilerinden bulunmuş olan ortak parça silinir, döngünün başına tekrar dönlür. Bu işlem opcode dizilerinden herhangi birinde değer kalmayana dek devam eder. Son LCS rate değeri ise maksimum uzun parçanın değeri olarak çıktı verilir.

Bulunan maksimum değerler fonksiyon sayısına bölünerek ortalama benzerlik değeri bulunur. Bu işlem sadece bir opcode dizisi kategorisi için geçerlidir ve daha sonra Jaya algoritmasında kullanılmak üzere kaydedilir.

İki varyant için oluşturulan fonksiyon katarları bir matris gibi düşünülerek LCS algoritması ile karşılaştırılır. Karşılaştırılan iki fonksiyon için benzerlik değeri %70'den daha büyük ise benzer olarak kabul edilmiştir ve elde edilen değer, iki fonksiyonun benzerlik değerini oluşturur. Tablo 3.17'de opcode dizisi benzerliği bulma algoritması verilmiştir, bu algoritma, iki varyant için işaretlenmiş fonksiyon listelerini giriş olarak başlar. İlk varyantın birinci fonksiyonundan başlayarak diğer varyant içerisindeki fonksiyonları tek tek kontrol eder ve algoritmada işlem gören fonksiyona en benzer olan fonksiyonun LCS değerini bir arraylist içerisine atar. Tüm fonksiyon benzerliği işlemleri bittiğinde iki varyantın fonksiyon sayılarının çarpımı kadar LCS değeri elde edilmiş olur. LCS değerlerini tutan arraylist içerisindeki maksimum değer, devam eden iterasyonun benzerlik değeri olarak tanımlanır ve yeni iterasyona geçilir. Sonlandırma kriteri sağlanıncaya kadar bu döngü devam eder.

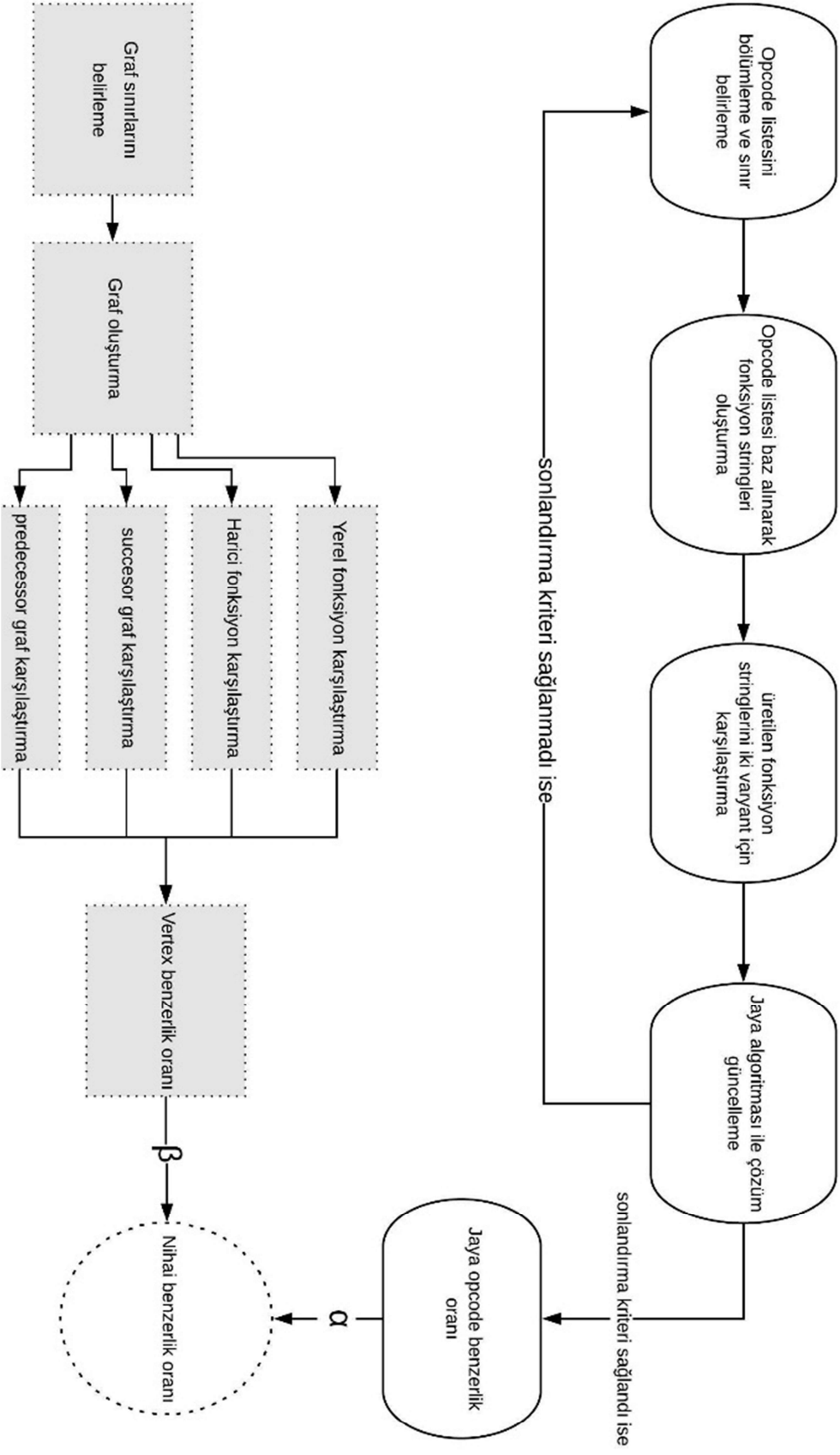
Tablo 3.17: Önerilen opcode dizisi benzerliği bulma algoritması

```
//Input: Marked Function List1, Marked Function List2, Lcs rate=0, common_katar
from LCS
//Output: longest common subsequence rate
Foreach(int i : Marked Function List1)
    Foreach(int j: Marked Function List2)
        Opcode_katar1 ← Opcode strings from Marked Function List1(i)
        Opcode_katar2 ← Opcode katars from Marked Function List2(j)
        While(opcode_katar1 or opcode_katar2 is not empty)
            Lcs rate ← Lcs(opcode_katar1, Opcode_katar2)
            If(Lcs rate_temp>Lcs rate)
                Update Lcs rate
            Remove common_katar from opcode_string1
            Remove common_katar from opcode_string2
        End
    End
End
End
```

3.5.3 Jaya Algoritması ile Opcode Benzerlik Oranı Bulma

Bir optimizasyon algoritması olan Jaya, en uygun çözümü bulmada hem hızlı olması hem de parametresiz olması yönüyle oldukça avantajlıdır. Büyük boyutta opcode kategorilerinin birer popülasyon olarak alındığı ve çok kez çalıştırıldığı bu çalışmada hızlı bir algoritma oldukça önemlidir.

Vertex benzerliği ölçmek için assembly kodunda fonksiyonların çağrı grafiği kullanılır ve fonksiyon içerisinde ne kadar çok “call” komutu ile çağrı yapılırsa ölçümün doğruluk payı o kadar yüksek olur. Bunun yanısıra opcode benzerliği yöntemi içerisinde hiç “call” komutu geçerse dahi benzerliği sıralı opcode’lar içerisinde ölçülebilir. Aynı zamanda opcode benzerliği yöntemi metamorfik gizlenme teknikleri olan “fonksiyonların yerini değiştirme”, “değişken isimlerini değiştirme” ve “işlevsiz kod ekleme” yöntemlerinden bir çoğu için bütünüyle bir tespit yöntemi olarak kullanılabilir. Vertex benzerliği yönteminde de içerisinde “call” komutu yoğun olarak bulunmayan assembly dosyalarında benzerlik ölçümünde dezavantajlı duruma düşmektedir. Aynı şekilde opcode benzerliği yönteminde ise yüksek yapay zeka kullanılmış metamorfik kötücül kod assembly dosyaları için dezavantajlıdır. Bu sebeple, Şekil 3.5’te görüldüğü üzere uygulama, benzerlik ölçümü için iki farklı yöntem kullanır ve bunları oransal olarak birleştirir. Bunun temel sebebi vertex benzerliği ve opcode benzerliği yöntemlerinin birbirlerinin dezavantajını ortadan kaldırmaya çalışmasıdır. Bu sebeple Deshpande (2013), tarafından kullanılmış olan vertex benzerliği bu çalışmada da opcode benzerliği için dengeleyici unsur olarak kullanılmıştır.



Şekil 3.5: Önerilen Benzerlik ölçüm şeması

α ve β katsayıları, opcode benzerliği ve vertex benzerliği birleşimi için kullanılan 0 ile 1 arasında katsayı değerleridir. Opcode benzerliğinin α katsayısı ile çarpılması ve vertex benzerliği oranının β katsayısı ile çarpılması, daha sonra bu çarpımların toplanması ile iki varyant arasındaki benzerlik değerine ulaşılır. α ve β katsayılarının belirlenmesi Bölüm 3.6'da açıklanmıştır.

Jaya algoritması ile opcode benzerliği ölçümü, öncelikle birden otuz dört'e kadar rastgele seçilen bölüt değerleri başlar. Otuz dört bölüt değeri, opcode listesinin maksimum uzunluğu göz önüne alındığı için en uygun bölüt değeri olarak seçilmiştir. Bölüt değeri opcode listesini gruplandırmak için kullanılır. Metamorfik kötücül kodların iki farklı varyantı içerisindeki fonksiyonların her birinin komut listesi bir katar haline dönüştürülür. Ve bu katar dizisi, gruplandırıldıktan sonra elde edilen yeni opcode listesini kullanarak işaretlenmiş bir katar dizisi oluşturur.

İki varyant için ayrı ayrı oluşturulmuş işaretli katar dizilerini karşılaştırmak ve benzerlik oranı bulmak için LCS algoritması kullanılır. Her fonksiyon için karşılaştırıldığı diğer fonksiyonlar arasından aldığı en yüksek değeri benzerlik oranı olarak kabul edilir. Oran, 0 ile 1 arasındadır ve bir katar içerisinde en uzun var olan katarın toplam uzunluğa bölünmesi ile bulunur. Ne kadar uzun bir benzer katar bulunursa o kadar 1'e yaklaşılmış olur.

Tablo 3.18'de görüldüğü üzere Jaya algoritması popülasyon değerini ve sonlandırma kriterini belirleyerek başlar, sonlandırma kriteri 10 döngü şeklinde belirlenmiştir. Popülasyon büyüklüğü birden otuz dört'e kadar seçilmiş olan bölüt sayıları ile oluşturulan opcode kategorileri içersinden rastgele seçilen 10 adet opcode kategorisini ifade eder. Her opcode kategorisi için opcode benzerlik oranı tekrar hesaplanır.

Nihai benzerlik değeri (NBD), opcode benzerlik değeri ve vertex benzerlik değerinden sayıca daha büyük olanı alınarak (3.2) eşitliği ile bulunur.

$$NBD = MAX(opcode\ benzerlik\ değeri, vertex\ benzerlik\ değeri) \quad (3.2)$$

Tablo 3.18: Jaya algoritması ile opcode benzerliği ölçümü

```

//Input: Population_size; Termination Condition; opcode_rate;
//Output: Optimum result
While(Termination_Condition)
Best <- Max value of opcode_rates
Worst <- Min value of opcode_rates
r1 < Random(0,1)
r2 < Random(0,1)
 $X_{J,K,i}' = X_{J,K,i} + r_{2\ i,j}(X_{J,K,i} - |X_{j,best,i}|) - r_{1\ i,j}(X_{J,K,i} - |X_{j,worst,i}|)$ 
If ( $X_{J,K,i}' > X_{J,K,i}$ ) Then
    Update the previous solution
Else
    No update in the previous solution
End

```

En iyi (Best), opcode benzerlik sonuçları arasında maksimum değer ve aynı şekilde En kötü (Worst), opcode benzerlik sonuçları arasında minimum değerdir. Daha sonra algoritma içerisinde random olarak 0 ve 1 arasında değer üretilir bu değerler (3.3) eşitliğinde r1 ve r2 olarak gösterilmiştir. Çözüm değerini güncellemek için mevcut çözüm en iyi ve en kötü değerlerden çıkarılır ve random üretilmiş olan değerler ile çarpılır. Daha sonra, sonuca mevcut değer eklenerek güncel çözüm bulunur. Güncel çözüm mevcut çözümden daha iyi olması durumunda mevcut çözümün yerini güncel çözüm alır.

$$X_{J,K,i}' = X_{J,K,i} + r_{2\ i,j}(X_{J,K,i} - |X_{j,best,i}|) - r_{1\ i,j}(X_{J,K,i} - |X_{j,worst,i}|) \quad (3.3)$$

Her bir opcode kategorisi için değerler üretildikten sonra maksimum değer tutulur ve Jaya algoritması, sonlandırma kriterine kadar tekrar eder. Algoritma bittiğinde mevcut çözüm optimum sonucu ifade eder. Bu sonuç opcode benzerliği için bulunan sonuçtur.

Tablo 3.19’da Mwor metamorfik virüsünün iki farklı varyantı arasında ölçülen opcode benzerliğinin Jaya iterasyonları ile değişimi gösterilmektedir. Bahsi geçen iterasyon, Jaya optimizasyon algoritmasının her bir döngüsünü ifade eder. Vertex benzerlik oranı, yerel fonksiyon benzerliği, harici fonksiyon benzerliği ve graf komşu benzerliği ile ölçülmüş olan vertex benzerlik oranını ifade eder. Bölüt değerleri, on adet birden otuz dört’e kadar rastgele değer içeren diziden oluşur. Rastgele değerler birden fazla kez seçilebilir.

Her bir bölüt değeri daha sonra opcode kategorisi oluşturmak için kullanılır. Bir opcode kategorisi içerisinde bölüt değeri kadar opcode oluşturulur. Oluşturulan kategoriler ile her iterasyonda her fonksiyon için yeniden işaretleme yapılır ve diğer varyant ile karşılaştırılır. Bu işlemde elde edilen benzerlik oranları Tablo 3.19’da opcode benzerlik oranları olarak gösterilmiştir. Jaya çözüm değeri, her iterasyonda bölümlenmiş opcode kategorisi kullanarak ölçülen Jaya değerini ifade eder ve her iterasyonda güncellenir.

Tablo 3.19: Mwor virüsü örneği

İterasyon	Vertex Benzerlik Oranı	Bölüt Değerleri	Opcod Benzerlik oranları(%)	Jaya Çözüm Değeri
İ1	% 80.0	{28,2,25,25,7,19,22,10,6,22}	[88.0, 85.19, 87.36, 88.51, 85.19, 87.36, 87.36, 87.36, 87.36, 88.51]	%87.57
İ2	% 80.0	{10,25,0,14,9,9,21,8,12,10}	[87.36, 87.36, 87.36, 88.51, 87.36, 88.0, 88.51, 88.51, 87.36, 88.0]	%87.61
İ3	% 80.0	{22,21,1,22,5,28,24,12,18,15}	[87.36, 88.0, 88.51, 87.36, 86.67, 87.36, 87.5, 87.36, 87.36, 87.36]	%87.72
İ4	% 80.0	{3,24,21,19,24,24,15,27,18,1}	[86.67, 86.67, 88.0, 87.36, 86.67, 85.19, 87.36, 87.5, 87.36, 87.36]	%87.78
İ5	% 80.0	{11,14,14,5,27,19,5,7,27,20,}	[86.67, 86.67, 88.0, 87.36, 88.0, 88.0, 88.0, 87.36, 87.36, 88.0]	%87.78
İ6	% 80.0	{11,12,26,12,1,28,27,0,10,12}	[87.36, 87.36, 87.36, 87.36, 88.51, 87.36, 88.0, 87.36, 87.36, 88.0]	%87.80
İ7	% 80.0	{8,22,10,0,1,8,2,0,18,4}	[87.36, 85.19, 87.36, 87.36, 87.5, 87.36, 87.36, 87.36, 85.19, 87.36]	%87.87
İ8	% 80.0	{20,14,26,6,14,9,22,27,12,19}	[87.5, 87.36, 87.36, 87.36, 87.5, 88.0, 86.67, 87.36, 87.36, 88.51]	%87.92
İ9	% 80.0	{12,2,1,4,13,18,21,17,29,0}	[86.67, 87.36, 88.51, 87.36, 88.0, 88.0, 85.19, 86.67, 87.36, 88.51]	%87.99
İ10	% 80.0	{3,13,18,20,1,16,29,24,14,21}	[87.36, 87.36, 86.02, 86.67, 87.36, 88.51, 85.19, 86.67, 87.36, 87.36]	%88.06

Tablo 3.20: Mwor virüsü çağrı komutu azaltılmış örneği

İterasyon	Vertex Benzerlik Oranı	Bölüt Değerleri	Opcode Benzerlik oranları	Jaya Çözüm Değeri
İ1	%60.0	{15,24,0,3,6,4,11,20,8,5}	[86.96, 87.3, 85.19, 87.5, 86.96, 86.11, 86.96, 86.96, 86.96, 86.9]	%87.03
İ2	%60.0	{4,25,1,16,23,13,10,26,6,24}	[86.11, 87.3, 85.33, 86.96, 87.3, 86.96, 86.96, 86.36, 86.96, 87.3]	%87.19
İ3	%60.0	{11,3,11,25,25,11,9,13,18,27}	[86.96, 87.5, 86.96, 87.3, 87.3, 86.96, 86.96, 86.96, 86.96, 86.36]	%87.27
İ4	%60.0	{23,9,14,19,24,29,23,24,29,17}	[87.3, 86.96, 86.96, 86.96, 87.3, 86.36, 87.3, 87.3, 86.36, 86.96]	%87.34
İ5	%60.0	{29,13,20,17,18,22,25,7,18,16}	[86.36, 86.96, 86.96, 86.96, 86.96, 86.96, 87.3, 87.3, 86.96, 86.96]	%87.36
İ6	%60.0	{0,12,21,18,6,3,1,4,5,15}	[85.19, 86.96, 86.96, 86.96, 86.96, 87.5, 85.33, 86.11, 86.9, 86.96]	%87.57
İ7	%60.0	{11,27,5,15,4,23,18,15,2,13}	[86.96, 86.36, 86.9, 86.96, 86.11, 87.3, 86.96, 86.96, 86.11, 86.96]	%87.66
İ8	%60.0	{22,22,8,12,29,11,23,25,14,3}	[86.96, 86.96, 86.96, 86.96, 86.36, 86.96, 87.3, 87.3, 86.96, 87.5]	%87.74
İ9	%60.0	{14,23,19,22,29,3,22,12,10,24}	[86.96, 87.3, 86.96, 86.96, 86.36, 87.5, 86.96, 86.96, 86.96, 87.3]	%87.74
İ10	%60.0	{8,16,11,1,11,3,20,27,2,26}	[86.96, 86.96, 86.96, 85.33, 86.96, 87.5, 86.96, 86.36, 86.11, 86.36]	%87.75

Tablo 3.19, Mwor virüsünün rastgele seçilen ve içerisinde çağrı komutu bulunduran fonksiyon örnekleri ile oluşturulmuştur. Tablo 3.20 ise Mwor virüsünün fonksiyonları ile hazırlanmış “call” komutu yoğunluklu olmayan (%50’den daha az) fonksiyonları deney grubu olarak seçilmiştir.

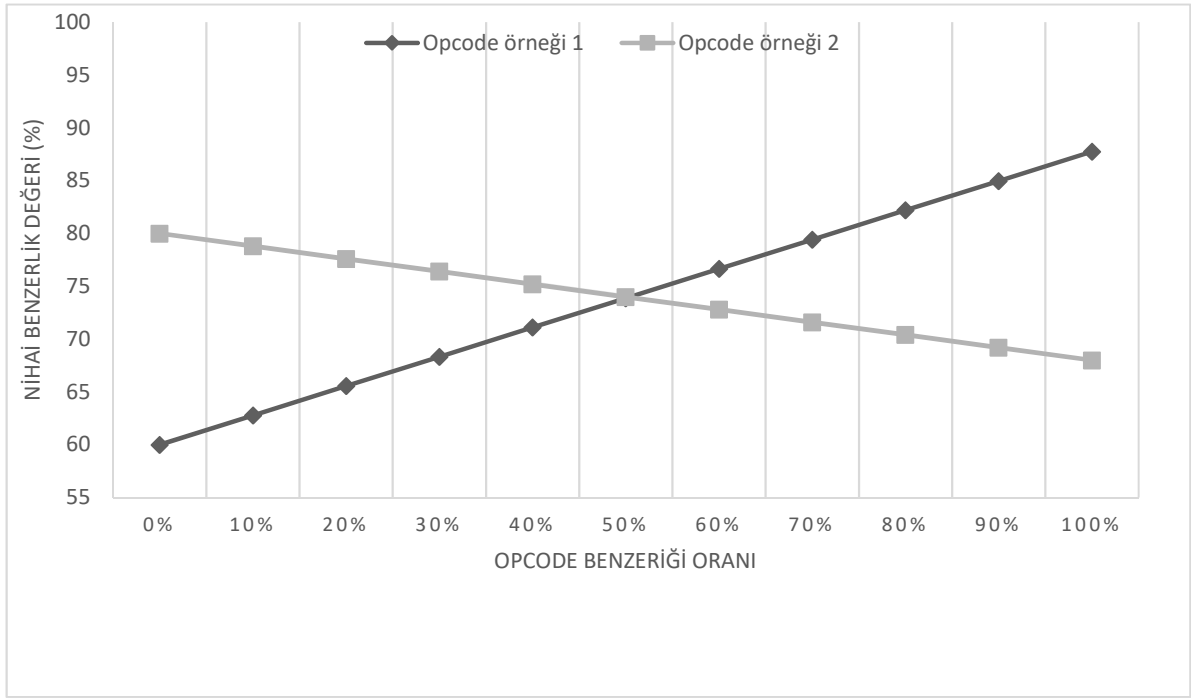
Vertex benzerliği ölçümünde “call” komutu graf oluşumunda etkili olduğu için çağrı komutu içermeyen veya daha az içeren fonksiyonların ölçümünde bu vertex benzerliği yöntemi yetersiz kalmaktadır. Bu dezavantajı ortadan kaldırmak için benzerlik ölçümünde opcode benzerliğinin de dikkate alınması gerekmektedir. Tablo 3.20 içerisinde “call” komutu daha az barındıran fonksiyonlardan oluşan deney grubunda opcode benzerliği ölçümünde aynı virüsün farklı varyantları olduğunu %50’nin üzerinde benzerlik göstererek (Jaya ölçümü ile %88.06) ölçebilmiştir.

Vertex benzerliği yöntemi, çağrı komutu yoğunluklu assembly kodları için güvenilir bir yöntem olsa da, çağrı komutlarının azaldığı durumlarda yanlış sonuçlar verebilmektedir. Bu olumsuzluğun önlenmesi ve dengelenmesi için bu çalışmada opcode benzerliği yöntemi kullanılmış ve iki yöntem arasında her ikisinden de yararlanacak şekilde dengeli bir ölçüm yapılmıştır.

Tablo 3.21’de Mwor virüsünün metamorfoz geçirmiş fonksiyon sayısı artırılmış örneği verilmiştir. Bu örnekte “call” çağrılarının fazla olması sebebiyle vertex ölçümü doğruluk payı yükselirken, opcode benzerlik ölçümü için çok fazla metamorfoz geçirmeye ve kendini %50’nin üstünde değiştirmeye müsait fonksiyonların ölçümünde doğruluk payının düştüğü gözlemlenmiştir. Hem Tablo 3.19’da hem de Tablo 3.20’de verilen örneklerde her iki benzerlik ölçüm yöntemi de aynı metamorfik virüsün iki farklı varyantı için %50’nin üzerinde benzerlik tespit etmiştir. Ancak, fazlaca metamorfoz geçirmeye müsait olunması veya “call” çağrı komutu barındırmaması gibi kısıtlarda oluşabilecek dezavantajı ortadan kaldırmak için benzerlik oranlarının belirli bir seviyede birleştirilmesi gerekmektedir. Bu oranları belirlemek için α ve β katsayıları kullanılır.

Tablo 3.21: Mwor virüsü metamorfoz geçiren fonksiyon sayısı artırılmış örneği

İterasyon	Vertex Benzerlik Oranı	Bölüt Değerleri	Opcode Benzerlik oranları(%)	Jaya Çözüm Değeri
i1	%80.0	{6,18,9,12,15,11,3,26,26,13}	[65.13, 65.13, 65.13, 65.13, 65.13, 65.13, 65.45, 65.78, 65.78, 65.13]	%65.15
i2	%80.0	{20,6,20,16,16,18,8,3,26,10}	[65.13, 65.13, 65.13, 65.13, 65.13, 65.13, 65.13, 65.45, 65.78, 65.13]	%65.20
i3	%80.0	{19,2,1,0,28,6,1,9,0,7}	[65.13, 62.42, 59.93, 57.41, 65.78, 65.13, 59.93, 65.13, 57.41, 69.46]	%65.51
i4	%80.0	{27,22,14,24,3,12,26,5,20,12}	[65.78, 65.13, 65.13, 69.46, 65.45, 65.13, 65.78, 62.42, 65.13, 65.13]	%65.83
i5	%80.0	{21,5,14,5,13,23,25,25,14,17}	[65.13, 62.42, 65.13, 62.42, 65.13, 69.46, 69.46, 69.46, 65.13, 65.13]	%66.40
i6	%80.0	{3,7,5,2,2,18,19,21,13,15}	[65.45, 69.46, 62.42, 62.42, 62.42, 65.13, 65.13, 65.13, 65.13, 65.13]	%66.75
i7	%80.0	{7,29,10,29,3,11,18,3,2,11}	[69.46, 65.78, 65.13, 65.78, 65.45, 65.13, 65.13, 65.45, 62.42, 65.13]	%67.15
i8	%80.0	{4,11,26,6,23,12,26,4,26,10}	[62.42, 65.13, 65.78, 65.13, 69.46, 65.13, 65.78, 62.42, 65.78, 65.13]	%67.60
i9	%80.0	{28,22,5,12,17,8,14,3,23,27}	[65.78, 65.13, 62.42, 65.13, 65.13, 65.13, 65.13, 65.45, 69.46, 65.78]	%67.89
i10	%80.0	{19,6,19,22,3,15,16,3,2,3,}	[65.13, 65.13, 65.13, 65.13, 65.45, 65.13, 65.13, 65.45, 62.42, 65.45]	%68.11



Şekil 3.6: Opcode örnekleri için nihai sonuç grafiği

Şekil 3.6, sırasıyla Tablo 3.20 ve 3.21’de verilen opcode örnekleri için hazırlanmış nihai sonuç grafiğidir. Grafik opcode benzerliği değeri oranının sıfırdan yüze kadar değişiminin nihai benzerlik değerine etkisini göstermektedir. Opcode örneği 1, metamorfoz geçirmiş fonksiyon sayısı artırılmış örneği ifade etmektedir. Bu örnek için opcode benzerlik değerinin yüzdesinin artması nihai sonucu azaltmakta ve yanlış pozitif oranını artırmaktadır. Metamorfik virüslerin büyük çoğunluğunu ölçse dahi opcode benzerlik yöntemi yüksek yapay zeka kullanılan virüsler için vertex benzerliğine ihtiyaç duymaktadır. Opcode örneği 2 ise fonksiyonlar içerisinde “call” çağrı komutu artırılmış örneği ifade etmektedir. Ve bu örnek için opcode benzerlik oranının artması nihai benzerlik oranını artırır ve daha doğru bir sonuç vermesine yardımcı olur. “call” çağrı komutunun olmadığı fonksiyonlar için vertex benzerliği ölçümü mümkün olmamaktadır. Bu noktada opcode benzerliği yöntemi kurtarıcı olmaktadır. Bu iki örnekten yola çıkarak, opcode ve vertex benzerlik oranlarının belirlenmesinde maksimum değer kullanılması uygun görülmüştür.

3.6 Opcode ve Vertex Benzerliđi İin Katsayı Belirleme

Opcode benzerliđi ve vertex benzerliđi deđerleri lldkten sonra nihai benzerlik deđerini (tespit yzde deđerini) iin bu iki deđerden daha yksek olanı nihai sonu olarak kabul edilir. α katsayısı opcode ile ve vertex benzerliđi β katsayısı ile arpılıp toplanır. α ve β katsayıları 0 veya 1 deđerini alırlar.

Vertex benzerliđi ile metamorfik ktcl kod varyantı tespit etmek, ađrı graflarının yođun olmadığı fonksiyonlar iin dezavantajlı bir yntemdir. Diđer yandan opcode benzerlik lm de ok fazla metamorfoz geiren fonksiyonlar iin bir dezavantaj kabul edilebilir. Bu yntemlerin problem zm iin birlikte kullanılıyor olması zmn dođruluđunu artırmıřtır. Bu alıřmada incelenen Opcode benzerliđi yntemi byk lde dođru bir hesaplama yapmasına karřın istisnai durumlarda vertex benzerliđi lmne ihtiya duymaktadır. Bu sebeple her ikisi iin de lm yapmak ve daha yksek olan benzerlik deđerini nihai sonu olarak deđerlendirmek dođru olacaktır. Tablo 3.22’te sırasıyla Tablo 3.19, 3.20 ve 3.21’deki rnekler verilmiřtir. Opcode benzerlik deđerini ve vertex benzerlik deđerini hesaplanan rneklerde maksimum deđer nihai sonu olarak alınmıřtır.

Tablo 3.22: Nihai sonu lm tablosu

	Opcode Benzerlik Deđerini	Vertex Benzerlik Deđerini	Nihai Sonu Deđerini
Opcode rneđi 1	%88.06	%80.0	%88.06
Opcode rneđi 2	%87,75	%60.0	%87.75
Opcode rneđi 3	%68.11	%80.0	%80.0

4. SONUÇLAR VE ÖNERİLER

Bilimin tanım ve özellikleri arasında sürekli gelişme halinde olma özelliği mevcuttur. Buradan hareketle bilgisayar dünyasının en önemli problemlerinden olan kötücül kodların ekonomiye, zamana, bilim dünyasına, insan emeğine olan zararlarını önlemek veya azaltmak adına çalışılan bu konunun ve yapmaya çalışılan uygulamanın sonucunda en tehlikeli virüslerden olan metamorfik virüsleri tespit edebilen bir yazılımı oluşturma ve çalışır hale getirme başarılmıştır.

Bu konunun seçilmesindeki amaç, metamorfik virüslerin klasik imza tabanlı yöntemlerle çözülememesinin yarattığı problemlerinin, sezgisel yöntemler ise çözülebilmesine olanak sağlayarak yazılımları oluşturmaktı ve sonuç olarak bu başarılmış durumdadır.

Opcode benzerlik yöntemi, benzerliği ölçmek için sadece komut sırasına baktığına için “fonksiyon sırasını değiştirme” ve “değişken isimlerini değiştirme” gibi metamorfik virüslerin gizlenme tekniklerinden etkilenmez. Aynı zamanda “işlevsiz komut ekleme” tekniğinin matematiksel işlemler içeren bir çok versiyonu için avantaj sağlamaktadır. Bu yöntem ile vertex benzerliği ölçümü için bir kısıtlama olan çağrı komutu azaltılmış örnekler, opcode benzerliği yöntemi ile yüksek oranda başarımlar göstererek ölçülebilmektedir. Bu yazılımın geliştirilerek diğer işletim sistemi platformlarında da çalıştırılabilir olması ayrıca farklı kötücül kodlar bakımından kapsamının geliştirilmesi, arka planda ve web üzerinde çalışabilir olması bu çalışma temel alınarak yapılacak ilerideki çalışmalardandır.

Bu alanda yeterli kaynak bulunmaması da zorlanılan durumlardan olmuştur. Kabul edilmelidir ki yeni bir alanda çalışabilir bir uygulama oluşturmak uzun emek ve çalışma gerektirmektedir. Tüm bilimsel çalışmalarda olduğu gibi bu bilimsel çalışma da bir sonuç değil, bu alandaki süreç noktalarından birisidir. Bilim dünyasına eklenecek küçük bir katkı, bu katkıyı sağlayanlar için önemli bir motivasyon kaynağı olacaktır.

5. KAYNAKLAR

- Aycock, J. "Computer Viruses and Malware" *Advances in Information Security*. New York: Springer-Verlag (2006).
- Baliga, A., Kamat, P., Iftode, L., "Lurking in the shadows: Identifying systemic threats to kernel data", *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 246-251, (2007).
- Bayođlu, B., Sođukpınar, İ., "Graph based signature classes for detecting polymorphic worms via content analysis", *Computer Networks*, 56(2), 832-844, (2012).
- Bazrafshan , Z., "A Survey on Heuristic Malware Detection Techniques", *5th Conference on Information and Knowledge Technology (IKT)*, 113-120, (2013)
- Bickford, J., et al. "Rootkits on smart phones: attacks, implications and opportunities." *Proceedings of the eleventh workshop on mobile computing systems & applications*. ACM, 49-54, (2010).
- Boldt, M. and B. Carlsson, "Privacy-Invasive Software and Preventive Mechanisms", *Systems and Networks Communications*, 2006. ICSNC'06. International 60 Conference on, pp. 21-21, IEEE, (2006).
- Canbek, G., Sađırođlu, Ő., "Kötücül ve Casus Yazılımlara Karşı Elektronik İmzanın Sađlamıő Olduđu Korunma Düzeyi", *uluslararası katılımlı bilgi güvenliđi ve kriptoloji konferansı*, 263-269, (2007).
- Carrera E., Erdelyi G., "Digital genome mapping-advanced binary malware analysis", *In: Proceeding Virus Bulletin Conference*, 187–197, (2004)
- Chouchane, M, R., and Lakhotia, A., "Using engine signature to detect metamorphic malware", *In WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode*, 73-78, New York, NY, USA, (2006).
- Christodorescu M., Jha, S., "Testing malware detectors" , *In ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 34–44, (2004).
- Cohen, F., "Computer viruses", *Computers & security* 6.1, 22-35, (1987).
- Çarkacı, N., and Sođukpınar, İ. "Frequency based metamorphic malware detection.", *Signal Processing and Communication Application Conference (SIU)*, 2016 24th. IEEE, 421-424, (2016).

- Davarcı, E., “Malware Analysis With Side-Channel Information On Android Smartphones”, B.S., *Electrical Electronics Engineering*, Boğaziçi University, (2012).
- Deshpande, P., “Metamorphic Detection Using Function Call Graph Analysis”, Master's Thesis, *San Jose State University*, (2013).
- Dezfouli F, N., Dehghantanha A., Mahmood R., N. F. B. M. Sani, S. B. Shamsuddin, and F. Daryabar, “A Survey on Malware Analysis and Detection Techniques”, *International Journal of Advancements in Computing Technology*, 42, (2013).
- Gordon, Jason. "Lessons from virus developers: The Beagle worm history through" SecurityFocus Guest Feature Forum, (2004).
- Griffin, K., Schneider, S., Hu, X. and Chiueh, T. “Automatic generation of string signatures for malware detection”, *Proceedings of the 12th International Symposium, RAID*, Sep 23- 25, 129, (2009).
- Gutmann P., “The Commercial Malware Industry.”, *DEFCON conference*, (2007).
- Govindaraju, A., "Exhaustive statistical analysis for detection of metamorphic malware.", Master's Thesis, *San Jose State University*, San Jose, (2010).
- Jacob G., Debar H., Filiol E., “Behavioral detection of malware: from a survey towards an established taxonomy,” *Journal in computer Virology*, 251–266, (2008).
- Kaushal, Kevadia, Prashant Swadas, and Nilesh Prajapati. "Metamorphic malware detection using statistical analysis." *International Journal of Soft Computing and Engineering (IJSCE)* 2.3, 49-53 (2012)
- Kienzle, D, M., and Matthew C, E., "Recent worms: a survey and trends." *Proceedings of the 2003 ACM Workshop on Rapid malcode*. ACM, 1-10, (2003).
- Kirda, E., et al. "Behavior-based Spyware Detection.", *Usenix Security Symposium*, 694, (2006).
- Konstantinou, E., “Metamorphic Virus: Analysis and Detection”, *Department of Mathematics, Royal Holloway, University of London*, 15, (2008).
- Lakhotia, A., Kapoor, A., Kumar E, U., “Are metamorphic computer viruses really invisible?” part 1. *Virus Bulletin*, 5-7, (2004).
- Leder, F., Steinbock, B., & Martini, P., "Classification and detection of metamorphic malware using value set analysis.", *Malicious and Unwanted Software (MALWARE)*, 2009 4th International Conference on. *IEEE*, 39-46, (2009).

- Lenny, Z., "Malware: Fighting Malicious Code [M]", *Indiana: Prentice Hall*, 34-45, (2003).
- Ling Y., "Short Review on Metamorphic Malware Detection in Hidden Markov Models", *international Journal of Advanced Research in Computer Science and Software Engineering*, 62-69, (2017).
- Ming, X., Lingfei W., Shuhui Q., Jian X., Haiping Z., Yizhi R., Ning Z., "A similarity metric method of obfuscated malware using function-call graph", *Journal in Computer Virology and Hacking Techniques*, 9(1):35-47, (2013)
- Rad, B. B., Masrom, M. And Ibrahim, S. "Camouflage in malware: from encryption to metamorphism." *International Journal of Computer Science and Network Security*, 12.8, 74-83, (2012).
- Nachenberg, C., "Computer virus-antivirus coevolution", *Communications of the ACM*, 40(1): 46–51, (1997)
- Nair, Vinod P., et al. "MEDUSA: Metamorphic malware dynamic analysis usingsignature from API.", *Proceedings of the 3rd International Conference on Security of Information and Networks.*, ACM, 263-269, (2010).
- Nwokedi, I., Mathur, A, P., "A survey of malware detection techniques.", *Purdue University* 48, (2007).
- Pandey, H, M., "Jaya a novel optimization algorithm: what, How and why?", 2016 6th International Conference - Cloud System and Big Data Engineering, 728-730, (2016).
- Pektaş, A., "Behavior based malicious software detection and classification", Master's Thesis, *Galatasaray University, İstanbul*, 15-26, (2012).
- Rao, R,V., Waghmare, G, G., "A new optimization algorithm for solving complex constrained design optimization problems." *Engineering Optimization*, 60-83, (2017).
- Rao, R. "Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems." *International Journal of Industrial Engineering Computations*, 19-34, (2016).
- Rao, R.V. "Teaching Learning Based Optimization And Its Engineering Applications", *Springer Verlag, London*, (2015).
- Rao, R.V., Patel, V. "Comparative performance of an elitist teaching-learning-based optimization algorithm for solving unconstrained optimization problems", *International Journal of Industrial Engineering Computations*, 29-50, (2013).

- Saroiu, S., Gribble S, D., Levy H, M., "*Measurement and Analysis of Spyware in a University Environment*", *Measurement and Analysis of Spyware in a University Environment*. In NSDI 141-153, (2004).
- Shang S.,Zhen N., Xu J., Xu M. and Zhang H., "Detecting malware variants via function-call graph similarity", *5th International Conference Malicious and Unwanted Software*, 113–120, (2010)
- Sharma, A., Sahay K., "Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey", *IJCA Journal*, (2014)
- Szor P., "The Art of Computer: Virus Research and Defense". *Symantec Press, NJ, USA, first edition*, (2005).
- Szor, P., and Ferrie P., "Hunting for metamorphic." *Virus bulletin conference*. (2001).
- Szor, P., "Metamorphic computer virus detection." U.S. Patent No. 7,937,764. (2011).
- Toderici, A. H., Stamp, M., "Chi-squared distance and metamorphic virus detection", *Journal of Computer Virology and Hacking Techniques*, 1-14, (2014).
- Tran, N. and Lee, M. "High performance string matching for security applications", *roceedings of the International Conference on ICT for Smart Society*, 1-5, (2013).
- Vinod, P., et al. "Survey on malware detection methods", *Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09)*, 74-79, (2009).
- Walenstein, Andrew, et al. "Normalizing metamorphic malware using term rewriting.", *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on. IEEE*, 75-84, (2006).
- Wong, W., "Analysis and detection of metamorphic computer viruses", *Department of Computer Science, Master's Thesis, San Jose State University*, (2006)
- Zhang Q.,Douglas S, R., "Metaaware: Identifying metamorphic malware." *Computer Security Applications Conference, ACSAC 2007. IEEE*, 411-420, (2007).
- Zhihong Z., Qing-xin Z., Zhou M., "On the time complexity of computer viruses", *IEEE Transactions on Information Theory*, 51(8): 2962-2966, (2005).
- Web 1: <https://www.av-test.org>, (2018).
- Web2: KALPA, "Introduction to Malware", http://securityresearch.in/index.php/projects/malware_lab/introduction-to-malware/8/, (2011).

EKLER

6. EKLER

EK A: Opcode Dizisi Alt Bölütlemesi

Tablo A.23: Opcode Dizisi Alt Bölütlemesi

	Bölüt İşareti						
	0	1	2	3	4	5	6
Opcode	Mov	Push	İn	Lea	Sti	Add	Or
	Movdqu	Pop	Out	Lds	Cli	Sub	And
	Pshufd	Pusha		Les	Std	İnc	Por
	Cvti2sd	Popa			Cld	Cwd	Pandn
	Ldmscr				Stc	Psrlq	Pand
	Stmscr				Cle	Cmp	Pxor
	Cmovns				Lahf	Mul	Setnl
	Movapd				Ucomisd	Pcmptd	Sets
	Cvtss2s				Ucomiss	Pcmpeqd	Setp
	Cvti2ss				Cmc	Paddsb	Setnb
	Movdqa					Divss	Xorpd
	Movss					Addss	Xorps
	Cmovbe					Mulss	Setle
	Movaps					Divsd	Setnbe
	Movsxd					Subss	Setz
	Cmovge					Dec	Setbe
	Movd					Mulsd	Cdq
	Cmovnb					Addsd	Setb
	Cmovle					Shld	Setnz
	Cmova					Ror	Xor
	Cmovb					Cdq	Not
	Cmovl					Cwde	Test
	Cmovs					Shrd	Setl
	Movzx					Rol	Setnle
	Cmovg					Shl	
	Cmovz					İdiv	
	Cmovnz					Sbb	
						Shr	
						Div	
						Neg	
						Rcr	
						Adc	
					Rcl		
					Sar		

Tablo A.24: Opcode Dizisi Alt Bölütlemesi (Devamı)

		Bölüt İşareti							
		7	8	9	10	11	12	13	14
Opcode	Movsb	Jmp	Jz	Loop	Hlt	Bsf	Proc	Fld	
	Xchg	Retf	Jnp	Repnz		Bts	Cpuid	Cvtt2sd	
	Lodsd	Leave	Jp	Repz		Btr	Icebp	Subsd	
	Lodsw	Iret	Jns	Repne		Btc	Enter	Fcos	
	Lodsb	Ret	Js	Repe		Bt	Nop	Cvtpd2ps	
	Stosd	Retn	Jbe	Rep		Bsr		Unpcklpd	
	Stosw	Call	Jnb	Loopz				Cvtps2pd	
	Stosd		Ja	Loopnz				Fisttp	
	Stosb		Jle	Loopne				Unpcklps	
	Scasd		Jnz	Loope				Fild	
	Scasb		Jb	Loopx				Fsubp	
	Scasw		Jg					Fmul	
	Cmpsw		Jge					Fldcw	
	Cmpsd		Jl					Fist	
	Cmpsb		Jne					Fucomip	
	Movsx		Je					Fnstcw	
	Movsd							Fldl	
	Movsw							Fmulp	
								Fxch	
								Fadd	
								Fucomi	
								Fldz	
								Fninit	
								Fstps	
							Fst		

EK B : 4 ile Bölütlenmiş Opcode Alt Bölütleme Örneği

Tablo B.25: 4 ile Bölütlenmiş Opcode Alt Bölütleme Örneği

Opcode	Bölüt İşareti									
	0	1	2	3	4	5	6			
0	Mov	7 Push	8 In	9 Lea	10 Sti	13 Add	22 Or			
	Movdqu							Out	Lds	Cli
	Pshufd		Pusha	Les				Std	Inc	Por
	Cvti2sd									
1	Ldmscr				11 Stc	14 Psrdq	23 Pand			
	Stmscr							Cmp	Pxor	
	Cmovns							Lahf	Mul	Setnl
	Movapd							Ucomisd	Pcmptd	Sets
2	Cvttss2s				12 Ucomiss	15 Pcmpeqd	24 Setp			
	Cvti2ss							Cmc	Paddsb	Setnb
	Movdqa				16 Mulss	Divss	Xorpd			
	Movss							Addss	Xorps	
3	Cmovbe				17 Mulss	16 Divsd	25 Setle			
	Movaps							Subss	Setnbe	
	Movsxd							Dec	Setz	
	Cmovge								Setbe	
4	Movd				17 Mulsd	17 Addsd	26 Cdqe			
	Cmovnb							Shld	Setb	
	Cmovle							Ror	Setnz	
	Cmova								Xor	
5	Cmovb				18 Cdq	18 Cwde	27 Not			
	Cmovl							Shrd	Test	
	Cmovs							Rol	Setl	
	Movzx								Setnle	
6	Cmovg				19 Shl	19 Idiv				
	Cmovz							Sbb		
	Cmovnz							20 Div	Neg	
					21 Rcl					
							Sar			

Tablo B.26: 4 ile Bölütlenmiş Opcode Alt Bölütleme Örneği (Devamı)

		Bölüt İşareti														
		7	8	9	10	11	12	13	14							
Opcode	28	Movsb	33	Jmp	35	Jz	39	Loop	42	Hlt	43	Bsf	45	Proc	47	Fld
		Xchg		Retf		Jnp		Repnz		Bts		Cpuid		Cvttsd2s		
		Lodsd		Leave		Jp		Repz		Btr		İcebpb		Subsd		
		Lodsw		İret		Jns		Repne		Btc		Enter		Fcos		
	29	Lodsb	34	Ret	36	Jc	40	Repe	44	Bt	46	Nop	48	Cvtpd2ps		
		Stosd		Retn		Jbe		Rep		Bsr		Unpcklpd				
		Stosw		Call		Jnb		Loopz				Cvtps2pd				
		Stosd				Jnc		Loopnz				Fisttp				
	30	Stosb	37	Jle	41	Jle	41	Loopne					49	Unpcklpd		
		Scasd		Jnz		Jnz		Loope						Fild		
		Scasb		Jb		Jb		Loopx						Fsubp		
		Scasw		Jg		Jg								Fmul		
	31	Cmpsw	38	Jge		Jge							50	Fldcw		
		Cmpsd		Jl		Jl								Fist		
		Cmpsb		Jne		Jne								Fucomip		
		Movsx		Je		Je								Fnstcw		
32	Movsd											51	Fldl			
	Movsw						Fmulp									
													52	Fxch		
														Fadd		
														Fucomi		
														Fldz		
														Fninit		
														Fstp		
														53	Fst	

EK C : Türetilmiş Kısa Kod Üzerinden Opcode Benzerliği Yönteminin İncelenmesi

Bu ekte idapro64 (versiyon 7.0) programının çalıştırılabilir (.exe) dosyasından çıkarılmış olan assembly kodlarından iki farklı versiyon oluşturularak program çıktı değerleri verilecektir. Program kodlarının çok uzun olması sebebiyle fonksiyonlar kısaltılmış ve örnek oluşturması amacıyla düzenlenmiştir.

Tablo C.27’de orijinal, metamorfoz geçirmemiş olan kodlardan oluşan 3 farklı fonksiyon vardır. Aynı tabloda verilen metamorfoz ile türetilmiş kodlarda ise fonksiyon sırasının değiştirilmesi, değişken isimlerinin değiştirilmesi, komut sıralarının değiştirilmesi, fonksiyon isimlerinin değiştirilmesi ve geçersiz kod ekleme teknikleri ile şaşırtma yöntemleri uygulanarak örnek bir kod parçası türetilmiştir.

Bu ekte yöntemin, basit bir örnekle gösterilmesi ve geçerliliğinin, doğrulanabilirliğinin sınanabilmesi amaçlanmıştır.

Tablo C.27: Orijinal ve Türetilmiş Kod Karşılaştırma Örneği

Orijinal Kod	Metamorfoz İle Türetilmiş Kod
<pre> sub_140191740 proc near mov edx, 0Ah lea rcx, aAddons mov [rsp+68h], rax mov rcx, [rsi+38h] call cs:?addButton mov r8d, 3 lea rdx, [rsp+68h] mov rcx, [rcx+68h] mov [rsi+40h], rax lea rcx, [rsp+68h] mov dword ptr [rsp+28h], 0 lea rax, a1infoclicked mov [rsp+20h], rax mov r9, rsi call qword lea r8, a2clicked mov rdx, [rsi+40h] lea rcx, [rsp+68h] lea rcx, [rsp+68h] mov rax, [rsi+38h] mov rcx, [rax+48h] sub_140191740 endp sub_1400D4680 proc near mov [rsp+8], rbx mov [rsp+10h], rbp </pre>	<pre> sub_140111010 proc near push rpi sub rsp, 40h add rsp, 0 mov rpi, rdx mov rdx, [rsp+70h] test rdx, rdx jnz short loc_14012104A mov rcx, [rsi] lea rdx, aNew mov rcx, [rsi] lea rdx, aAddStandardEnu add rsp, 40h add rcx, 48h pop rpi nop jmp sub_140012FF0 mov [rsp+50h], rbx mov [rsp+58h], rdi lea rdi, [rcx+0B8h] mov rax, [rdi] mov rcx, 0FF0000000000000h mov [rsp+60h], r14 sub_140111010 endp sub_140191740 proc near mov edx, 0Ah </pre>

<pre> mov [rsp+18h], rsi mov [rsp+20h], rdi push r14 sub rsp, 20h mov rax, [rcx+8] mov rbx, rcx xor ecx, ecx movzx ebp, r8b test rax, rax mov r14, rdx lea rdi, [rax-1] cmovz rdi, rcx lea rsi, [rdi+2] cmp rax, rsi jnb short loc_1400D46F4 cmp rsi, [rbx+10h] jbe short loc_1400D46E0 mov rdx, [rbx] call sub_140191740 lea r9d, [rcx+1] mov rcx, rbx sub_1400D4680 endp sub_140121010 proc near push rsi sub rsp, 40h mov rsi, rdx mov rdx, [rsp+70h] test rdx, rdx jnz short loc_14012104A mov rcx, [rsi] lea rdx, aNew mov rcx, [rsi] lea rdx, aAddStandardEnu add rcx, 48h add rsp, 40h pop rsi jmp sub_140012FF0 mov [rsp+50h], rbx mov [rsp+58h], rdi lea rdi, [rcx+0B8h] mov rax, [rdi] mov rcx, 0FF0000000000000h mov [rsp+60h], r14 sub_140121010 endp </pre>	<pre> lea rcx, aAddons mov [rsp+68h], rax mov rcx, [rsi+38h] call cs:?addButton nop mov r8d, 3 lea rdx, [rsp+68h] mov rcx, [rcx+68h] mov [rsi+40h], rax lea rcx, [rsp+68h] mov dword ptr [rsp+28h], 0 lea rax, a1infoclicked mov [rsp+20h], rax mov r9, rsi call qword lea r8, a2clicked mov rdx, [rsi+40h] lea rcx, [rsp+68h] lea rcx, [rsp+68h] mov rax, [rsi+38h] mov rcx, [rax+48h] sub_140191740 endp sub_1400D4680 proc near mov [edx+8], rbx mov [edx+10h], rbp mov [edx+18h], rsi mov [edx+20h], rdi push r14 sub edx, 20h mov rax, [rcx+8] mov rbx, rcx xor ecx, ecx movzx ebp, r8b test rax, rax mov r14, rdx lea rdi, [rax-1] cmovz rdi, rcx lea rsi, [rdi+2] cmp rax, rsi jnb short loc_1400D46F4 cmp rsi, [rbx+10h] jbe short loc_1400D46E0 mov rdx, [rbx] call sub_140191740 lea r9d, [rcx+1] nop mov rcx, rbx sub_1400D4680 endp </pre>
--	--

Tablo C.28: Türetilmiş kod için program çıktısı tablosu

İterasyon	Bölüt Değerleri	Opcode Benzerlik oranları(%)	Jaya Çözüm Değeri
İ1	{29,23,5,7,18,13,8,17,2,18}	[78.95, 79.9, 81.08, 80.82, 79.9, 79.9, 78.65, 79.9, 81.29, 79.9]	%82.35
İ2	{29,29,11,10,16,13,1,11,28}	[78.95, 78.95, 78.65, 78.65, 79.9, 79.9, 82.44, 78.65, 78.95, 79.9]	%82.56
İ3	{22,0,3,21,18,21,18,27,19,4}	[79.9, 78.99, 81.08, 79.9, 79.9, 79.9, 79.9, 78.95, 79.9, 81.08]	%82.62
İ4	{7,21,8,27,11,20,26,10,18,22}	[80.82, 79.9, 78.65, 78.95, 78.65, 79.9, 78.95, 78.65, 79.9, 79.9]	%82.64
İ5	{3,0,21,24,15,18,4,23,28,5}	[81.08, 78.99, 79.9, 79.9, 79.9, 79.9, 81.08, 79.9, 78.95, 81.08]	%82.72
İ6	{1,14,22,1,27,25,23,21,16,22}	[82.44, 79.9, 79.9, 82.44, 78.95, 79.9, 79.9, 79.9, 79.9, 79.9]	%82.81
İ7	{17,12,29,0,26,1,20,15,9,17}	[79.9, 78.65, 78.95, 78.99, 78.95, 82.44, 79.9, 79.9, 78.65, 79.9]	%83.11
İ8	{10,16,27,2,2,19,18,12,17,9}	[78.65, 79.9, 78.95, 81.29, 81.29, 79.9, 79.9, 78.65, 79.9, 78.65]	%83.47
İ9	{29,21,11,27,29,12,17,4,28,11}	[78.95, 79.9, 78.65, 78.95, 78.95, 78.65, 79.9, 81.08, 78.95, 78.65]	%83.55
İ10	{2,0,2,18,6,23,18,3,13,15}	[81.29, 78.99, 81.29, 79.9, 80.82, 79.9, 79.9, 81.08, 79.9, 79.9]	%83.90

ÖZGEÇMİŞ

Adı Soyadı : KÜBRA NUR ATASEVER

Doğum Yeri ve Tarihi : EMET / 30.07.1992

Lisans Üniversite : DUMLUPINAR ÜNİVERSİTESİ

Elektronik posta : knatasever@gmail.com

İletişim Adresi : Durak mah. Hoca Ahmet Yesevi cad. 13/4
Merkez/UŞAK