ANDROID AUTHORSHIP ATTRIBUTION USING SOURCE CODE-BASED FEATURES

KAYNAK KODU TABANLI ÖZELLİKLER KULLANILARAK ANDROID YAZARLIK İLIŞKİLENDİRMESİ

EMRE AYDOĞAN

PROF. DR. SEVIL ŞEN

Supervisor

Submitted to

Graduate School of Science and Engineering of Hacettepe University

as a Partial Fulfillment to the Requirements

for the Award of the Degree of Doctor of Philosophy

in Computer Engineering

January 2024

ABSTRACT

ANDROID AUTHORSHIP ATTRIBUTION USING SOURCE CODE-BASED FEATURES

Emre Aydoğan

Doctor of Philosophy , Computer Engineering

Supervisor: Prof. Dr. Sevil ŞEN

January 2024, 154 pages

With the widespread use of mobile devices, Android has become the most popular operating system, and new applications are uploaded to the Android market every day. However, the simplicity of modifying and repackaging Android binaries enables other developers to easily alter and mimic Android applications, subsequently releasing them on third-party Android markets. Determining the original developers of Android applications is a challenging problem known as authorship attribution. This study explored the distinctive features of Android applications to identify their authors. Software developers generally leave a footprint that describes their writing styles on their applications. This footprint, which can be extracted from either the source code or binary code, can help identify the authors of software applications. Due to the impracticality of accessing the source code for applications found in real-world scenarios, particularly in the case of malware, researchers concentrate their efforts on analyzing the binaries of these applications. Therefore, this study proposes an approach that identifies Android developers by deriving a wide range of features from different parts of Android applications, such as smali files, libraries, manifest files, and metadata information. Moreover, other features such as configuration, dex code, resource-based,

and string-related features are inherited from other studies in Android authorship attribution and fused with the proposed feature set. The proposed approach was evaluated on benign and malware datasets and compared with those of other studies. The results show that the proposed features increased the accuracy by showing 82.5% and 95.6% in the market and malware datasets, respectively. The results demonstrate the positive effect of the proposed features on Android authorship attribution.

Keywords: Authorship Attribution, Mobile Malware, Source Code-Based, Android, Obfuscation, Metadata

ÖZET

KAYNAK KODU TABANLI ÖZELLİKLER KULLANILARAK ANDROID YAZARLIK İLİŞKİLENDİRMESİ

Emre Aydoğan

Doktora, Bilgisayar Mühendisliği

Danışman: Prof. Dr. Sevil ŞEN

Ocak 2024, 154 sayfa

Mobil cihazların yaygınlaşmasıyla birlikte Android, en popüler işletim sistemlerinden biri haline gelmiştir. Her gün Android platformuna yeni uygulamalar eklenmektedir. Ancak, Android uygulamalarının ikili dosyalarını kolayca değiştirip yeniden paketleyebilme imkanı nedeniyle, bu uygulamalar diğer geliştiriciler tarafından rahatça modifiye edilebilir ve taklit edilebilir hale gelmiştir. Bu durum, uygulamaların orijinal geliştiricilerini belirlemeyi, başka bir deyişle yazarlık ilişkilendirmesini oldukça zor bir sorun haline getirmiştir. Bu çalışma, Android uygulamalarının özgün karakteristik özelliklerini tespit etmeyi amaçlamaktadır. Geliştiriciler, genellikle uygulamalarında kod yazım stillerini tanımlayan izler bırakırlar. Bu izler, kaynak kodu veya ikili kod üzerinden tespit edilebilir. Bu durum, uygulamalarının gelitiricilerinin belirlenmesinde önemli bir yardımcı olabilir. Kaynak kodunun elde edilmesi, özellikle kötü amaçlı yazılımlar söz konusu olduğunda, her zaman mümkün olmayabilir. Bu yüzden araştırmacılar, uygulamaların ikili dosyaları üzerine odaklanmayı tercih etmişlerdir. Bu çalışma, Android uygulamalarının çeşitli bölümlerinden - örneğin smali dosyalar, kütüphaneler, manifest dosyaları ve meta veri bilgileri - geniş bir

öznitelik yelpazesi çıkarmayı ve bu öznitelikler üzerinden Android geliştiricilerini tanımlamayı önermektedir. Diğer Android yazarlık ilişkilendirmesi çalışmalarından alınan yapılandırma, dex kodu, kaynak tabanlı ve dizgi ile ilgili öznitelikler de bu çalışmada önerilen öznitelik setine entegre edilmiştir. Önerilen yaklaşım zararlı olmayan ve kötü amaçlı veri kümeleri üzerinde test edilmiş ve diğer çalışmalarla karşılaştırılmıştır. Elde edilen sonuçlar, önerilen özniteliklerin doğruluk oranını artırdığını göstermiş; market ve kötü amaçlı yazılım veri kümelerinde sırasıyla %82,5 ve %95,6 oranında doğruluk sağlanmıştır. Bu sonuçlar, Android yazarlık ilişkilendirmesinde önerilen özniteliklerin olumlu etkisini ortaya koymaktadır.

Keywords: Yazarlık İlişkilendirmesi, Mobil Zararlı Yazılımlar, Kaynak Kod, Android, Karıştırma, Üstveri

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor, Prof. Dr. Sevil ŞEN, for her valuable advice, guidance, and encouragement. She patiently supported me with her knowledge and experiences throughout the preparation of this thesis and my doctorate education.

Besides, I would like to thank my thesis committee members, Prof. Dr. Ali Gökhan YAVUZ, Asst. Dr. Burcu CAN for their valuable advice and guidance through my thesis, and Prof. Dr. İlyas ÇİÇEKLİ and Asst. Dr. Engin DEMİR for reviewing this thesis and giving their insightful comments.

I wish to show my gratitude to the current and former administrative board as well as the academic personnel at the Department of Computer Engineering at Antalya Akdeniz University for letting me study at Hacettepe University.

I would like to thank my friends Dr. Selim YILMAZ, Dr. Levent KARACAN, Dr. Çağdaş BAŞ, Dr. Aysun KOÇAK, Dr. Cemil ZALLUHOĞLU, all the members of *Wireless Networks and Intelligent Secure Systems Laboratory* (WISE Lab.) of Hacettepe University, and all the academic personnel and research assistants I worked with at the Department of Computer Engineering at Hacettepe University.

I would like to thank my teammates at Karel, where I work, for supporting me.

I wish to express my huge gratitude to my mother and my father for their self-devotion, love, and giving me opportunities that have made me who I am and to my brother for their love and support that I have felt throughout my life.

I would like to pay my special regards to my beloved wife, Derya, for continually encouraging me during my Ph.D. and for her support that I feel every time I need it during my academic life.

GENİŞLETİLMİŞ ÖZET

GİRİŞ

Yazarlık ilişkilendirmesi (Yİ), bir bilgisayar programının geliştiricisini belirlemeyi amaçlar. Temelde yazılım hırsızlığını tespit etmek ve telif hakları sorunlarını çözmek için kullanılsa da, dijital adli tıp ve kötü amaçlı yazılım analizi gibi alanlarda da kullanılır. Yazılım geliştiriciler, genellikle uygulamalarına, kodlama stillerini tanımlayan izler bırakır. Bu nedenle, aynı geliştirici tarafından geliştirilen uygulamalar arasında güçlü bir korelasyon vardır [1]. Literatürde, bu izler ilk kez uygulama geliştiricilerini belirlemek için kullanılmıştır. Orijinal yazılımların modifikasyonları olan yazılım varyantlarının ortaya çıkışıyla, araştırmacılar otomatik olarak yazılım hırsızlığını ve telif hakkı sorunlarını belirleme konusuna odaklanmışlardır.

Yazarlık ilişkilendirmesinin önemli uygulamalarından biri, kötü amaçlı yazılım geliştiricilerini belirlemektir. Bu durum, yeni kötü amaçlı yazılım sayısında ve mevcut kötü amaçlı yazılımların yeni varyasyonlarında gözlemlenen önemli artıştan kaynaklanmaktadır [2]. Yİ, yeni kötü amaçlı yazılımların geliştiricilerini belirlemek için kullanılabilir. Geliştiricileri izleyerek kötü amaçlı yazılımların evrim sürecini gözlemlemeye yardımcı olabilir. Aynı şekilde, bilinen saldırganlar tarafından geliştirilen yeni programların kontrol edilmesi için de kullanılabilir. Yİ üzerine çok sayıda çalışma olmasına rağmen, özellikle Android kötü amaçlı yazılımları üzerindeki araştırmalar araştırmacılar tarafından yeterince derinlemesine incelenmemiştir [3–5].

Yazarlık ilişkilendirilmesinde *kaynak kod* ve *ikili kod* tabanlı olmak üzere iki ana yaklaşım bulunmaktadır. Başlangıçta araştırmacılar, bilinmeyen geliştiricileri belirlemek için yazılımların kaynak koduna odaklanmışlardır [6–10]. Bunun önemli sebebi kaynak koddan çıkarılan bazı belirgin özelliklerin, derleme tamamlandığında kaybolmasıdır. Kullandıkları teknikler, satır uzunluğu, ortalama prosedür uzunluğu, metod sayısı, isimlendirme kuralları, yorumlar, programlama düzeni özellikleri (boşluk, girintileme vb.), değişken isimleri, kontrol akış yapıları ve geliştirme ortamı (bilgisayar

platformu, programlama dili, derleyici) gibi birçok farklı özelliği içerir. Ancak, özellikle kötü amaçlı olanlar olmak üzere, yazılımların kaynak kodlarını bulmanın zorlukları vardır. Bu gibi durumlarda, bir uygulamanın geliştiricisini belirlemek için kullanılabilecek tek kaynak, uygulamanın ikili kodudur. Literatürde, yazılımların ikili kodu üzerinde çalışan araştırmacılar bulunmaktadır [1, 11, 12]. Ancak, ikili koda dayalı geliştiricileri belirlemek, kaynak koda dayalı geliştiricileri belirlemekten çok daha zordur. Derleme süreci sırasında, derleyiciler optimizasyon teknikleri aracılığıyla ikili kodun yapısını değiştirebilir ve bu da Yİ'yi etkileyebilir. Örneğin, Yİ açısından değerli bazı özellikleri, dilbilgisi ve biçimlendirme gibi, kaldırabilirler. Ayrıca, farklı derleyicilerin kullanımı nedeniyle, çoğu akademik çalışma belirli derleyicilere ve derleme ayarlarına bağlıdır [13]. Tüm bu zorluklara rağmen, kaynak kodunun bulunamaması nedeniyle kötü amaçlı yazılımların yazarlık ilişkilendirilmesi için ikili koda güvenilmek zorunda kalınmıştır. Üstelik, yazarlık ilişkilendirilmesinde kullanılan popüler tekniklerin, makine öğrenimi ve derin öğrenme gibi, performansı büyük ölçüde eğitim verisinin boyutuna bağlıdır. Uygulamaların ikilileri kolayca elde edilebildiğinden, ikili kod Yİ araştırılması gereken bir alan olarak değerlidir.

Bu tezin amacı, özellikle kötü amaçlı yazılımlar olmak üzere, belirli bir grup içerisindeki uygulamaların geliştiricilerini tanımlamaktır. Bu, bazı geliştiricilerin başkalarına ait uygulamaları kendi uygulamalarıymış gibi uygulama marketlerine yüklemeleri durumunda kritik önem taşır. Geliştiriciler, başkalarının çalışmalarını geri derleme yapıp değiştirerek bu durumu gerçekleştirirler. Bu süreç, kötü amaçlı yazılım alanında yaygın bir pratik olan yeniden paketleme olarak bilinir. Bu sebeplerle, bu çalışmada önerilen yöntem, belirli bir yazar tarafından yazılmış gibi görünen fakat aslında yazılmamış uygulamaların tespit edilmesine odaklanmıştır. Ek olarak, önerilen yöntem, ticari amaçlarla geliştirilen ancak daha sonra başka bir geliştirici tarafından yeniden paketlenip farklı bir isim altında piyasaya sürülen uygulamaların orijinal geliştiricisinin tespiti sağlayarak telif hakkı sorunlarını önleyebilir. Ayrıca, bu aynı süreç öğrenci ödevlerine de uygulanabilir ve öğrencilerin geliştirme sırasında kodu başka bir yerden alıp almadıklarını belirlemede yardımcı olur. Sonuç olarak, bu araştırma sadece yazılım

geliştirmede önemli bir zorluğu ele almakla kalmaz, aynı zamanda akademik dürüstlüğü koruma konusunda da yenilikçi bir yaklaşım sunar.

Kötü amaçlı yazılım geliştiricileriyle ilgili senaryo özellikle önemlidir. Bu geliştiriciler sıklıkla, uygulamalarının erken sürümlerini farklı takma adlar altında çeşitli platformlara, alternatif marketlere ve çevrimiçi analiz araçlarına yükleyerek dolaylı geri bildirim alırlar. Aldıkları bu geri bildirimleri kullanarak, antivirüs sistemlerini atlatma stratejileri geliştirirler. Bu çalışmada önerilen yöntem, bu kurnaz taktiği adresler ve herhangi bir ortama yüklenen bir uygulamanın analiz edilmesine olanak tanır. Önemli bir şekilde, önerilen yöntem, bir uygulama başlangıçta geleneksel tespit sistemleri tarafından kötü amaçlı olarak algılanmasa bile, onun bilinen bir kötü amaçlı yazılım geliştiricisine ait olduğunu belirleyebilir. Bu sayede, söz konusu uygulamalar potansiyel bir kötü amaçlı uygulama olarak işaretlenebilir. Bu proaktif yaklaşım, kötü amaçlı yazılım analistleri ve antivirüs sistemleri için önemli bir avantaj sağlar.

Ayrıca, önerilen yöntem kötü amaçlı yazılım araştırmaları için önemli bir avantaj sağlar. Farklı zamanlarda piyasaya sürülen aynı kötü amaçlı yazılımın sürümlerini kümeleyebilme yeteneği, kötü amaçlı yazılımların evrimini kapsamlı bir şekilde anlamak için kritik öneme sahiptir. Bu bilgiler, özellikle Android kötü amaçlı yazılım geliştirme evrimi alanında son derece yararlıdır.

Bu çalışma, Android uygulamalarının Yazarlık İlişkilendirmesi için ayırt edici özniteliklerini incelemektedir. Araştırma kapsamı, uygulamaların *smali* dosyaları, kütüphaneleri ve izinleri gibi bileşenlerinden ve alternatif marketlerdeki metadata bilgilerinden toplanan yeni öznitelikleri içermektedir. Bu yeni toplanan öznitelikler, daha önceki çalışmalarda kullanılan özniteliklerle [14–16] birlikte, kapsamlı bir analize tabi tutulmaktadır.

Bu araştırmada özellikle, farklı öznitelik gruplarının Yİ üzerindeki etkisini incelenmektedir. Mevcut literatürde önerilen özelliklerin farklı veri setleri kullanılarak analiz edilmiş olması göz önünde bulundurularak, adil bir karşılaştırma sağlamak amacıyla ortak bir deneysel çerçeve oluşturulmuştur. Kullanılan Veri seti, çeşitli

Android marketlerden ve çalışmalardan zararsız, kötü amaçlı ve karıştırılmış uygulamaları içermekte olup, Bölüm 5.2.'de detaylandırılmıştır.

Çalışmaya dahil edilen öznitelikler, önceki araştırmalardan çıkarılmış [14–16], yapılandırma, DEX (Dalvik Executable) kodu, kaynak ve dizgi tabanlı öğeleri kapsamaktadır. Her öznitelik kategorisi, APK dosyalarının belirli bileşenlerinden çıkarılmıştır:

- **Yapılandırma Öznitelikleri**: Bunlar APK dosyalarının manifest dosyasından elde edilmektedir. Manifest dosyası, uygulamanın kurulumu ve izinleri hakkında bilgiler sağlayarak, yapılandırma verileri için zengin bir kaynak oluşturur.
- **DEX Kod Öznitelikleri**: *dex* dosyasından doğrudan çıkarılan bu öznitelikler, uygulamanın yapısal öğelerine, örneğin metodlara, sınıflara ve alan yapılarına odaklanır. *dex* dosyası, uygulamanın kaynak kodunun derlenmiş versiyonudur.
- **Kaynak Öznitelikleri**: Bu öznitelikler, geri derleme yapılan APK dosyalarının "res" dizininden toplanır. Resimler ve yerleşim dosyaları gibi kaynakları içeren "res" dizini, uygulamanın görsel ve yapısal yönleri hakkında bilgiler sağlar.
- **Dizgi Tabanlı Öznitelikler**: Burada, hem *strings.xml* dosyasındaki hem de *dex* dosyasındaki dizgiler kullanılır. Bu dizgilere n-gram analizi uygulanarak, uygulamanın kodlama stili ve içeriğine özgü desenleri ve dizgileri ortaya çıkarılabilir.

KATKI

Bu araştırmada, Android Yİ için yenilikçi ve etkili bir yaklaşım önerilmektedir. Bu tezin ana katkıları aşağıdaki şekilde özetlenebilir:

- Kaynak kodu Yİ'den miras alınan öznitelikler *smali* dosyalarından elde edilmiştir. Bu öznitelikler bu çalışmada Android Yİ için ilk kez kullanılmıştır.

Sonuçlar, kaynak kodu bazlı özniteliklerin uygulamaları ilgili geliştiricilerine ilişkilendirmede doğruluğu artırdığını göstermiştir.

- Bu çalışmada, daha önceki araştırmalarda dizgi öznitelikleri olarak kullanılan TPL ve izin öznitelikleri, ikili öznitelikler olarak kullanılmıştır.

- Geleneksel uygulama dağıtım mekanizmalarının tersine, Android uygulamaları mobil marketler aracılığıyla merkezi bir şekilde dağıtılır. Bu durum, uygulamanın kendisinin yanı sıra, uygulama açıklamaları ve kullanıcı yorumları gibi Android uygulamaları hakkında zengin metadata elde edilmesini sağlar. Bu çalışmada, Android Yİ üzerinde metinsel metadatanın etkisi incelenmiştir. Özellikle, metadata özniteliklerinin, bu çalışmada kullanılan kaynak kodu bazlı özniteliklerle birleştirildiğinde, Yİ alanında olumlu bir etkisi olduğu gözlemlenmiştir. Bu çalışma, Android Yİ alanında metadata açıklamalarını kullanan ilk çalışmalardan biridir.

- Bu çalışmada önerilen yaklaşımın performansı kapsamlı bir şekilde analiz edilmiştir. Android Yİ üzerinde her öznitelik setinin etkisini analiz etmenin yanı sıra, literatürde önerilen çalışmalar karşılaştırılmıştır. Deneysel sonuçlar, bu çalışmada önerilen özniteliklerin yazarlık ilişkilendirmede olumlu etkisini göstermektedir.

- Bu çalışmada ayrıca, Yİ perspektifinden hem versiyon uygulamalar hem de karıştırılmış uygulamalar detaylı bir şekilde ele alınmıştır. Bu çalışma, versiyon uygulamalarının analizine odaklanan ilk çalışmadır.

- Zararsız, kötü amaçlı, versiyon ve karıştırılmış gibi farklı karakteristiklere sahip çeşitli uygulamalar toplanmıştır. Veri seti, beş farklı alternatif Android marketten toplanan ve her biri 10'dan fazla uygulamaya katkıda bulunmuş 488 benzersiz geliştiriciden olan 10,385 zararsız Android uygulamasından oluşmaktadır. Ayrıca, çeşitli kaynaklardan elde edilen 153 geliştiriciden 3,000'den fazla kötü amaçlı uygulamayı içermektedir. "Obfuscapk" aracı kullanılarak yaklaşık 6,000

karıştırılmış uygulama ve zararsız veri setinden elde edilen 1,000'den fazla versiyon uygulama içermektedir.

- Bu çalışma sonucu oluşan kodlar aşağıdaki adreste paylaşılmıştır: `https://github.com/emreaydogan/SourceCodeBasedAAA`

İLGİLİ ÇALIŞMALAR

Son yıllarda, mobil cihazlar kullanımları, popülerlikleri ve satış açısından masaüstü cihazları geride bırakmıştır. Bu çalışma Android ortamında zararlı ve zararsız uygulamaların yazarlarını bulmayı amaçladığı için, Android Yazarlık İlişkilendirmesi üzerine yapılan ilgili çalışmalar incelenmiştir. Ayrıca, yeni önerilen *smali* dosyalarından çıkarılan kaynak kodu tabanlı, metadata ve üçüncü parti kütüphane (TPL) öznitelik setleri, aşağıda verilen çalışmalarda kullanılan özellik setleriyle karşılaştırılmıştır.

APK dosyalarının dizgi analizine dayalı bir çalışma, Kalgutkar vd. tarafından önerilmiştir [14]. Bu çalışmada üç tür dizgi kullanılmıştır: *dex* dosyasındaki dizgi tanımlayıcı listesi, *dex* dosyasında sunulan tüm dizgi bileşenleri ve strings.xml dosyasından çıkarılan dizgiler. Daha sonra, bu dizgi listeleri üzerinde n-gram analizi uygulanarak, bir geliştiriciyi temsil eden öznitelik vektörü oluşturulmuştur. Son olarak, geliştiriciler Destek Vektör Makineleri (SVM) kullanılarak sınıflandırılmıştır. Önerilen yaklaşım, zararsız, kötü amaçlı ve karıştırılmış uygulamaları içeren farklı veri setlerinde değerlendirilmiştir ve sırasıyla %98, %96 ve %71 doğruluk oranları elde edilmiştir.

Gonzalez vd. [15] tarafından, *dex* dosyasından alınan bayt kodları *smali* temsiline dönüştürerek dizi ile ilgili, dizi ile ilgisiz ve n-gram özelliklerine dayanan bir yöntem önerilmiştir. Bu yöntem, profil oluşturma ve artımlı analiz olmak üzere iki aşamadan oluşmaktadır. İlk aşamada, yazar profilleri Rastgele Orman algoritması kullanılarak oluşturulmuştur. İkinci aşamada, yeni uygulamalar mevcut profillerle ilişkilendirilerek ve henüz mevcut yazarlara ilişkilendirilmemiş uygulamalar için yeni olası profiller bulunarak artımlı analiz uygulanmıştır. 33 yazar ve 1428 uygulama içeren bir veri

setinde %97,5 doğruluk oranı elde edilmiştir. Ayrıca, çeşitli kaynaklardan toplanan 131.000'den fazla uygulamaya ilgili yöntem uygulanmıştır.

Guoai vd. [16] tarafından önerilen yeni bir yaklaşım olan AppAuth, etiket isimleri, ikonlar, benzer paket isimleri ve dosya boyutları gibi ortak özniteliklere sahip bir grup yeniden paketlenmiş Android uygulamasının orijinal yazarını tespit etmektedir. APK dosyalarından çeşitli kodlama stili ile ilgili öznitelikler çıkarılmıştır: i) *dex* dosyasından ikili öznitelikler; ii) geri derleme yapılan dosyalardan kaynak öznitelikleri; ve iii) manifest dosyasından alınan yapılandırma öznitelikleri. 75 klon çiftinin 69 orijinal yazarı (%92) başarıyla tanımlanmıştır. Ayrıca, bağımsız geliştiricileri ve geliştirme takımlarını kullanarak tahmin performansı değerlendirilmiştir. Bu çalışmanın ana dezavantajı, karşılaştırdıkları uygulamaların orijinal bir uygulamanın yeniden paketlenmiş halleri olması durumunda, orijinal yazar yerine sahte bir yazar bulmalarıdır.

Wang vd. [17] üç tür dizgi ile ilgili özellik çıkarmıştır: DEX tabanlı, manifest tabanlı ve lib tabanlı özellikler. *dex* dosyalarından, tanımlayıcı isimler, talimat dizileri ve Android API'lerinin kullanımı toplanmıştır. Etkinliklerin, sağlayıcıların, hizmetlerin ve yayın alıcılarının isimleri çıkarılmış ve manifest dosyasından öznitelikler kullanılmıştır. Son olarak, uygulamalarda kullanılan TPL'lerin isimleri elde edilmiştir. Öznitelikleri geliştiricilerin profil vektörlerine dönüştürmek için *CountVectorizer*, *TFIDFVectorizer* ve *word2vec* modelleri kullanılmıştır. En iyi sonucu *word2vec* modeli vermiş ve sonraki deneylerde kullanılmıştır. Daha sonra, bu vektörlere üç makine öğrenimi modeli (Doğrusal SVM, Rastgele Orman ve Lojistik Regresyon) uygulanmış ve zararsız, kötü amaçlı ve karıştırılmış uygulamalar içeren farklı veri setlerinde geliştiricileri sınıflandırmak için kullanılmıştır. Sonuçlar, önerilen yaklaşımın tüm set için ortalama %92,5 doğruluk elde ettiğini göstermiştir. Ayrıca, 2,900 uygulamada AppAuth'tan daha üstün performans göstererek, yazarlık tanımlamayı %3,4 oranında iyileştirmiştir.

Daha önceki çalışmalardan farklı olarak, bu çalışmada önerilen yaklaşım uygulamaların market sayfalarından metadata bilgilerini içermektedir. Ayrıca, adil bir karşılaştırma

sağlamak için ortak bir veri seti kullanarak, literatürdeki Android Yİ alanındaki tüm özellik setleri karşılaştırılmıştır. Veri seti, diğer çalışmalarda kullanılanlardan daha büyüktür ve hem zararsız hem de kötü amaçlı uygulamaların kapsamlı bir karışımını içermektedir. Bu çalışma ve yukarıda bahsedilen çalışmalar Tablo 2.1'de karşılaştırılmıştır.

ANDROID

Android, Linux çekirdeği tabanlı en yaygın kullanılan mobil işletim sistemidir (OS) ve Ekim 2023 itibarıyla dünya çapında %70.76 kullanım oranıyla mobil endüstrinin temel taşlarından biridir [18]. Geliştiricilerin uygulamalarını kolayca dağıtabilmeleri için resmi bir market olan Google Play [19] sitesine sahiptir. Ayrıca, geliştiricilerin uygulamalarını Aptoide [20] ve Apkmirror [21] gibi alternatif Android marketlere yüklemelerine de olanak tanır. Bu nedenle, Android uygulamaları, *APK* dosyaları, bu tür marketlerden kolayca toplanabilir. Android uygulamaları, Java veya Kotlin programlama dilleri kullanılarak geliştirilir. Daha sonra, Java sanal makinesi için bir baytkoda derlenirler. Son olarak, bu Java baytkodları, Dalvik baytkodlarına çevrilir ve *dex* dosyalarında saklanır. Dalvik baytkodların okunması veya değiştirilmesi zor olduğundan, genellikle class dosyalarının içine bakmak için *smali* ara dili kullanılır. *smali*, Dalvik baytkodunun okunabilir temsilidir. *smali* dosyaları aynı zamanda doğrudan APK dosyalarından da çıkarılabilir.

Bir Android uygulaması *dex* dosyaları, manifest dosyası, kaynak dosyaları ve dizgi dosyaları gibi çeşitli öğelerden oluşur. Bu dosyaların tümü, uygulama hakkında bilgi toplamak için kullanılabilir. Örneğin, manifest dosyası uygulamalarda kullanılan servis, aktivite ve izinlerin sayısı gibi bilgileri içerirken, kaynak dizinlerinde uygulamalarda kullanılan resimler, kütüphaneler ve XML dosyaları bulunabilir. Dizgi kaynakları, uygulamalar için metin dizgileri sağlar [22]. Üç tür dizgi vardır: Dizgi, Dizgi Dizisi ve Miktar Dizgileri. Tüm dizgiler, stil işaretleme ve biçimlendirme argümanlarını uygulama kapasitesine sahiptir. *dex* dosyası, Java/Kotlin sınıf dosyalarını içerir. Sınıf dosyalarını *dex* dosyalarının ara temsilleri olan *smali* dosyalarına dönüştürmek

mümkündür. Derleme işlemi gerçekleştirildiğinde bir kaynak kodunun bazı dilbilgisi ve biçimlendirme özellikleri kaybolmasına rağmen, Dalvik tabanlı çalıştırılabilir dosyalar, *smali* dosyalarında bazı içgörülü bilgiler içerir. Şekil 3.1, örnek bir Java/C++ uyumlu kod parçasını, ona karşılık gelen *smali* ve assembly kodlarını sırasıyla göstermektedir. *smali* kodu, değişken isimleri ve metod imzaları gibi bilgileri korurken, assembly kodu, derleme sürecinden sonra bu tür bilgileri kaybeder. Bu nedenle, kaynak kodu Yİ ile ilgili bazı öznitelikler *smali* dosyalarından elde edilebilir.

Android uygulamaları, bu çalışmada kullanılan farklı öznitelik setlerine karşılık gelen çeşitli bileşenlere ayrılmıştır.

- **Yapılandırma**: Android uygulamaları, aktiviteler, servisler, yayın alıcılar ve içerik sağlayıcılar olmak üzere dört ana unsurdan oluşurlar. Her bileşenin kendi amacı ve yaşam döngüsü vardır ve bu döngü bileşenlerin nasıl oluşturulup yok edileceğini tanımlar. Aktiviteler, kullanıcıların uygulamalarla nasıl etkileşimde bulunduklarını tanımlar. Kullanıcıların sistem kaynaklarını kullanmalarına olanak tanıyan kullanıcı arayüzlerini tanımlamak için kullanılırlar. Android'deki her tekil uygulama ekranı aktivitelere denk gelir. Bazı aktiviteler, diğer aktiviteleri tetikleyebilir. Servisler, arka planda uzun süreli işlemler gerçekleştirir ve bir kullanıcı arayüzü içermezler. Yayın alıcılar, bir uygulamanın bir sistem veya bir uygulama olayına bağlanmasını sağlarlar. Android işletim sistemi, bir olay tetiklendiğinde bağlı uygulamayı bilgilendirir. İçerik sağlayıcılar, uygulama veritabanlarındaki verilere erişimde kullanılır. Bir uygulama, kendi veritabanını diğer uygulamalarla paylaşmak için içerik sağlayıcıları kullanabilir. Böylece, tek bir içerik parçası birden fazla uygulama arasında dağıtılabilir. Tüm bileşenler, Android manifest dosyasında beyan edilir. Geliştiriciler, uygulamalarını geliştirirken aynı Android manifest dosyasını yeniden kullanma eğilimindedir. Bu nedenle, manifest dosyasındaki bilgiler, örneğin bileşenlerin sayısı, geliştiricileri birbirinden ayırt etmekte yardımcı olabilir.

- **Kaynak Dosyaları**: Android uygulamaları, *res* dizininde saklanan resimler, sesler, ikonlar ve yerel kütüphaneler gibi kaynak dosyaları içerebilir. Dosyalar, dil desteği sağlama ve kullanıcı arayüzü için resimler sunma gibi çeşitli nedenlerle kullanılabilir. *res* dizinindeki bazı dosyalar, dinamik kütüphaneler, veritabanı dosyaları ve yük dosyaları gibi, çalışma zamanında erişilebilir. Geliştiriciler, bu dosyalara bazı özel ve kritik bilgileri yerleştirir. Bu nedenle, bu dizinlerin özelliklerini analiz etmek önemlidir.

- **DEX Kodu**: *dex* dosyası, Şekil 3.2'de gösterildiği gibi, uygulamaların yapısı hakkında bilgi içerir. AppAuth [16], *dex* dosyalarındaki veri bölümünü temel olarak analiz eder ve metodlar, sınıflar ve alan yapılarına, ayrıca notasyon, arayüz, hata ayıklama bilgileri vb. üzerine odaklanır. Uygulama boyutları önemli ölçüde değişebileceğinden, bu özelliklerin sayısal değerleri yerine oran değerlerini kullanır.

- **Dizgi**: strings.xml dosyasındaki dizgiler, uygulama dizgi öznitelikleri olarak satır satır çıkarılır. Bu dizgiler, uygulamaların kaynak koduna veya diğer kaynak dosyalarına referanslar içerir. Bunlar, uygulamanın adı gibi kullanıcılara gösterilen statik dizgilerdir. *dex* dosyası, dizgi id listesi, tip id listesi ve metod id listesi gibi farklı bölümlerden oluşur. Dizgi id listesi esas olarak bir uygulamanın kaynak kodunda kullanılan dizgileri içerir. Ancak, kötü amaçlı yazılım geliştiricileri, kötücül kodlarını çalışma zamanında çalıştırmak ve analizden kaçınmak için dizgi ID listesine yerleştirir. Bir *dex* dosyasının dizgi id listesi bölümündeki dizgiler, DEX dizgi öznitelikleri olarak çıkarılır.

- **Kaynak Kodu**: *smali* dosyaları, sınıf dosyalarındaki Java baytkodlarından üretilen Dalvik-byte/*smali* kodlarını içerir. Yüksek seviyeli bir dilden üretilirler ve *smali* için Android Runtime (ART) aracılığıyla makine koduna dönüştürülürler. İşletim sistemi, bu makine kodlarını çalıştırarak uygulamaları çalıştırır. Şekil 3.1'de, *find_maximum* adında bir Java fonksiyonuna karşılık gelen *smali* ve assembly kodları gösterilmiştir. Şekil 3.1'de görüldüğü gibi, orijinal

kaynak kodlarındaki fonksiyon parametrelerinin ve global ve yerel değişkenlerin isimleri assembly dosyalarında korunmaz, ancak *smali* dosyalarında korunur. Örneğin, *n, max, index* yerel değişkenleri ve *a, n* fonksiyon parametreleri şekilde verilen *smali* kodunda korunmuştur. Geliştiricilerin, değişken isimlerinde $ karakteri veya rakamlar gibi tanımlayıcılarda aynı formatı kullanma eğiliminde oldukları gösterilmiştir. Bu nedenle, bu tür isimlerin geri derleme yapılmış kodda korunması, geliştiricileri tanımlamada yardımcı olabilir.

- **İzin**: Mobil cihazlar cihazın sistem kaynaklarını kullanmak için izinlere ihtiyaç duyar. Uygulamalar, yalnızca manifest dosyasında tanımlanan izinler aracılığıyla kamera ve GPS gibi cihaz kaynaklarına erişebilir ve bunları kullanabilir. Normal izinler otomatik olarak verilirken, tehlikeli izinler kullanıcı onayı gerektirir. Çoğu uygulama, kullandıklarından daha fazla izin ister. İstenen izinler ile çalışma zamanında kullanılan izinler arasındaki farkı değerlendiren bir çalışmaya göre, uygulamalar ortalama olarak ihtiyaç duyduklarından %30 daha fazla izin kullanırlar[23]. Bir geliştirici farklı kategorilere ait uygulamalar geliştirmiş olsa da, kolaylığı nedeniyle bu uygulamalarda aynı manifest dosyasını kullanabilir. Ayrıca, aynı yazar tarafından geliştirilen uygulamalarda aynı TPL'yi kullandıkları için, bu kütüphanelerin gerektirdiği aynı izinleri kullanmaları gerekir.

- **Kütüphane**: Geliştiriciler, bazı işlevlerini uygulamalarında sağlamak için sıfırdan geliştirme yapmak yerine TPL (Üçüncü Parti Kütüphaneler) kullanmayı tercih ederler. Çoğu uygulamanın 20'den fazla TPL kullandığı gösterilmiştir [24] ve uygulama kodunun büyük bir kısmı bu tür kütüphanelere aittir [25, 26]. Geliştiriciler, aynı işlevselliği sağlamak için genellikle tanıdık oldukları aynı kütüphaneyi kullanma eğilimindedir.

- **Metadata**: Android uygulamalarının merkezi mobil marketler aracılığıyla dağıtımı, geleneksel uygulama dağıtım yöntemlerinden farklıdır. Bu marketler, uygulama açıklaması, uygulama derecelendirmesi ve kullanıcı incelemeleri gibi,

uygulama hakkında bilgi içeren, metadata olarak bilinen bilgileri uygulamanın kendisiyle birlikte içerirler.

YAZARLIK İLİŞKİLENDİRMESİ

Yazarlık ilişkilendirmesi, bir metin veya yazılımın orijinal yaratıcısını belirlemeyi amaçlar. Literatürde [27, 28], yazılımda yazarlık ilişkilendirmesi süreci, Şekil 4.1'de gösterildiği gibi dört adımlık bir yaklaşımı takip eder. Yazarlık ilişkilendirmesi süreci, hem ikili hem de kaynak kodu kullanır. Kodun özellikleri, sözcüksel, sözdizimsel, anlamsal, davranışsal ve uygulamaya özgü öznitelikler de dahil olmak üzere, analiz için koddan çıkarılır. Daha sonra kodun çeşitli temsilleri kullanılır, bunlar arasında andaçlar, dizgiler, n-gramlar, deyimler, çizgeler ve ağaçlar bulunur. Yazarlık modelleri, profil tabanlı, örnek tabanlı veya hibrit olarak kategorize edilebilir. Bu modeller, benzerlik tabanlı, vektör uzayı, olasılıksal ve meta-öğrenme yöntemleri gibi metodolojilerle desteklenir. Bu sürecin sonucu olarak, yazarlık ilişkilendirmesi, intihal tespiti ve yazar niyetlerinin belirlenmesi gibi sonuçlar elde edilir.

Kalgutkar vd. [27] ile Gonzalez [28], yazarlık ilişkilendirmesi sürecinde kullanılan öznitelikleri ayrıntılı bir şekilde açıklamışlardır. Bu öznitelikler, sözcüksel, sözdizimsel, anlamsal, davranışsal ve uygulamaya özgü olmak üzere çeşitli kategorilere ayrılır. Her bir özellik tipinin avantajları ve dezavantajları, kodun analizi ve yazarın belirlenmesi sürecinde önemli faktörler olarak öne çıkar.

Sözcüksel özellikler, kodun temel andaç ve dizgilerinden türetilir ve programlama dilinden bağımsızdır, ancak kod biçimlendiricileri tarafından kolayca değiştirilebilirler. Sözdizimsel özellikler, kodun yapısını ve bir yazarın problem çözme yaklaşımını yansıtır, ancak dil bağımlılıkları ve özellik seçimi zorluklarıyla karşı karşıyadırlar. Anlamsal özellikler, kodun mantığını odak alır ve kod dönüşümüne karşı dirençlidirler, ancak karmaşık çıkarım süreçleri gerektirebilirler. Davranışsal özellikler, yazılımın sistem çağrıları ve ağ bağlantıları gibi işlevlerini inceler, güçlü analiz potansiyeline sahiptir fakat yanlış pozitif ve negatif sonuçlara yol açabilir. Uygulamaya özgü özellikler,

kaynak koduna erişim gerektirmeyen, ancak kod karıştırmaya karşı savunmasız olan ek bilgileri kullanır.

Yİ'de kullanılan temsil yöntemleri arasında andaçlar, dizgiler, n-gramlar, ifadeler (idioms), çizgeler, ağaçlar ve gömüler bulunur. Andaçlar ve dizgiler kodun en küçük anlamlı parçalarını ve belirli metin dizgilerini temsil eder. N-gramlar, kodun yapısal özelliklerini yakalar. İfadeler, dil özgü özelliklerdir ve karıştırmaya karşı dirençlidir. Çizgeler ve ağaçlar, kodun yapısını ve sözdizimsel hiyerarşisini temsil eder, semantik anlamlar türetmeye yardımcı olur. Gömüler, tek bir temsilin yetersiz kaldığı durumlarda farklı temsillerin kombinasyonunu kullanır.

Yİ süreci, çok sınıflı, tek etiketli bir kategorizasyon görevi olarak tanımlanır. Bu süreçte, profil tabanlı ve örnek tabanlı modeller arasında bir seçim yapılır. Profil tabanlı model, bir yazarın tüm çalışmalarını kapsayan benzersiz bir stil atfederken, örnek tabanlı model her bir uygulamayı ayrı bir varlık olarak değerlendirir. Bu modellerin seçimi, benzerlik tabanlı, vektör uzayı, olasılıksal yöntemler ve makine öğrenimi gibi çeşitli karşılaştırma metodolojileriyle desteklenir. Bu metodolojiler, bilinmeyen bir kod örneğinin yazarını belirlemede kritik rol oynar. Makine öğrenimi teknolojilerinin gelişimi, Yİ alanında daha doğru ve hızlı sonuçlar elde edilmesini sağlar.

ÖNERİLEN YÖNTEM

Şekil 5.1, önerilen modelin mimarisinin şematik temsilini gösterir. Geri derleme yapılan APK dosyaları, öznitelik çıkarımı için giriş noktası olarak hizmet eder ve sonuç olarak, sınıflandırma için tasarlanmış sabit boyutlu öznitelik vektörleri oluşturulur.

APK dosyalarına ilk olarak apktool [29] kullanılarak geri derleme yapılmış ve *smali* dosyaları elde edilmiştir. Daha sonra, uygulamaların Dizgi, AppAuth ve SPL öznitelikleri bu dosyalardan ve APK dosyalarından toplanmış ve her uygulama için sabit boyutlu bir öznitelik vektörleri sınıflandırıcılara giriş olarak verilmiştir. *scikit-learn* kütüphanesindeki *SimpleImputer* [30], öznitelik vektörlerindeki eksik değerleri doldurmak için kullanılmıştır. Bu imputasyon işleminden sonra,

bu öznitelikler, *scikit-learn* kütüphanesindeki *StandardScaler* [31] kullanılarak ortalamadan çıkarılarak birim varyansa ölçeklenmiştir.

Sınıflandırma algoritmaları yürütülmeden önce iki aşamalı boyut indirgeme kullanılmıştır. İlk olarak, veri setinden sıfır varyanslı öznitelikler elenmiştir. Daha sonra, tek değişkenli istatistiksel testlere dayanarak en iyi özellikleri seçen tek değişkenli öznitelik seçimi uygulanmıştır. *scikit-learn* kütüphanesi [32] kullanılarak beş farklı sınıflandırma algoritması uygulanmıştır. Daha sonra, katmanlı onlu çapraz doğrulama kullanılmıştır. Katmanlı onlu çapraz doğrulama, orijinal dağılımdaki pozitif ve negatif örneklerin oranının tüm iterasyonlarda korunduğundan emin olduğu için, özellikle veri setinin dengesiz olduğu durumlarda yararlıdır [33]. Yöntem beş kez çalıştırılmış ve bu beş çalışmanın ortalaması sonuçlarda sunulmuştur.

Sınıflandırma algoritmaları, Android uygulama alanı içinde yazarlığı tahmin etmek için uygulanmıştır. Bu çalışmada incelenen sınıflandırma algoritmaları Rastgele Orman, K-En Yakın Komşular, SVM, Gaussian Naive Bayes ve LightGBM algoritmalarıdır.

Mümkün olan en iyi model performansına ulaşmak için, *GridSearchCV* tekniği kullanılmıştır. *GridSearchCV*, *scikit-learn* kütüphanesi tarafından sağlanan, belirli bir parametre ızgarası üzerinde tükenmiş bir arama yaparak verilen bir tahminci için en iyi mümkün hiperparametre kombinasyonunu belirlemeye olanak tanıyan bir yöntemdir. Her makine öğrenmesi algoritması için başlangıç parametreleri ve değerleri Tablo 5.7'de verilmiştir. *GridSearchCV*'yi katmanlı onlu çapraz doğrulama tekniğiyle birleştirerek, seçilen hiperparametrelerin veri setinin farklı alt kümelerinde iyi performans gösterdiğinden emin olunmuştur. Bu yaklaşım, makine öğrenimi modellerinin genelleştirilmesini iyileştirmektedir. *GridSearchCV*'nin çapraz doğrulama bağlamında kullanılması, modellerin ince ayar yapılmasına ve tahmin güçlerini artırılmasına yardımcı olduğu için araştırma bu çalışma metodolojisinin ayrılmaz bir parçasıdır.

VERİ SETİ

Bu çalışmada, market ve kötü amaçlı yazılımlar olmak üzere iki farklı veri seti oluşturulmuştur. Market veri setini oluşturmak için, çeşitli alternatif marketlerden zararsız uygulamalar toplamak üzere *Scrapy* kütüphanesi [34] kullanılarak bir ağ kazıma uygulaması hazırlanmıştır. Ayrıca, bu veri setine diğer çalışmalardan [14, 15, 35, 36] uygulamalar dahil edilmiştir.

Zararsız veri seti, Ocak 2020 ve Ağustos 2020 arasında Apkpure [37], Apkmirror [21], Onemobile [38] ve Aptoide [20] gibi farklı alternatif Android marketlerden toplanan uygulamaları içermektedir. Ancak, güncel uygulamaların eski versiyonlarının mevcudiyeti nedeniyle 2020 yılından çok daha erken yazılmış uygulamaları da içermektedir. Zararsız veri setlerinin yıllara göre dağılımı Tablo 5.1'de sunulmuştur. Başlangıçta, 200.000 uygulama indirilmiştir. Sonrasında, daha önceki çalışmalar en az on uygulama içeren geliştiricileri kullandığı [14, 16] için, ondan az uygulamaya sahip geliştiriciler elenmiştir. Bunun sebebi, geliştiriciler arasında ayrım yapabilen iyi bir model oluşturmak için her geliştirici için yeterli uygulamaya sahip olmak gerekliliğinden doğmaktadır.

Kötü amaçlı yazılım veri seti, Ransomware, Adware ve SMS [39] gibi kötü amaçlı yazılım ailelerine ait uygulamaları ve güvenlikle ilgili çalışmalarda tanıtılan ve kullanılan Rmvdroid [40], Drebin [35], Genome [15] ve Koodous [14] gibi veri setlerinden uygulamaları içerir.

Her uygulama, bir Android cihazına yüklenmiş bir geliştirici sertifikası ile imzalanmalıdır. Bu nedenle, aynı imzayı paylaşan uygulamalar aynı geliştiriciye atanır ve her uygulama ilgili imzaya göre gruplandırılır. Bu çalışamda imzalar, APK dosyalarından *print-apk-signature* aracı [41] kullanılarak çıkarılmıştır.

DENEY SONUÇLARI

Bu çalışmada Android uygulamalarında Yazarlık İlişkilendirmesine dair aşağıdaki araştırma sorularına (AS) cevap bulmaya çalışılmıştır:

- **AS1 - Sınıflandırma algoritmalarının Yİ problemi çözümünde performansları nasıl?**

  Deneylerde Rastgele Orman, K-En Yakın Komşular, Destek Vektör Makineleri, Gaussian Naive Bayes ve LightGBM algoritmaları kullanılmıştır. Bu denetimli makine öğrenimi teknikleri, literatürde kod Yazarlık İlişkilendirme için kullanılan popüler yöntemler arasındadır [27]. Market ve kötü amaçlı yazılım veri setlerinde bu algoritmaların hiperparametrelerini ayarlamak için scikit-learn'deki GridSearchCV fonksiyonu kullanılmıştır. GridSearchCV, bir tahminci için belirtilen parametre değerlerinin tükenmiş bir aramasını gerçekleştirir. Bu deneyde, AppAuth [16] ve Dizgi Analizi [14]'de kullanılan öznitelikler dahil olmak üzere tüm öznitelik setleri kullanılmıştır. Bazı uygulamaların açıklamaları olmadığı için yalnızca metadata öznitelikleri hariç tutulmuştur. Algoritmalar hem zararsız hem de kötü amaçlı yazılım veri setleri kullanılarak çalıştırılmıştır. Performans metrikleri olarak doğruluk ve F1 skoru kullanılmıştır. Bunun nedeni, etkili bir modelin hem yüksek hassasiyet hem de yüksek duyarlılık göstermesi gerekliliğidir. Ek olarak, her bir algoritmanın sınıflandırma süresi de değerlendirilmiştir.

  Karşılaştırma sonuçları Tablo 6.1'de gösterilmiştir. Yüksek doğruluk ve F1 skorlarının yanı sıra, Rastgele Orman (RF) algoritmasının sınıflandırma süresi diğer algoritmalara göre daha makul olduğundan, sonraki deneylerde RF algoritması kullanılmıştır. Ayrıca, RF'nin varsayılan parametreleri, optimal parametreleriyle elde edilen sonuçlara çok benzer bir performans üretmiştir. LightGBM, RF'den daha iyi performans göstermesine rağmen, sınıflandırma süresi yüksektir. Üstelik, sonuçlar farklı parametreler için oldukça değişkendir. Ancak, yüksek doğruluğu nedeniyle, LightGBM'in kullanımı gelecekte araştırılmaya değerdir.

- **AS2 - N-gram öznitelikleri için ideal set büyüklüğü nedir?**

  [14]'de önerildiği gibi, bu deneyde 3-gramlar kullanılmıştır. İlgili çalışmada, sistemin, n-gram boyutunun 1'den 3'e artırılmasıyla daha iyi performans

gösterdiği gözlemlemişlerdir. Ayrıca, n-gram boyutunun daha da artırılmasıyla sistem performansının düştüğünü belirlemişlerdir. Bu deneyde, optimal n değerini belirlemek için ön analiz yapılmıştır. Yaklaşım [14]'dekiyle karşılaştırıldığı için, n-gramlar için aynı n değerinin kullanılmasına karar verilmiştir. Yüksek sayıda uygulama içeren veri setleri için deneylerde kullanılan n-gramların sayısını sınırlamak ve bellek tüketimini azaltmak için bir *HashingVectorizer* kullanılmıştır.

3-gramların sayısının sınıflandırma doğruluğu ve eğitim süresi üzerindeki etkileri sırasıyla Şekil 6.1a ve 6.1b'de gösterilmiştir. Şekilde açıkça görüldüğü üzere, sınıflandırmada 10.000'den fazla 3-gram kullanıldığında eğitim süresi artmaktadır. Ancak, bu, doğruluk açısından önemli bir iyileşme sonucu doğurmamaktadır. Dikkate değer bir durum ise, boyutu ve büyük miktarda 3-gram üretmesi nedeniyle, market veri seti en fazla 50.000 3-gram ile çalıştırılabilmiştir.

- **AS3 - TPL (Üçüncü Parti Kütüphanelerin) kullanımı, Yİ problemi çözümüne iyileştirmeler getiriyor mu?**

  Yİ üzerinde TPL'nin etkisini göstermek için iki ayar kullanılmıştır. İlk ayarda, kaynak kod tabanlı öznitelikler yalnızca geliştiriciler tarafından yazılan özel(custom) kodlardan çıkarılmıştır. İkinci ayarda ise, öznitelikler yine aynı uygulamalardan çıkarılmış, ancak bu sefer TPL'ler dahil edilmiştir (**tüm** kodlar).

  Tablo 6.2'de gösterildiği gibi, kaynak kod tabanlı öznitelikleri çıkarırken TPL'nin dahil edilmesi çok daha iyi bir performans sağlamıştır (özel src vs. tüm src). Daha sonra, özel kod Tablo 5.6'de sunulan diğer özniteliklerle zenginleştirilmiş ve Tablo 6.2'deki tüm kod ile karşılaştırılmıştır. Sonuçlar açıkça göstermektedir ki, TPL'den kaynak kod tabanlı öznitelikler çıkarmak yerine, TPL'nin kendi başına öznitelik olarak varlığı Yİ için yeterlidir ve TPL kodunun dahil edilmesinden çok daha iyi sonuçlar üretir. Bu nedenle, sonraki deneylerde, kaynak kod tabanlı öznitelikler yalnızca özel koddan çıkarılmış ve TPL öznitelikleriyle birlikte kullanılmıştır (özel src+lib).

- **AS4 - Önerilen yaklaşım uygulamaların geliştiricisini belirlemede ne kadar etkilidir?**

  Bu deneyde, Önceki Araştırma Sorularında deneysel ayarlar belirlendikten sonra, metadata hariç yeni önerilen öznitelik setleri (src+perm+lib veya SPL), [14, 16]'de kullanılan diğer öznitelik setleriyle karşılaştırılmıştır. Bu amaçla, [14]'de verilen dizgi ile ilgili öznitelikleri çıkarmak için sıfırdan kod geliştirilmiştir. AppAuth [16] yazarları, makalelerinde bahsedilen öznitelikleri çıkarmamızı sağlayacak kaynak kodlarını paylaşmıştır. Bu sayede, AppAuth özniteliklerini elde etmek için bu kodlar doğrudan uygulanmıştır.

  Tablo 6.3'de gösterildiği gibi, yeni önerilen öznitelikler AppAuth'tan daha iyi performans gösterirken, [14]'deki dizgi özniteliklerinde daha az doğruluk üretmektedir. Ancak, yukarıda belirtildiği gibi, tüm 3-gram özniteliklerini çıkarmak, sınıflandırma süresi açısından uygulanabilir olmayabilir. Bu nedenle, burada yalnızca 10.000 3-gram kullanılmaktadır. [14]'de değerlendirme için kullanılan ve göreceli olarak küçük bir veri seti olan Genome veri setinden ise tüm 3-gramlar çıkarılmıştır. Genome veri setinde, önerilen öznitelikler, dizgi ile ilgili özniteliklere göre biraz daha iyi performans göstermiştir. Tüm öznitelikler dahil edildiğinde, market ve kötü amaçlı yazılım veri setlerinde en iyi performans elde edilmiştir. Ayrıca, her bir öznitelik setini çıkarmak için gereken süre de hesaplanmıştır. Tablo 6.4 göstermektedir ki, n-gram öznitelikleri, diğer öznitelik setlerinden en az bir buçuk kat daha fazla zaman gerektirmektedir. SPL ve AppAuth özniteliklerinin çıkarma süreleri nispeten yakındır. Bir öznitelik çıkarıcı yalnızca bir kez gerçekleştirilse de, öznitelik çıkarma işlemi, önemli miktarda n-gram üretildiğinde, özellikle sınırlı RAM'e sahip sistemler için, bellek ile ilgili zorluklara yol açabilir. [42]'de vurgulandığı gibi, n-gram yaklaşımının birincil sınırlılığı, metin/uygulama boyutu arttıkça n-gramların üssel olarak büyüme eğilimidir. Bu karmaşıklık, sınırlı hesaplama kapasitesine sahip sistemler için yöntemi genellikle uygulanamaz hale getirir.

- **AS5 - Uygulamaların metadata bilgisi, uygulamaların geliştiricisini ilişkilendirmede yardımcı oluyor mu?**

  Burada, öncelikle optimal n değeri ve n-gram sayısının ne olması gerektiği araştırıldı. Bu amaçla, açıklama metinlerindeki her n ardışık kelimenin n-gramları çıkarıldı ve her adımda bir kelime sağa doğru ilerlendi. Optimal n değerini belirlemek için, 1'den 5'e kadar farklı n-gram boyutlarında ön denemeler yapıldı. Sonrasında, en iyi sonuçları elde etmek için gerekli olan n-gram sayısı tespit edildi. İkinci aşamada ise, metadata özniteliklerinin diğer öznitelik setleriyle birleştirilmesinin doğruluk üzerindeki etkisi incelendi.

  Bu deneyde, market veri setinin tamamı ve kötü amaçlı yazılım veri setleri arasından yalnızca Rmvdroid [40] kullanıldı. Bunu sebebi yalnızca bu veri setleri uygulama açıklamalarını içermektedir. Şekil 6.5a, farklı n değerleri için doğruluk sonuçlarını göstermektedir. n-gram sayısı 10.000 olarak belirlenmiştir. Gözlemlenebileceği üzere, 1-gramlar diğer n-gramlardan daha iyi sonuçlar üretmektedir. n değerini belirledikten sonra, daha sonraki deneylerde optimal 1-gram öznitelik sayısını araştırdık. Şekil 6.5b, yalnızca metadata öznitelikleri kullanılarak elde edilen doğruluk sonuçlarını farklı sayıda 1-gramlar altında göstermektedir. Şekil 6.5b'de gösterildiği gibi, 1-gramların sayısı sonuçlar üzerinde önemli bir etkiye sahiptir. n-gram analizi önemli miktarda sistem kaynağı gerektirdiğinden, 100.000 1-gramdan sonra sonuç alınamamıştır. 100.000 1-gram dahil edildiğinde, market ve kötü amaçlı yazılım veri setleri için sırasıyla yaklaşık %84 ve %74 doğruluk elde edilmiştir. Bu, açıkça uygulama açıklamalarının Android Yİ için faydalı olduğunu göstermektedir. Ancak, 10.000 1-gramın sonuçlarının da oldukça yeterli olduğu ve gereken zamanın 100.000 1-gramdan çok daha az olduğu görülebilir. Bu nedenle, metadata analizindeki gelecek deneyler için 10.000 1-gram seçilmiştir.

- **AS6 - Uygulamaları geliştiricilerine ilişkilendirmede en önemli öznitelikler nelerdir?**

Sonuçlar, metadata hariç tüm öznitelikler ve Rastgele Orman algoritması kullanılarak elde edilmiştir. Daha sonra, tüm özniteliklerin önem değerleri *scikit-learn* kütüphanesinden çıkarılmış ve sırasıyla market ve kötü amaçlı yazılım veri setleri için Şekil 6.6a ve 6.6b'de gösterilmiştir.

Sonuçlar, her iki veri setinde de kaynak kod tabanlı ve dizgi/n-gram özniteliklerinin diğer özniteliklere göre daha büyük bir etkiye sahip olduğunu göstermektedir. AppAuth'ta kullanılan öznitelikler (conf, dex, rsrc), özellikle kötü amaçlı yazılım veri setinde olumlu bir etkiye sahiptir. Tablo 6.10, önem açısından ilk 50 öznitelik arasında yer alan her veri seti için öznitelik sayısını göstermektedir. En önemli 50 özniteliğin sıralaması Tablo 6.11'de listelenmiştir. Bu sonuçlar ayrıca, özellikle kötü amaçlı yazılım veri setinde, kaynak kod tabanlı özniteliklerin diğer özniteliklere kıyasla çok daha fazla etkiye sahip olduğunu, n-gram özniteliklerinin ise market veri setinde baskın olduğunu göstermektedir.

- **AS7 - Modelin performansını azaltmadan öznitelik sayısını azaltabilir miyiz?**

  Tüm özniteliklerin en iyi %20, %40, %60, %80 ve %100'lük kısımları, scikit-learn'deki *f_classif()* kullanılarak ANOVA F-değerleri hesaplanarak çıkarıldı. ANOVA, öznitelik seçimi için kullanılan parametrik bir istatistiksel hipotez testidir. İki veya daha fazla veri örneğinin ortalamaları önce hesaplanır ve sonra bu veri örneklerinin aynı dağılımdan gelip gelmediği belirlenir.

  Her öznitelik setinin performansı Şekil 6.7'de gösterilmiştir. Öznitelik vektöründe sıfır etkiye sahip öznitelikler de bulunmaktadır. Kötü amaçlı yazılım veri setinin (435), market veri setine (174) kıyasla daha fazla sıfır etkiye sahip özniteliğe sahip olması nedeniyle, son yüzdelikte çok fazla iyileşme olmamıştır. Bu şekil ayrıca, zaman ve kaynak kullanımı önemliyse, tüm özniteliklerin yerine daha az öznitelik kullanılabileceğini (örneğin, %60) göstermektedir.

- **AS8 - Geliştirici başına düşen uygulama sayısı, sınıflandırma performansını etkiliyor mu?**

  Bu çalışmada kullanılan veri seti, piyasada ondan fazla uygulamaya sahip bireysel geliştiricileri elde etmenin zor olması nedeniyle, esas olarak birden fazla geliştirici tarafından yazılan şirket uygulamalarını içermektedir. Ayrıca, geliştiricilerin farklı sayıda uygulamalara sahip olması nedeniyle veri seti dengesizdir. Bu nedenle, zararsız ve kotücül veri setlerinde en az 40 uygulamaya sahip geliştiriciler öncelikle çıkarılmış ve sırasıyla market ve kötü amaçlı yazılım veri setleri için 37 ve 19 yazar elde edilmiştir. Daha sonra geliştirici başına rastgele 10, 20, 30 ve 40 uygulama on kez seçilmiştir. Bu nedenle, her seferinde farklı uygulama grupları seçilmiştir.

  Şekil 6.8, geliştirici başına uygulama sayısı artırıldığında, Android Yİ'nin doğruluğunun da arttığını göstermektedir. Daha az yazarı olan daha küçük bir veri seti olduğu için, kötü amaçlı yazılım veri setindeki sonuçlar, geliştirici başına 20 uygulama ile eğitildiğinde bile çok yüksektir. Bu nedenle, kötü amaçlı yazılım veri setinde geliştirici başına 20, 30 ve 40 uygulama kullanılarak eğitilen modeller arasındaki farklar birbirine benzerdir.

- AS9 - **Önerilen model, aynı yazar tarafından geliştirilen uygulamaların farklı versiyonlarını başarıyla belirleyebiliyor mu?**

  Bu çalışmada, iki veri seti oluşturulmuştur. İlk olarak, market veri setinde uygulamaların birden fazla versiyonu varsa, yinelenen uygulamalar elenmiş ve yalnızca uygulamanın markette bulunan ilk versiyonu pazar veri setinde bırakılmıştır. Bu eğitim veri setine sürüm içermeyen veri seti denilmiştir; versiyonları ise başka bir veri setine, test seti adı verilen sete konulmuştur. Bunun sonucunda, 42 geliştirici tarafından uygulanan 1.193 eğitim ve 1.183 test uygulaması elde edilmiştir.. Sürüm içermeyen veri setindeki tüm geliştiricilerin, önceki deneylerde olduğu gibi, en az on uygulaması bulunmaktadır.

İlk olarak, sürüm içermeyen sette katmanlı onlu çapraz doğrulama uygulanarak %80.7 doğruluk elde edilmiştir. Tablo 6.3'de gösterildiği gibi, tüm sürümler dahil edildiğinde, doğruluk biraz daha yüksek (%82.6) olmuştur. Daha sonra, "bir uygulamanın bir versiyonu eğitimde yer alıyorsa, bu uygulamanın yeni versiyonları önerilen yaklaşımla tespit edilebilir mi?" sorusunu yanıtlamak için, model sürüm içermeyen bir veri seti kullanılarak eğitim yapılmış ve test veri setinde değerlendirilmiştir. Burada, yüksek doğruluk (%91.7) elde edilmiştir. Uygulamaların farklı versiyonları arasındaki benzerlikler SimiDroid [43] kullanılarak hesaplanmıştır. Ancak, eğitimdeki ilk mevcut versiyonlara olan benzerliklerine göre uygulamaların tanımlanamayan versiyonları arasında önemli bir korelasyon bulunamamıştır. İlk mevcut versiyona olan benzerlikler genellikle versiyon numaralarına orantılı olarak azalsa da, test setinde istisnai durumlar bulunmaktadır. Bir uygulamanın versiyonu ile eğitim setindeki ilk mevcut versiyon arasındaki benzerlik düşük olsa bile, önerilen yaklaşım bu tür versiyonların yazarlarını başarıyla tanımlayabilir. Bu sonuçlar, aynı uygulamanın farklı versiyonları arasındaki benzerlik düşük olsa bile, geliştiricinin imzasının korunduğunu göstermektedir.

- **AS10 - Android Yİ üzerinde karıştırma tekniklerinin etkisi nedir?**

  Bu çalışmada, Android Yİ'nin karıştırılmış uygulamalardaki performansını daha iyi anlamak için, market ve Genome veri setlerindeki uygulamalarda "Obfuscapk" [44] aracı kullanılarak karıştırılmış uygulamalar elde edilmiştir. "Obfuscapk" [44], kara kutu şeklinde çalışır, ileri düzey karştırma özelliklerini destekler ve yeni tekniklerle kolayca genişletilebilen modüler bir mimariye sahiptir. Karıştırma tekniklerinin uygulanması sırasında karşılaşılan bazı hatalar nedeniyle, bu deneyde kullanılan uygulama sayısı orijinal veri setinden (≈40%) düşüktür. Ancak, [14]'den çok daha büyük bir veri setidir (320 yazardan 6055 uygulama). Tablo 6.12'de listelenen altı farklı karıştırma tekniği kullanılmıştır. Bu karıştırma teknikleri, şifreleme ve yeniden adlandırma olmak üzere iki kategoriye gruplandırılabilir.

Farklı karıştırma tekniklerinin etkileri Tablo 6.13'de gösterilmektedir. Şifreleme karıştırma teknikleri sonuçları etkilemezken, yeni önerilen kaynak kod tabanlı öznitelik setleri, Tablo 6.13'de gösterildiği gibi, yeniden adlandırma karıştırma tekniklerine karşı hassastır, çünkü bunlar uygulamaların değişken, metod ve sınıf isimlerinden çıkarılmaktadır. Dizgi öznitelikleri [14] de yeniden adlandırma tekniklerinden etkilenmektedir, ancak bu etki yeni önerilen öznitelik setlerinden daha azdır çünkü dizgi öznitelikleri yalnızca kaynak kodlardan çıkarılan dizgileri değil, APK dosyalarına yerleştirilen tüm dizgileri kullanır.

- **AS11 - Veri setlerinde herhangi bir klon uygulama var mı? Bunlar Android Yİ üzerindeki performansı nasıl etkiler?**

  Veri setindeki uygulama klonlarını tespit etmek için Romadroid aracı [45] kullanılmıştır. Romadroid, karşılaştırılacak iki uygulamanın her bir manifest dosyasından bir dizgi oluşturur ve iki dizgi arasındaki benzerliği LCS algoritması kullanarak ölçer.

  Deneylerde %70 ve %90 eşik değerleri kullanılmıştır, Market, kötü amaçlı yazılım ve Genome veri setlerindeki 488, 153 ve 39 geliştiricinin uygulamalarının benzerlik skorları Romadroid kullanarak hesaplanmıştır. Sonuç olarak, uygulamaların ikili karşılaştırmasıyla $n*(n-1)/2$ benzerlik sonucu elde edilmiştir. Burada, n toplam uygulama sayısını belirtir. Bazı uygulamalarda hatalarla karşılaşıldığı için, her çift için benzerlik skorları elde edilememiştir. Birçok çalışma, tipik olarak, uygulama çiftlerinin benzerlik skorları %70 veya %90'ı aştığında bir uygulama çiftini uygulama klonu olarak kabul eder [43, 45, 46]. Bu çalışmadaki yaklaşımda, başka bir uygulama ile %70'in üzerinde benzerlik gösteren tüm uygulamaları hariç tutulmuştur.

  Tablo 6.14'da gösterildiği gibi, market veri setinin %20.6'sı ve kötü amaçlı yazılım veri setlerinin %22.3'ü %70 benzerlik oranının üzerinde benzer uygulamalara sahipti. Kötü amaçlı yazılım ve market veri setleri için sırasıyla Tablo 6.16 ve 6.15'de klon uygulamaların sonuçlar üzerindeki etkileri verilmiştir. Sonuçlar, özellikle market veri seti için, benzer uygulamaları çıkarmak doğruluk

ve F1 skorlarını iyileştirdiğini göstermektedir. Farklı geliştiricilerden benzer uygulamalar, modeli eğitim sırasında yanıltabilir, bu yüzden uygulama klonlarını çıkarmak sonuçlar üzerinde olumlu bir etkiye sahiptir.

SONUÇ

Bu çalışma, Android Yazarlık İlişkilendirmesi için yeni öznitelikler sunmakta ve çeşitli özniteliklerin kullanımını araştırmıştır. Ayrıca literatürdeki özniteliklerle karşılaştırma sağlanarak zararız, kötücül, versiyon ve karışıtırılmış gibi bir çok tür uygulama türünde sonuçlar alınmıştır. Sonuçlar yeni önerilen özniteliklerin Yİ probleminde önemli olduğunu göstermektedir. Gelecek araştırmalar, özellikle kötü niyetli uygulamalar üzerinde olanlar olmak üzere, klon uygulamalar için bu yaklaşımın kullanımını incelemelidir görüşündeyiz. Ayrıca, farklı kaynaklardan gelen kötü amaçlı yazılımlar arasındaki benzerlikleri bulmak önemlidir; böylece, kötü amaçlı yazılımların nasıl evrileceği tahmin edilebilir ve gelecekte ortaya çıkacak kötü amaçlı yazılımlara karşı yeni koruma teknikleri geliştirilebilir. Kötü amaçlı yazılımın yazarıyla ilgili öznitelikler, bu hedefe ulaşmak için yararlı olabilir. Bu nedenle, piyasadaki yeni uygulamalar, yazarlarının bakış açısından analiz edilebilir.

CONTENTS

Page

TABLES

FIGURES

Page

ABBREVIATIONS

LLM	:	**L**arge **L**anguage **M**odels
TPL	:	**T**hird **P**arty **L**ibrary
AST	:	**A**bstract **S**yntax **T**ree
CFG	:	**C**ontrol **F**low **G**raph
SVM	:	**S**upport **V**ector **M**achine
TFIDF	:	**T**erm **F**requency **I**nverse **D**ocument **F**requency
API	:	**A**pplication **P**rogramming **I**nterface
APK	:	**A**ndroid **P**ackage **K**it
LCS	:	**L**ongest **C**ommon **S**ubsequence
COPPA	:	**C**hildren's **O**nline **P**rivacy **P**rotection **A**ct
XML	:	**E**xtensible **M**arkup **L**anguage
UI	:	**U**ser **I**nterface
ART	:	**A**ndroid **R**untime
GPS	:	**G**lobal **P**ositioning **S**ystem
ANOVA	:	**An**alysis **o**f **Va**riance
RAM	:	**R**andom **A**ccess **M**emory
CPU	:	**C**entral **P**rocess **U**nit
NDK	:	**N**ative **D**evelopment **K**it

1. INTRODUCTION

Mobile devices play a pivotal role in our daily lives and facilitate tasks such as banking and communication from anytime and anywhere. These activities produce a considerable volume of critical data. Application developers already collect and use this data with our permission. But, can this data be collected without our consent? This concern becomes particularly relevant considering the activities of mobile malware writers. They can easily gather this data using malicious software they develop due to the carelessness of mobile users and the inadequacy of anti-virus systems.

This issue of unauthorized data collection is not just theoretical. A report by Kaspersky highlighted that Google Play, the main repository for Android applications, experienced over 600 million malware downloads in 2023 [47]. Data from the AV-TEST Institute's [48], which reported the existence of roughly a billion malicious applications in the same timeframe, further exacerbates this alarming statistic. Some developers also modify these malicious mobile applications to produce new variants of them. According to the same report from AV-TEST, only about 50 million of these were new, unique malware instances, with the remainder being modifications of existing malware. A significant 86% of Android malware utilizes the repackaging method, as stated in [49]. This method involves decompiling or reverse engineering a legitimate app, modifying its logic, inserting malicious code, and then repackaging it. Identifying similarities in malware from various sources is key to predicting its evolution and developing future protection techniques. Features associated with the authors of apps can be useful in achieving this goal.

Authorship Attribution (AA) is a process aimed at identifying the author of a computer program. Although it is primarily used to detect software theft and solve copyright issues, it is also used in digital forensics and malware analysis. Software developers generally leave footprints on their applications that describe their coding style. As a result, applications developed by the same developer exhibit strong similarities, and this

programmer-specific coding style is retained within the program binaries, as noted by Rosenblum [1]. Initially, these unique footprints were leveraged to pinpoint the authors of software applications. However, with the advent of software variants—modifications of the original software—research focus has pivoted towards the automated detection of software theft and copyright infringements.

One of the most promising applications of AA is in the identification of malware authors, as there has been a significant increase in the number of new malware and new variations in existing malware [2]. AA can pinpoint the creators of new malware, thereby facilitating the observation of malware evolution in the wild through the monitoring of its developers. Similarly, AA can mark new programs crafted by known attackers, making these programs prime candidates for further analysis. While there are numerous works on AA, its application in the realm of malware [3–5], particularly with regard to Android, remains insufficiently investigated.

There are two main approaches to AA: source code AA and binary code AA. At first, the researchers are focused on analyzing the source code of software to identify the author of unknown programs [6–10]. The preference for source code AA rather than binary code AA in the literature stems from the fact that many distinctive features in the source code are lost following the compilation process. The techniques employed in source code AA draw upon a diverse set of features, including but not limited to line length, average procedure length, method counts, naming conventions, comments, programming layout attributes (such as spacing and indentation), variable names, control flow structures, and aspects of the development environment like the computer platform, programming language, and compiler.

However, obtaining the source codes of applications, especially those of malicious ones, poses significant challenges due to the difficulties in accessing the original source code of software, particularly in the context of malware analysis. Therefore, in such cases, an application's binary code is the only source that can be used to identify the application's author [1, 11, 12]. Nevertheless, identifying authors based on the

binary code AA is much more challenging than identifying authors based on the source code. The compilation phase can significantly alter the binary code's structure through optimization techniques, thereby affecting AA's effectiveness. Valuable features for AA, such as linguistic and formatting elements, may be stripped away during compilation. Moreover, the dependency on specific compilers and their configurations further complicates binary code AA [13].

Despite these challenges, binary code remains a critical data source for malware AA due to the unavailability of source code. Additionally, the efficacy of popular AA methodologies, such as machine learning and deep learning techniques, largely hinges on the volume of training data available. Given that binaries are more readily accessible, binary code AA presents a compelling area of research, underscoring the need to adapt and refine AA techniques to work effectively with the binary representations of software.

Malware authors skillfully avoid detection by constantly testing their software with various dynamic analysis tools during development. These tools provide them with critical feedback, allowing them to adapt their software to bypass antivirus systems undetected. These individuals often distribute their malicious applications across both official and third-party Android markets. The fact that they distribute under various aliases in different markets further complicates the task of tracking their digital footprint. In such scenarios, AA helps identify the people behind these deceptive practices, even if they use multiple identities. Authorship attribution can also help address the challenge of detecting malware that evades static analysis by updating its malicious code at runtime [50]. This approach allows for the identification and tracking of malware creators based on their coding style and other unique markers, providing an effective method for combating threats that dynamically alter their behavior.

The fact that applications are typically developed by multiple authors presents an additional challenge. Researchers in the field of AA generally prefer to analyze works attributed to a single author rather than multiple authors. This preference underscores the complexity involved in determining the exact number of contributors

in a collaborative project. Therefore, accurately ascertaining how many authors have developed a particular application remains an open issue in the field of AA, especially when applied to malware analysis.

In conclusion, AA in software development is an area full of potential and challenges. As the dependence on digital technologies increases, it has become increasingly critical to accurately identify code authors for security, maintenance, or copyright enforcement.

The purpose of this thesis is to identify the authors of different types of applications. This is crucial, as some developers may upload applications that are not originally their own to application markets. By applying reverse engineering, they can access and change the source code of applications that do not belong to them. This process, known as repackaging, is a common practice, especially in the malware domain. Our method enables the detection of applications that seem to be written by a specific author but are in fact not. Consequently, our method can prevent copyright issues by identifying the original developer of applications that were developed for commercial purposes but then repackaged by another developer and uploaded to markets under a different name. Additionally, our method can be applied to student assignments. It helps in determining whether students have plagiarized code during development. Ultimately, this research not only addresses a significant challenge in software development but also offers a novel approach to upholding academic integrity.

The scenario involving malware developers is particularly noteworthy. They often receive indirect feedback by uploading early versions of their applications under different aliases to various platforms, such as alternative markets and online analysis tools. Utilizing this feedback, they devise strategies to evade antivirus systems. The method we propose addresses this cunning tactic. It allows for the analysis of any application uploaded to any environment. Importantly, our method can identify an malicious application as belonging to a known malware developer, even if it is not initially detected as malicious by conventional detection systems. Therefore, these

applications can be preemptively flagged as potentially malicious. This proactive approach offers a significant advantage to malware analysts and antivirus systems.

Additionally, this approach offers a significant advantage in malware research. It enables the clustering of versions of the same malware uploaded to markets at different times. This capability is crucial for a comprehensive understanding of the evolution of malware. Such information is especially useful in the evolving field of Android malware development.

1.1. Scope Of The Thesis

This study investigates the discriminating features of Android applications for AA. We have expanded the scope of investigation by collecting new features from the applications' components, including smali files, libraries, and permissions, as well as their metadata information available in the market. These newly gathered features, alongside those used in prior studies [14–16], are subjected to thorough analysis.

In our research, we specifically investigate the impact of different feature groups on AA. Considering that features proposed in the existing literature have been analyzed using different datasets, we established a common experimental framework to ensure a fair comparison. Our dataset, detailed in Section 5.2., encompasses benign, malicious, and obfuscated applications from a range of Android markets and studies.

Smali, serving as an intermediate language akin to *assembly*, preserves many insightful source code features such as variable and function names and code organization, yet offers greater readability than assembly. It is possible to decompile an Android application, called the Android Package Kit (*APK*) file, into Smali language. As a result, smali files that have a similar class and function name to Java/Kotlin files of the original source code of the application are obtained. Additionally, if any third-party libraries are used during development, smali files from these libraries are also generated. After decompilation, anyone can modify smali files easily and then recompile them into an *APK* file. The extraction of smali files from Android binaries opens up the

possibility of applying certain source code-based features, traditionally used in different programming languages, to Android binary applications.

Developers often tend to use the same third-party libraries due to their ease of use and familiarity. It is also shown that most applications use more than 20 third-party libraries [24] and a large part of the application code belongs to such libraries [25, 26]. Similarly, it is shown by [23] that most applications, on average, request 30% more permissions than they need. This is because developers use the same manifest file in their different applications. Consequently, analyzing which third-party libraries and permissions are utilized in an application can provide valuable insights for AA.

The features incorporated into our study, drawn from prior research [14–16], encompass configuration, (Dalvik Executable) DEX code, resources, and string-based elements. Each category of features is extracted from specific components of the *APK* files:

- **Configuration Features**: These are sourced from the manifest file of the *APK* files. The manifest file provides a wealth of information about the application's setup and permissions, making it a rich source for configuration data.
- **DEX Code Features**: Directly extracted from the *.dex* file, these features focus on the structural elements of the application, such as methods, classes, and field structures. The *.dex* file is being the compiled version of the application's source code.
- **Resource Features**: These are gathered from the "res" directory of the decompiled *APK* files. The "res" directory, which contains resources like images and layout files, provides insights into the visual and structural aspects of the application.
- **String-Based Features**: For this analysis, strings present in both the strings.xml file and the *.dex* file are utilized. By applying n-gram analysis to these strings, we can uncover patterns and sequences that are characteristic of the application's coding style and content.

An essential initial step in this study is to refine the dataset by removing very similar applications. It is important to note that many original apps may share common code segments due to the use of third-party libraries or the implementation of specific functionalities. Thus, it is crucial to be meticulous in selecting only those apps that are nearly identical. The main goal is to pinpoint and separate cloned applications present in the dataset. This aspect is especially crucial for Android applications, given that the ease of modifying these apps has resulted in an increase in cloned versions, particularly prevalent in alternative Android markets. Researchers are delving deep into this area not only to pinpoint the original authors of these applications, as cited in studies like [16], but also to detect distinct function blocks within the source code. Recently, the use of Large Language Models (LLMs) has become increasingly popular for identifying code clones [51, 52]. These models excel at analyzing specific portions of the source code.

We seek to find answers to the following research questions (RQs) on how our model identifies the author of Android applications, including benign and malicious applications:

- **RQ1** - *What is the performance of classification algorithms in solving the AA problem?*
- **RQ2** - *What is the ideal set size for n-gram features?*
- **RQ3** - *Does the use of TPL bring improvements in solving the authorship problem?*
- **RQ4** - *How effective is the proposed approach in identifying the developer of applications?*
- **RQ5** - *Does metadata of applications help to attribute the developer of applications?*
- **RQ6** - *What are the most important features for attributing applications to their developers?*

- **RQ7** - *Can we reduce the number of features without decreasing the performance of the model?*
- **RQ8** - *Does the number of applications per developer affect classification performance?*
- **RQ9** - *Does the proposed model successfully identify different versions of applications developed by the same author?*
- **RQ10** - *What is the effect of obfuscation on Android AA?*
- **RQ11** - *Are there any clone applications in the datasets? How do they affect the performance on Android AA?*

1.2. Contributions

In this research, we propose a novel and efficient approach for Android AA. The main contributions of this thesis can be summarized as follows:

- New features extracted from smali files were introduced for Android AA. To the best of our knowledge, the features inherited from the source code AA in smali codes were first used in Android in this study. The results showed that source code-based features increased the accuracy of attributing applications to their corresponding authors.
- In this study, TPL and permission features, previously used as string features in earlier research, have been utilized as binary features.
- Unlike traditional application distribution mechanisms, Android applications are centrally distributed in mobile markets. Therefore, in addition to the application code, we can obtain metadata about Android applications, such as application descriptions and user reviews. Consequently, this study explored the effect of textual metadata on Android AA. A positive effect of metadata-based features

was observed, especially when applied with the source code-based features employed in this study. This is the first study to use metadata descriptions in the realm of Android AA.

- The performance of the proposed approach was extensively analyzed. In addition to analyzing the effect of each feature set on Android AA, the studies proposed in the literature were compared. The experimental results show the positive effect of the features proposed in this study on authorship attribution.
- This study also explores version and obfuscated applications from the AA perspective, marking the first time analyses of version apps have been conducted.
- We collect a diverse range of applications with distinct characteristics, such as benign, malicious, versioned, and obfuscated. Our dataset is comprised of 10,385 benign Android apps, sourced from 488 unique developers, each contributing more than 10 apps, and gathered from five different markets. Additionally, it includes over 3,000 malicious apps from 153 developers, obtained from various sources. We also constructed approximately 6,000 obfuscated apps using the Obfuscapk tool [44], and over 1,000 applications for version analysis, both derived from the benign dataset.
- We shared the source code of our study with the research community at: `https://github.com/emreaydogan/SourceCodeBasedAAA`

1.3. Organization

This thesis is systematically organized into eight chapters. The organization of the thesis is as follows:

Chapter 1. introduces the research topic, outlining the scope, contributions, and overall structure of the thesis.

Chapter 2. provides a comprehensive review of the existing literature and research in the field of source and binary code AA, with a specific focus on its application

in malware, Android, and methods used for finding similarities between apps. This chapter incorporates an extensive review of code clone detection studies across varied domains, including Android and malware, thereby establishing a contextual foundation for our research.

Chapter 3. offers an in-depth background necessary for the fundamentals of Android architecture, the typical structure and features of Android applications, including those used in AppAuth [16] and StringAA [14], and proposed features for analysis. This chapter also delves into the Android app signing process and alternative Android markets, setting the stage for understanding the ecosystem in which AA operates.

Chapter 4. details the methodologies employed for attributing authorship in both source and binary code. It begins with a discussion on "features", highlighting their role in AA. This is followed by an examination of "representations", which focuses on the portrayal of data. The next section, "attribution models", sheds light on various models employed in authorship attribution. The chapter then transitions into a detailed exploration of "machine learning", segmented into three areas: "classification algorithms", "cross-validation", and "grid search cv". Each of these subsections contributes to a deeper understanding of the techniques applied in authorship attribution.

Chapter 5. introduces the methodology developed. It details the model architecture, dataset selection and preparation, methods for finding similar apps, the process of Android decompilation, and the intricate steps involved in feature extraction, including descriptions, processing methods, dimension reduction techniques, and the classification algorithms used, along with cross-validation and hyperparameter tuning strategies.

Chapter 6. presents the findings of the research. This chapter is structured around various research questions (RQ1 to RQ11), each addressing a specific aspect of Android AA, such as the performance of classification algorithms, the impact of n-gram set size,

the analysis of custom code versus all code including TPL, and several other factors that affect the effectiveness of the proposed approach.

Chapter 7. synthesizes the findings from the experiments and discusses them in the context of broader implications in the field. This includes discussions on the usage of native code, obfuscation techniques, challenges and insights gained from the dataset, and the phenomenon of clone/repackaged applications and applications developed by multiple authors.

Chapter 8. summarizes the key findings of the research, drawing conclusions from the experiments and discussions in the previous chapters. This chapter also provides recommendations for future research and the potential implications of this study in the field of Android AA.

2. RELATED WORK

Authorship Attribution is related to many different areas, such as literary work analysis [53, 54], code plagiarism detection [6], code AA [55], forensic investigation [56] and malware analysis [2, 57, 58]. It has different objectives such as authorship identification, clustering, evolution, profiling, and verification [27]. In the field of AA, two primary methodologies emerge: one that leverages source code analysis and another that focuses on binary code analysis. Although studies based on the source code of applications have appeared in the literature before, binary AA started to emerge due to the lack of and difficulty in gathering source codes. These studies are explained in detail in Section 2.1. and 2.2.

Since the early 2010s, mobile devices have begun to significantly affect many aspects of our lives, necessitating a careful examination of mobile applications. Android is the most used operating system on mobile devices; therefore, we shifted our focus to Android applications. We proposed new feature sets for Android AA that are based on the source code of Smali files, TPL, and metadata information. We also compared our feature sets with those of the literature. Studies in the field of Android AA are detailed in Section 2.3.

Most studies have primarily focused on desktop and benign applications. However, with the evolving landscape of cyber threats, understanding the authorship of malware has emerged as a critical area of concern in cybersecurity. This shift is not just about expanding the scope of AA but also about addressing unique challenges in tracing sophisticated and often concealed malicious code origins. Therefore, we explore AA in the context of malware to understand its specific implications and applications. Studies about malware AA are detailed in Section 2.4..

In order to build an AA model, unique applications are needed for each author in the dataset used. In the realm of Android, it is common for apps to be modified and then re-uploaded to markets in repackaged and obfuscated forms. Such practices can

introduce “clone applications” into the dataset. Overfitting occurs when a model learns the details and noise in the training data to such an extent that it negatively impacts the performance of the model on new data [59]. Therefore, it is essential to identify and eliminate these clones to ensure that the AA model remains accurate when applied in real-world scenarios, which may present different challenges and data characteristics compared to the controlled environment of the training phase. Studies on finding clone applications are explained in detail.in Section 2.5.

2.1. Source Code Authorship Attribution

The source code of software can provide very useful features to identify the author of unknown software. The unique coding style of the author is evident in the source code. Given that software created by the same developers often exhibits shared characteristics, certain features can be identified by examining the coding style of the developer.

N-gram model is used by researchers for source code AA [6, 8]. Both studies employ information retrieval approaches with n-grams. Burrows et al. [6] address the issue of plagiarism and copyright infringement in academic and corporate settings, emphasizing the limitations of current methods that rely solely on textual similarity comparisons for detecting infringements. The authors propose using AA as a tool for identifying plagiarism. By analyzing a collection of 1,640 documents from 100 authors, they demonstrate the ability to correctly identify the author of a queried work in up to 67% of cases. Frantzeskou et. al. [8] propose a method for determining the most likely author of a computer program from a set of predefined candidates. They aim to trace the origins of the code used in cyberattacks. They conducted experiments using data sets in different programming languages (Java and C++) and with varying numbers of candidate authors (6 to 30). Their method performs well even with limited and short programming samples available for each programmer, a scenario often encountered in cybercrime investigations.

Kothari et al. [9] create a profile for each known developer by computing two types of metrics. While the first type is style-based, such as line size, leading spaces, and tokens (words) per line, the second type is based on patterns of character sequences. Their aim is to determine the closest matching profile for an unidentified source code. This paper introduces a method for identifying the authorship of source code, a tool that is particularly useful in situations where code ownership is disputed, such as in cases of plagiarism or intellectual property infringement. This method is also valuable for identifying the creators of malware. The approach involves first building profiles for a known group of authors using verified code samples. These profiles are constructed by computing a set of specific metrics. Then, the same metrics are computed on unidentified source code to find the closest matching profile from the known authors. The effectiveness of this method is demonstrated through a case study involving two different types of software: one created by open-source developers working on various projects, and the other by students completing assignments with identical requirements. The results of the case study are promising, showing over 70% accuracy in correctly identifying the author when choosing the single nearest match and more than 90% accuracy when selecting the top three closest matches.

A back propagation neural network based on particle swarm optimization is employed using Java source codes in [60]. They extract 19 features consisting of lexical, layout, structure, and syntax metrics. Alsulami et. al. [61] implement deep neural networks using Long Short-Term Memory (LSTM) and Bidirectional Long Short-Term Memory (BiLSTM) models based on Abstract Syntax Tree (AST) of source code.

Dauber et. al. [62] investigate the attribution of small and incomplete source code fragments. They discuss the privacy implications of program AA, especially for programmers who prefer to contribute code anonymously. Previous research in this field has primarily focused on attributing authorship to complete files that are individually authored, often using ideal datasets such as those from the Google Code Jam. However, this paper expands the scope of research by exploring AA “in the wild”, specifically looking at source code from open source version control systems.

Burrows et. al. [63] compare the AA techniques in terms of source code. They stated that the comparison between these techniques is not fair since they use different test sets and evaluation methodologies. It categorizes existing methods based on the software features (n-grams or software metrics) and the classification technique (information retrieval ranking and machine learning). The study aims to provide a direct comparison of these methods by testing them on identical source code collections across different programming languages and author types. Key findings include: ranking methods achieving around 90% accuracy and machine classifiers about 85% for a one-in-10 classification problem; neural networks and support vector machines being the most effective machine classifiers; potential in combining n-gram features with machine classifiers, despite scalability challenges; and information retrieval techniques currently surpassing machine learning methods in accuracy.

Frantzeskou et. al. [64] proposed five categories for source code AA by expanding the taxonomy developed by Zheng et. al. [65]. The process of *Authorship Identification* focuses on establishing if a specific author wrote a given piece of code. *Authorship Characterization*, on the other hand, analyzes the programming style and techniques to infer certain attributes of the programmer, such as their cultural and educational background or language proficiency. *Plagiarism Detection* involves comparing different sets of source code to identify similarities, aiming to uncover instances of plagiarism, defined as using someone else's work without proper credit. *Author Intent Determination* is concerned with understanding whether a code-caused malfunction was intentional or an accidental error. Lastly, *Author Discrimination* assesses if various code segments were created by one or multiple authors. Those problems can be solved by manual analysis which requires a security expert, similarity calculation which uses numerical values, statistical analysis and machine learning algorithms.

2.2. Binary Code Authorship Attribution

When the source codes of applications are unavailable, particularly in the realm of malware, the binary codes of applications serve as a valuable alternative. While most works use the pure binary of applications to extract features, some works, including ours, decompile the binary of applications and then extract features.

Rosenblum et. al. [1] introduced an innovative method for representing programs and techniques that can autonomously identify the stylistic characteristics of binary code. They aim to find stylistic similarities between programs and identify the authors of the programs. It is the first work to automatically determine the authors of software. They extracted syntax- and semantic-based features such as idioms, n-grams, and graphlets. They then rank the features to eliminate irrelevant features by computing the mutual information between the feature and the actual author label. They applied machine-learning techniques to these selected features. They claimed that the most prominent features are distinctive to each author and reflect the way authors write code.

Alrabaee et al. [12] propose a layered method, called OBA2, for binary AA. They extract semantic-based and syntax-based features. They stated that the unique features of each author were not connected to the author's coding style, in contrast to Rosenblum's inference. They conduct an experiment based on three levels: removing unrelated code, a syntax-based attribution layer, and a semantics-based attribution layer. Their results are compared with Rosenblum's work [1] and they stated that their results produce more accurate results than Rosenblum's work.

Caliskan et. al. [66] also proposed an approach that automatically identifies the authorship of software binaries, but they use syntactical features located in the source code of decompiled binary. Their approach consists of four steps: disassembly, decompilation, dimension reduction, and classification. They extract instruction-based features in the disassembly phase and lexical features in the decompilation phase. They

also obtain AST and CFG by employing fuzzy parsing. They then conduct information gain and correlation-based feature selection to select the top features. Finally, random forest classifiers are applied to these top features to determine authorship. They also compared their results with Rosenblum's work [1]. They claimed that their approach is robust to basic obfuscation techniques.

2.3. Android Authorship Attribution

In recent years, mobile devices have overtaken desktop devices in terms of usage, popularity, and sales. Because our study aims to find the authors of malicious and benign applications in the Android environment, we explain related studies on Android AA. Moreover, we compared our newly proposed feature sets based on the source code of the smali files, metadata, and TPL with the feature sets used in the studies given below.

A study based on string analysis of *APK* files was proposed by Kalgutkar et al. [14]. Three types of strings were employed: the string identifier list in the *.dex* file, all string components presented in the *.dex* file, and strings extracted from the strings.xml file. Subsequently, n-gram analysis was employed on these string lists to generate a feature vector representing an author. Finally, the authors were classified using Support Vector Machines (SVM). The proposed approach was evaluated in different datasets that consisted of benign, malicious, and obfuscated applications, and accuracies of 98%, 96%, and 71% were obtained, respectively.

Gonzalez et al. [15] proposed an AA method based on array-related, array-unrelated, and n-gram features by retrieving bytecodes from the *.dex* file and converting them into a smali representation. Their framework consisted of two phases: profile construction and incremental analysis. In the first phase, author profiles were constructed using the Random Forest algorithm. In the second phase, incremental analysis is applied to attribute new applications to existing profiles and find new possible profiles for applications that have not yet been attributed to the existing authors. They achieved

97.5% accuracy in a dataset with 33 authors and 1428 applications. They also applied their approach to more than 131,000 applications collected from various sources to find applications belonging to author profiles in the wild.

Guoai et al. [16] proposed a new approach, AppAuth, that detects the original author of a group of given repackaged Android applications with common properties such as label names, icons, similar package names, and even file sizes. Several coding style-related features were extracted from the *APK* files: i) binary features from the *.dex* file; ii) resource features from decompiled files; and iii) configuration features obtained from the manifest file. They successfully identified 69 original authors of 75 clone pairs (92%) in the wild. They also analyzed the impact of TPL on the results. They stated that removing code-level features introduced by TPL slightly increased the performance of their framework. They also evaluated the prediction performance of independent developers and development teams. However, their main drawback is that if the apps they compare are repackaged implementations of an original app, they find a fake author instead of the original author.

Wang et al. [17] extracted three types of string-related features: DEX-based, manifest-based, and lib-based features. In the *.dex* files, identifier names, instruction sequences, and the use of Android APIs are collected. They extract the names of activities, providers, services, and broadcast receivers and use features from the manifest file. Finally, they obtained the names of the TPL used in the applications. They compared the CountVectorizer, TFIDFVectorizer, and word2vec models to convert features into the author's profile vectors. Because the word2vec model yielded the best result, it was chosen for further evaluation. Then, three machine learning models (i.e., Linear SVM, Random Forest, and Logistic Regression) were applied to these vectors to classify authors on different datasets containing benign, malware, and obfuscated applications. The results show that the proposed approach achieved 92.5% accuracy on average for the entire set. Moreover, it outperformed AppAuth in 2,900 non-obfuscated applications, thereby improving authorship identification by 3.4%.

Differing from earlier studies, our approach incorporates metadata information from the applications' market pages. Additionally, we have compared all the feature sets in the field of Android AA in the literature, utilizing a common dataset to ensure a fair comparison. Our dataset is also larger than those used in other studies, encompassing a comprehensive range of both benign and malicious applications. The comparison between the studies mentioned above and our study is illustrated in Table 2.1.

2.4. Malware Authorship Attribution

Most studies, such as those mentioned earlier, have mainly used desktop and benign applications. However, malware analysts are also keen to find the original author of a malicious application.

Kalgutkar et al. [27] summarized various methods for both source and binary code AA, mainly from the perspective of malware domains. They state that such attribution is not just about naming the author but extends to understanding malware's writers' methods. Knowing who wrote a piece of malware can provide insights into the types of tools and techniques the author typically employs, as well as how the malware is likely to evolve.

It is mentioned that source and binary code AA face challenges due to the complexity and volume of modern malware, making these methods time-consuming and labor-intensive [67]. To overcome these limitations, the paper proposes a model for malware AA that utilizes automated analysis for rapid feature extraction and analysis. This method employs tools to analyze malware files and specific hash values without the need for expert intervention, making it significantly faster than traditional methods. The study conducted experiments using various machine learning classification algorithms on six different malware author groups.

Attackers employ evasion techniques such as encryption and obfuscation to hide their malicious intentions, which introduces new challenges to the AA problem. Only a few studies have been published on the identification of malicious authors. Layton

Study	**A3IDENT**[17]	**String Analysis**[14]	**AppAuth**[16]	**Authorship Attribution of Android Apps**[15]	**Our Study**
Dataset Size	F-Droid:416 apps 140 devs Benign:686 apps 17 devs Malware:14,564 apps 100 devs Obfuscated:2,900 apps 29 dev	Benign:1,559 apps 40 devs Malware:262 apps 10 devs Obfuscated:96 apps 9 devs	3,871 apps 273 devs	35000 apps	Benign:10,385 apps 488 devs Malware:3,268 apps 153 devs Obfuscated: 6,055 apps 320 devs
Methodology	Package Relation Graph Package Aggregation Package Clustering	Word level string n-grams	Coding-style feature extraction	Profile construction Incremental analysis	Smali codes, n-gram, metadata coding-style based analysis
Specific Focus	Malware apps	Lightweight, effective across app types	Cloned apps	Author profile creation Opcode analysis	Obfuscated, Version apps Metadata Analysis
Machine Learning Technique	SVM Random Forest Logistic Regression	SVM	LightGBM	Random Forest	Random Forest K-Nearest Neighbors SVM Gaussian Naive Bayes LightGBM
Contribution	Authorship decoupling dex, lib, manifest features Comparison with AppAuth	Comparison of opcode and bytecode features String component analysis	App repackaging detection dex, resource, configuration features	Attribute unknown apps to well-known profiles Opcode analysis	Comparison of different studies Source-code based features App Description Analysis

Table 2.1 Comparative metrics of Android authorship attribution studies

et al. [3] analyzed the source code of the Zeus botnet to determine the number of authors involved in developing malware and their roles. The evolution of the Zeus botnet over time was also analyzed in this study. Internet Relay Chat (IRC) messages were analyzed to match them with known aliases, which are real users hiding behind aliases [5]. The motivation for this study is that IRC messages frequently facilitate cybercriminal activities, including the sale of stolen credit card information, botnet access, and malware in online chat rooms.

Alrabaee et al. [2] compared existing studies based on benign binaries in the literature [1, 11, 12] and conducted an experiment to investigate the effect of their models on real malware. The study in [11] achieved better accuracy than the other two studies on a benign dataset. The standard k-means clustering algorithm was used to compare the three studies in the malware dataset. The study given in [1] performed the worst among them, whereas other studies achieved similar results. It is stated that the Rosenblum approach utilizes a blend of compiler and user features as its top-ranked features, leading to an increased occurrence of false positives.

2.5. Finding Similarities Between Applications

Before conducting authorship analysis, we need to be sure that the dataset does not include any similar, repackaged, or piggybacked apps. it is crucial to ensure the integrity and uniqueness of the dataset to avoid skewed or inaccurate results. A repackaged app typically involves the unauthorized modification, repackaging, and redistribution of a legitimate application, done without the consent of the original developer. On the other hand, a piggybacked app emerges when a malicious entity modifies a legitimate, and often popular, mobile application by incorporating malicious code. While the functionality of the original application is usually preserved to avoid suspicion, added code introduces harmful or unwanted features. Like piggybacked apps, repackaged apps are often distributed through third-party app stores or other channels

outside the official app stores. Users may mistakenly download these apps, thinking they are getting the original version.

A framework called SimiDroid is introduced that identifies and explains similarities in Android applications in [43]. SimiDroid consists of three plugins, namely MPlugin, CPlugin and RPlugin. Each plugin is performed using different kinds of features: method-based, component-based, and resource-based, respectively. They also pointed out what kinds of changes have been made among app versions and among repacked apps. They set up a scenario that composes benign apps and their piggybacked counterparts as a pair. They managed to identify the similarity scores of 0.9996, 1, and 0.8661 of pairs for each feature, respectively. They stated that the manipulation of resource files was much easier than others; therefore, the detection of resource-based approaches was a bit lower. They also compared themselves with other approaches, AndroGuard [68] and FSquaDRA [69], and outperformed them.

RomaDroid [45] is a system that efficiently identifies cloned apps on Android markets by analyzing the hierarchical tree structure of an app's manifest file, implicit intents, and component information. The system was tested using 148 original apps and their 620 obfuscated versions, created with popular obfuscators like ProGuard, Dex-Guard, and DashO. RomaDroid showed low false negative rates and minimal processing overhead. It leverages simple features from the AndroidManifest.xml file of each *APK* package, such as the tree structure of the manifest file and the class names of components associated with implicit intents. The Longest Common Subsequence (LCS) algorithm is employed for high accuracy in detecting cloned apps. The system is compared with SimiDroid, another app clone detector, which focuses on methods and component names for similarity scoring. RomaDroid focuses on the AndroidManifest.xml file, enabling fast generation of feature information and efficient app comparison. It does not address the detection of illegal use of parts of an app or TPL. The technique does not require source code, making it applicable to most apps.

3. ANDROID BACKGROUND

Android is the most widely used mobile operating system (OS) based on the Linux kernel, with a usage of 70.76% in October 2023 worldwide [18] and a cornerstone in the mobile industry. It offers an official market, Google Play [19], for developers to easily deploy their applications. It also allows developers to upload their applications to alternative markets such as Aptoide [20] and Apkmirror [21]. Therefore, Android applications, namely *APK* files, can be easily gathered from such markets. Android apps are developed using either Java or Kotlin programming languages. They are then compiled into a bytecode for the Java virtual machine. Finally, these Java bytecodes are translated into Dalvik bytecodes and stored in *.dex* files. The Dalvik bytecode corresponds to hexadecimal sequences of executables for Android; hence, it is the format that Android understands. The Dalvik bytecode is difficult to read or modify. Hence, the smali intermediate language is generally used to look inside class files. Smali is a human-readable representation of the Dalvik bytecode. Smali files can also be extracted directly from the *APK* files.

An Android application, an *APK* file, consists of several elements such as *.dex* files, a manifest file, resource files, and string files. All of these files can be used to gather information regarding the application. For example, while the manifest file contains information such as the number of services, activities, and permissions used in applications, we can find images, libraries, and XML files and folders used in applications in resource folders. String resources provide text strings for applications [22]. There are three types of strings: String, String Array, and Quantity Strings (Plurals). All strings were capable of applying styling markup and formatting arguments. The *.dex* file contains Java/Kotlin class files. It is possible to convert *class* files to *smali* files, which are intermediate representations of *.dex* files. Even though some linguistic and formatting features of a source code are lost when compilation is carried out, unlike assembly-based executables, Dalvik-based executables contain some insightful information in the smali files. Fig. 3.1 shows a sample Java/C++ compatible

```java
int find_maximum(int a[], int n) {
  int c, max, index;
  max = a[0];
  index = 0;
  for (c = 1; c < n; c++) {
    if (a[c] > max) {
      index = c;
      max = a[c];
    }
  }
  return index;
}
```

(a) Java

```java
public int find_maximum(int[] a, int n) {
    int max = a[0];
    int index = 0;
    for (int c = 1; c < n; c++) {
        if (a[c] > max) {
            index = c;
            max = a[c];
        }
    }
    return index;
}
```

(b) Decompiled Java

```
.method find_maximum([II)I
    .registers 7
    .param p1, "a"    # [I
    .param p2, "n"    # I
    .prologue
    .line 261
    const/4 v3, 0x0
    aget v2, p1, v3
    .line 262
    .local v2, "max":I
    const/4 v1, 0x0
    .line 263
    .local v1, "index":I
    const/4 v0, 0x1
    .local v0, "c":I
    :goto_5
    if-ge v0, p2, :cond_11
    .line 264
    aget v3, p1, v0
    if-le v3, v2, :cond_e
    .line 265
    move v1, v0
    .line 266
    aget v2, p1, v0
    .line 263
    :cond_e
    add-int/lit8 v0, v0, 0x1
    goto :goto_5
    .line 269
    :cond_11
    return v1
.end method
```

(c) Smali

```
push   %rbp
mov    %rsp,%rbp
mov    %rdi,-0x18(%rbp)
mov    %esi,-0x1c(%rbp)
mov    -0x18(%rbp),%rax
mov    (%rax),%eax
mov    %eax,-0x8(%rbp)
movl   $0x0,-0xc(%rbp)
movl   $0x1,-0x4(%rbp)
jmp    0x400555 <find_maximum+98>
mov    -0x4(%rbp),%eax
cltq
lea    0x0(,%rax,4),%rdx
mov    -0x18(%rbp),%rax
add    %rdx,%rax
mov    (%rax),%eax
cmp    -0x8(%rbp),%eax
jle    0x400551 <find_maximum+94>
mov    -0x4(%rbp),%eax
mov    %eax,-0xc(%rbp)
mov    -0x4(%rbp),%eax
cltq
lea    0x0(,%rax,4),%rdx
mov    -0x18(%rbp),%rax
add    %rdx,%rax
mov    (%rax),%eax
mov    %eax,-0x8(%rbp)
addl   $0x1,-0x4(%rbp)
mov    -0x4(%rbp),%eax
cmp    -0x1c(%rbp),%eax
jl     0x400517 <find_maximum+36>
mov    -0xc(%rbp),%eax
pop    %rbp
retq
```

(d) Assembly

Figure 3.1 Sample code fragments

code fragment, its corresponding smali, and assembly codes, respectively. While the smali code preserves information, such as variable names and method signatures, the assembly code loses such information after the compilation process. Therefore, some features related to source code AA can be obtained from smali files. Therefore, the

effects of such features were explored for the first time in this study.

3.1. Android Application

We have segmented Android applications into various components, each corresponding to the different feature sets utilized in our study.

- **Source Code**: Smali files contain Dalvik-byte/smali codes generated from Java bytecodes in the class files. Smali codes can be thought of as assembly codes for a C/C++ program. They are generated from a high-level language and converted to machine code via Android Runtime (ART) for smali. The operating system then executes these machine codes to run the applications. In Fig. 3.1, the smali and assembly codes corresponding to a Java function named *find_maximum* are shown. As seen in the Fig. 3.1, the names of the function parameters and global and local variables in the original source codes are not preserved in the assembly files but in the smali files. For example, the local variables *n, max, index* and function parameters *a, n* are preserved in the smali code given in the figure. It is shown that developers tend to use the same format in naming identifiers, such as using $ character or digits in variable names. Therefore, preserving such names in decompiled code can help identify developers.

- **Permission**: Permissions are required to use the system resources of mobile devices. Applications can only access and use device resources, such as cameras and GPS, via the permissions requested in the manifest file. While normal permissions can be granted automatically, dangerous permissions require user approval. Most applications request more permissions than they use. According to a study that evaluated the gap between the requested and the permission used at runtime, applications use, on average, 30% more permissions than they need [23]. Although a developer might develop applications belonging to different categories, they could use the same manifest file in these applications because of its convenience. Moreover, because they use the same TPL in applications

developed by the same author, they need to use the same permissions that these libraries require.

- **Library**: Developers prefer to use TPL to provide some functionalities in their applications rather than implementing them from scratch. It is shown that most applications use more than 20 TPL [24] and a large part of the application code belongs to such libraries [25, 26]. For the same functionality, developers tend to use the same library that they are familiar with.

- **Metadata;** The distribution of Android applications through centralized mobile markets differs from conventional methods of application distribution. In addition to the application itself, these markets include information about the application, such as the application description, application rating, and user reviews, known as metadata.

- **Configuration**: Android applications are composed of four key elements: activities, services, broadcast receivers, and content providers. Each component has its own purpose and life cycle, which define how it is created and destroyed. Activities define how users interact with applications. They are used to define user interfaces that allow users to use the system resources. Every single application screen refers to activities on Android. Some activities may trigger other activities. Services perform long-term operations in the background and do not contain a user interface. Broadcast receivers allow an application to be linked to a system or an application event. The Android operating system notifies the connected application when an event is triggered. Content providers are used to access data in the application databases. An application can also use these to share its database with other applications. Thus, a single piece of content can be distributed across multiple applications. All the components are declared in the Android manifest file. Developers tend to reuse the same Android manifest file when developing their applications. Therefore, information in the manifest file,

such as the number of components, can help distinguish developers from each other.

- **Resource**: Android applications may contain resource files such as images, sounds, icons, and native libraries stored in the *res* directory. The files can be used for various reasons, such as language support and the provision of images for UI. Some files in the *res* directory, such as dynamic libraries, database files, and payload files, can be accessed at runtime. Developers embed some specific and critical information into these files. Therefore, it is important to analyze the characteristics of this folder.

- **DEX Code**: The *.dex* file contains information about the structure of the applications, as shown in Fig. 3.2. AppAuth [16] mainly analyzes the data section in *.dex* files and focuses on methods, classes, and field structures, as well as annotation, interface, debug information, etc. Because application sizes might vary significantly, instead of obtaining the numerical values of these features, AppAuth use ratio values, such as the ratio of the number of abstract classes to all classes.

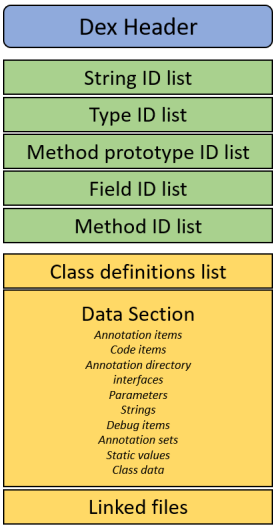


Figure 3.2 Format of a *.dex* file

- **String**: The strings in the strings.xml file are extracted line-by-line as app string features. Strings in the strings.xml file contain references to the source code or other resource files of the applications. These are static strings shown to users, such as the application's name. The *.dex* file consists of different parts, such as the string id list, type id list, and method id list. The string id list mainly contains the strings used in the source code of an application. However, malware developers put their payloads on the string ID list to run them at runtime and avoid analysis. Strings in the string id list part of a *.dex* file are extracted as DEX string features.

3.2. Android Application Signing Process

To release an Android app to official and alternative markets, the developer must sign the app with a certificate of its own. Therefore, most studies on Android AA in the literature assume that if apps share the same signature, they are written by a developer who has the signature. Applications are organized based on their signatures, with those sharing identical signatures being classified into the same group.

The Android Application Signing Process is a critical component of Android application development and distribution that ensures the integrity and authenticity of Android applications. It involves creating and managing cryptographic keys that sign the application's code and resources, verifying its origin and verifying that it has not been tampered with during deployment. Android's official app store, Google Play, encourages developers to use the Google Play App Signing service, which allows Google to manage the app signing key on the developer's behalf. This process not only simplifies key management but also increases security. Google Play also makes it easier to safely distribute app updates by ensuring that the app's signature matches the signature on file. Understanding and complying with the Android Application Signing Process is crucial to maintaining users' trust and protecting against unauthorized changes to Android applications [70].

Developer certificates are necessary to establish trust between applications published by application developers. They prevent attackers from installing malicious versions by ensuring that an application's digital signer is the same person who signed the signature. Android enables data access and process sharing by allowing application packages signed with the same certificate to run in the same process. Additionally, Android provides signature-based permission enforcement, allowing apps to expose functionality to another app signed with a specific certificate. Developer certificates do not need to be issued from a widely trusted certification authority (CA), and trust between developers and end users is established through registration. Android applications are digitally signed with two types of signatures: those applied to the *APK* using the signing key, and those applied to certificates corresponding to the *APK*s' signing keys. Google has made several requirements and recommendations for certificates, including two public key encryption algorithms and the validity period of the public key [71].

APK files are distributed through app markets such as Google Play and alternative app stores such as Huawei [72] or Tencent [73]. Each packet must be cryptographically signed to ensure integrity and authenticity. Google's signature scheme has been revised over time, and certificates are generally self-signed. However, Android does not have a trusted authority to verify the validity and accuracy of certificates. Since 2017, the Google Play Store has introduced the "Play App Signing" feature to prevent signing keys from being lost or compromised. This service further restricts the limited function of the signing certificate as an indicator of authorship, as applications can be signed by Google itself. In [74] stated that alternative Android markets enforce their own policies, such as uploading a pre-signed *APK* to APKMonk [75], Baidu [76] or APKMirror [21].

3.3. Alternative Android Markets

Alternative Android marketplaces such as APKMirror [21] and APKPure [37] are popular for offering a wider range of apps than the Google Play Store, including older versions of apps not available anywhere else. They appeal to users looking

for specific features or applications that are not available in their region. However, while these markets expand options, they also bring risks. Unlike the Google Play Store, which has strict security controls, these alternative platforms may not have the same level of scrutiny. This may raise concerns about the safety and security of some applications. Users who trust these sites generally need to be more careful and make sure they download safe and reliable applications. Despite these concerns, these alternative markets remain valuable resources for those looking for unique or hard-to-find applications.

4. AUTHORSHIP ATTRIBUTION

This chapter delves into the details of Authorship Attribution, a key aspect of our research. We begin by explaining various "Features" used in authorship attribution, discussing their relevance and how they contribute to identifying authorship patterns. Following this, we delve into "Representations", examining how features are represented in the attribution process. The chapter then shifts focus to "Attribution Models", where we outline and analyze different models used in the field, highlighting their strengths and limitations in the context of authorship attribution. A final portion of the chapter is dedicated to "Machine Learning", which is subdivided into three subsections. Firstly, "Classification Algorithms" are discussed, providing insights into the various algorithms employed. Next, we explore "Cross-Validation" methods, emphasizing their importance in ensuring the robustness and generalizability of our models. Lastly, "Grid Search CV" is examined as a method to fine-tune model parameters for optimal performance. Throughout this chapter, we aim to provide an overview of the methodologies and techniques applied in the domain of authorship attribution.

Authorship attribution aims to determine the original creator of a text or software. In the literature [27, 28], the process of attributing authorship in software follows a four-step approach, as depicted in Fig. 4.1. The process of AA utilizes both binary and source code. The code's characteristics, encompassing lexical, syntactic, semantic, behavioral, and application-specific features, are extracted from the code for analysis. Various representations of code are employed, including tokens, strings, n-grams, idioms, graphs, and trees. Models for authorship can be categorized as profile-based, example-based, or hybrid. These models are supported by methodologies such as similarity-based, vector space, probabilistic, and meta-learning methods. The final point of this process leads to outcomes like attributing authorship, detecting plagiarism, and identifying author intentions. Each step is explained in detail below.

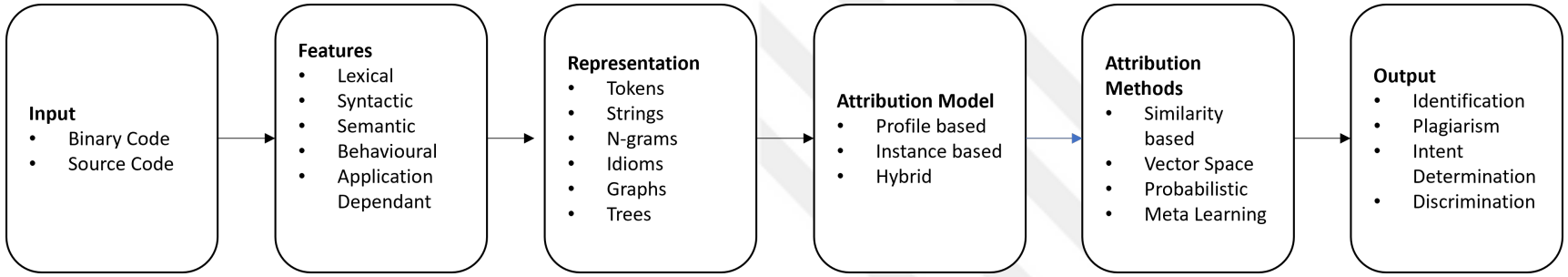


Figure 4.1 The steps of Authorship Attribution

4.1. Features

Kalgutkar et al. [27] and Gonzalez [28] provide a detailed explanation of each feature, along with their respective advantages and disadvantages, which are explained below.

Lexical features are derived from either source or binary code when this code is processed as a basic series of tokens or characters. Lines of code, operands, variables, spaces, and the frequency of words, tokens, characters (n-grams), and function names can be named for lexical features. These features are independent of any specific programming language, allowing the techniques used for their extraction to be readily adaptable across various programming languages and environments. However, their effectiveness is related to the quantity of training samples and the selection of the right features. Lexical features are susceptible to modification by code formatters, which can impact the dependability of these features. Additionally, developers of proprietary software and malware may use methods like renaming, addition or removal of redundant code, string obfuscation, and string encryption, all aimed at circumventing systems designed for code attribution.

Syntactic features have a significant weight in determining the author. Features like average function size, the use of custom macros, choices in data and control structures, and types of statements (input, conditional, and assignment) are considered syntactic features. These features can reflect an author's approach to problem-solving and contribute to creating robust author profiles. While syntactic features are more resistant to code formatters and obfuscators than lexical features, they also face other challenges, including language dependency and difficulty in feature selection. Furthermore, altering the structure of the code can neutralize these features.

Semantic features focus on the logic behind the code. Their resistance to code transformation and obfuscation techniques surpasses that of other features. Semantic features consist of aspects like loop usage, data and control flow analysis, and the application of specific algorithms that indicate an author's unique style. The primary

objective of utilizing semantic features is to capitalize on the concept that authors generally exhibit consistent logic in their applications. While an author's programming style may vary across different programming languages, the underlying problem-solving techniques, which are a reflection of the author's unique approach, tend to remain relatively constant and are challenging to alter. However, these features can be complex to extract and are vulnerable to advanced obfuscation techniques such as code optimization, string obfuscation/encryption, data transformation.

Behavioral patterns of software, such as system calls, file accesses, mutex creation, URL visits, dynamic value generation, and network connections, offer another way for author identification. These features are resilient against software protection and sophisticated code obfuscation. While revealing hidden functionalities, their reliance on data size and vulnerability to anti-analysis techniques can lead to high rates of false positives and negatives [27].

Application-dependent features, derived from sources like log files, property files, manifest files, and resource files, also aid in AA. This method is cost-effective and useful even without access to the source code. It facilitates the development of author profiles but requires advanced tools for feature extraction and is not immune to code obfuscation techniques [27].

4.2. Representations

It is important to transform the mentioned features into appropriate data structures so that they can be better handled and analyzed. The following section provides details on these data structures.

- **Tokens** represents the smallest meaningful pieces obtained from the source and binary code. For example, a character, word, keyword, operator, identifier, and basic block.

- **Strings** refers to specific text strings within the code. Lexical and syntactic features can be combined to form strings. However, they are unable to reveal the semantic meaning of the code.

- **N-grams** are sequences formed by combining n consecutive elements. This is used to capture structural features of the code. In contrast to tokens and strings, N-grams assist in gathering both contextual and neighboring information. However, the optimal choice of n value depends on the dataset and features, and there is no definitive method to determine its value. This requires extensive research to determine the optimal value.

- **Idioms**, which are language-specific features, usually consist of 3-6 instructions used in various programming languages. These idioms can be extracted from the assembly language of binary code. Unlike string, tokens, and n-gram attributes, it is more resistant to obfuscation techniques because it allows the use of a wildcard.

- **Graphs** represents the structure of the code and the relationships between elements. Semantic meanings are derived from these features, making them resistant to obfuscation. This involves a structure of nodes and directed edges, where nodes represent basic blocks or functions and edges indicate the relationships between functions, akin to a call graph or control flow graph.

- **Trees** are especially used for syntactic analysis; they show the hierarchical structure of the code as a tree structure. These structures are akin to graphs, which possess an acyclic and undirected nature. Examples of such structures include the Abstract Syntax Tree and the Parse Tree. An AST is a tree representation of the abstract syntactic structure of source code written in a programming language. Parse Tree, also known as a Concrete Syntax Tree, is a tree representation that reflects the syntax of the source code according to the grammar of the programming language. Unlike the AST, the Parse Tree contains every detail in the source code, including parentheses and unnecessary syntactic elements.

- **Embeddings** are combinations of different representations when a single representation is insufficient to capture the meaning of a program. For example, strings, n-grams, and trees can be embedded together inside a vector.

4.3. Attribution Models

Authorship attribution typically involves a multi-class, single-label categorization task where there is a set of training samples from a finite number of identified candidate authors, with each class representing one such author. Two tasks can be executed: 1) a task that new author can only be attributed to previously known authors and 2) a task that involves identifying and classifying new authors, regardless of whether they have been encountered before. The choice between profile-based and instance-based attribution models hinges on whether the training samples are considered cumulatively or individually.

In the Profile-Based model, each author is assigned a unique and distinct style of representation that encompasses all their applications. This model does not consider variations between different works by the same author as the style of the author is manifested through a series of shared features.

Contrastingly, the Instance-Based model generates separate styles for each individual app. Here, each work by the same author is considered a separate entity. This approach allows differences between different works by the same author to be taken into account. When determining the authorship of an unknown sample, it is compared to every sample in the corpus.

4.4. Attribution Methods

After designing the attribution model, the subsequent phase involves choosing an appropriate method for comparing author profiles or samples. This is essential for determining the author of an unknown code sample. The methods of attribution can be classified based on the approach taken for this comparison.

Similarity-based methods measure the pairwise similarity between the unidentified code and all training samples or author profiles in the training set. The final decision is based on identifying the author whose work shares the greatest similarity with the unknown sample. These methods are applicable in both Profile-Based and Instance-Based models.

Vector Space Methods in code authorship attribution involve representing code as vectors in a multidimensional space. Each dimension of the vector can represent different features of the code, such as syntax, structure, formatting, variable naming conventions, and comments. This transformation creates a numerical representation of the source code, which can be analyzed using various mathematical and machine learning techniques. Machine learning algorithms, such as Support Vector Machines (SVM), neural networks, or clustering algorithms, are then applied to these vectors for authorship attribution. Instance-based attribution models employ these vector space methods.

Probabilistic methods in authorship attribution calculate the likelihood that a given sample of code was written by a specific author from a predefined set of authors. The author who has the highest probability is considered the most likely creator of the code. These methods assess the probability P(C|A), where C is the code and A is a potential author. While commonly used in profile-based attribution models due to their probabilistic approach, these techniques can also serve as a similarity measure. They help in determining how closely an unseen code sample aligns with all the samples in the training dataset.

4.5. Machine Learning

Machine learning attempts to determine whether the code was written by a particular author by examining characteristics of the code, such as writing style, word usage, and grammatical structures, using various algorithms and statistical techniques. Nowadays,

with the development of machine learning technologies, more accurate and faster results can be obtained in the field of AA.

4.5.1. Classification Algorithms

In this section, we provide explanations of each machine learning algorithm that is both frequently used in the literature and employed in our research.

- **Random Forest**: Random Forest is a popular and powerful classification and regression method used in machine learning. It is an ensemble learning model consisting of decision trees. In this method, many decision trees are trained on random subsets, and their outputs are combined to make a more accurate and stable prediction. It consists of an ensemble of tree predictors, where each tree is influenced by the values of an independently sampled random vector, which follows the same distribution across all the trees in the forest [77]. Each tree predicts independently, and results are combined by majority vote or average. It avoids overfitting by using random subdatasets and features. The impact of each feature on the model's prediction performance can be measured, which is useful for feature selection.

- **K-Nearest Neighbors**: K-Nearest Neighbors (KNN) is a non-parametric algorithm in machine learning, applicable to both classification and regression tasks. It operates by considering the k closest training examples in the dataset, where "k" is a small, positive integer. In the k-NN classification, it assigns a class membership to an object based on the most frequent class among its k nearest neighbors. Specifically, if k equals 1, the object is assigned to the class of its single nearest neighbor.

- **Support Vector Machine - SVM**: Support Vector Machine (SVM) is a powerful and flexible supervised learning model used in machine learning. It can be used effectively in both classification and regression tasks. The main goal

of SVM is to find an optimal separation hyperplane (or a line in hyperspace) to separate data points. In a binary classification task that involves two distinct classes, the primary goal of an SVM is to establish the most effective classification function that can accurately differentiate between the members of these two classes as observed in the training dataset. The criterion for what constitutes the "best" classification function in this context is often interpreted geometrically [78, 79].

Currently, two main approaches are used for multi-class SVM. The first involves creating and combining several binary classifiers, and the second directly incorporates all data into a single optimization framework [80].

1. One-against-one method involves constructing $k(k-1)/2$ classifiers, each trained on data from two different classes.
2. One-against-all method builds k SVM models, corresponding to the number of classes, where the mth SVM is trained using all examples of the mth class with positive labels and all examples from other classes with negative labels.

- **Gaussian Naive Bayes**: Naive Bayes methods comprise a collection of supervised learning algorithms that apply Bayes' theorem under the simplifying assumption that each pair of features is conditionally independent, given the class variable's value [81]. In Gaussian NB, it is presumed that the features follow a Gaussian distribution in terms of their likelihood.

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{1}$$

In Eq. 1, $P(x_i|y)$ represents the probability of feature x_i given the class y, under the assumption that the values of x_i are distributed according to a Gaussian (normal) distribution.

- **LightGBM**: Gradient Boosting Decision Trees (GBDT), a prominent machine learning technique, is effectively implemented in various forms including

LightGBM, XGBoost and pGBRT. LightGBM, a highly efficient version of GBDT, is distinguished for its rapid training capability and high accuracy. Introduced by Ke et al. in 2017 [82], LightGBM incorporates unique features such as Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS optimizes the training set creation for the ensemble's base trees, while EFB consolidates sparse features into a single one, acting as a preprocessing step [83].

4.5.2. Cross-Validation

Cross-Validation is used to evaluate the performance of a predictive model. It involves splitting the dataset into subsets, fitting the model into some of these subsets, and validating it on the rest. This process is repeated to ensure robustness in performance estimation. The following introduces different cross-validation techniques:

- **K-Fold Cross-Validation**: The dataset is divided into "k" equally sized folds. The model undergoes training and evaluation k times, with each iteration utilizing a distinct fold for validation. The performance metrics obtained from each fold are then averaged, providing an estimate of the model's overall generalization capability.
- **Stratified K-Fold Cross-Validation**: Similar to K-Fold but ensures that instead of the splits being entirely random, the proportion of the target classes in each fold is maintained to be the same as that in the entire dataset.
- **Leave-One-Out Cross-Validation (LOOCV)**: Each observation is used as a validation set while the rest (N-1) observations are used for training.
- **Hold-out Cross-Validation:** The data set is splitted into training set and test set by some percentage.

Cross-Validation is a vital step in ensuring the reliability and generalizability of the model's performance. The choice of CV strategy should be tailored to the dataset's characteristics, and the model evaluation metrics should align with the specific goals of the analysis.

4.5.3. Grid Search CV

Grid Search perform exhaustive search over specified parameter values for an estimator [84]. Grid Search Cross-Validation involves a systematic approach to hyperparameter tuning for machine learning models. Initially, a hyperparameter grid is defined, consisting of a dictionary where keys represent the hyperparameters and values are the ranges or specific settings to be tested. The next step is choosing the appropriate machine learning model or estimator, alongside the evaluation metric to be used by GridSearchCV for assessing model performance across different hyperparameter sets. The process then involves GridSearchCV conducting cross-validation, where the model is trained on various data subsets with each unique hyperparameter combination. Finally, GridSearchCV identifies and retains the best parameters that yielded the highest score according to the chosen metric. Upon completion of the search, it presents the optimal hyperparameters that enhance the model's performance [85].

5. PROPOSED METHOD - ANDROID AUTHORSHIP ATTRIBUTION

The objective of this study is to introduce a comprehensive approach to AA in the context of Android applications. This section details the methodology, encompassing feature extraction, data processing, model architecture, and validation techniques employed to predict authorship within the Android app domain.

The proposed method aims to predict authorship in Android applications by leveraging an extensive array of features extracted from various components within the app's structure. The model integrates advanced data processing and classification techniques along with robust validation methodologies to enhance the accuracy and reliability of authorship predictions.

There are only a few studies [14–17] on AA for Android. AppAuth [16] extracts distinguishing features for AA from the *.dex* file, manifest file, and resource files of applications. In [14], they used all the string information placed in the string XML files and *.dex* files of applications. Unlike these studies, in this study, source code-based features extracted from smali files are used for the first time. In addition to the source code-based features, the effects of other proposed features (usage of permissions, TPL, and metadata-based features) is also analyzed. All of these features are given as SPL, which stands for Source code-based features, Permissions, and TPL.

5.1. Model

Figure 5.1 shows the schematic representation of the architecture of the proposed model. The decompiled *APK* files serve as the entry point for feature extraction, resulting in the creation of fixed-size feature vectors specifically designed for classification tasks.

The *APK* files were first decompiled using apktool [29], and the smali files were obtained. Subsequently, the String, AppAuth, and SPL features of the applications

are collected from these files and the *APK* files, and a fixed-size feature vector per application is given as input to the classifiers. SimpleImputer [30] was used in the scikit-learn library to fill in the missing values in the feature vectors. After this imputation process, these features were standardized by removing the mean and scaling to the unit variance using StandardScaler [31] in the scikit-learn library.

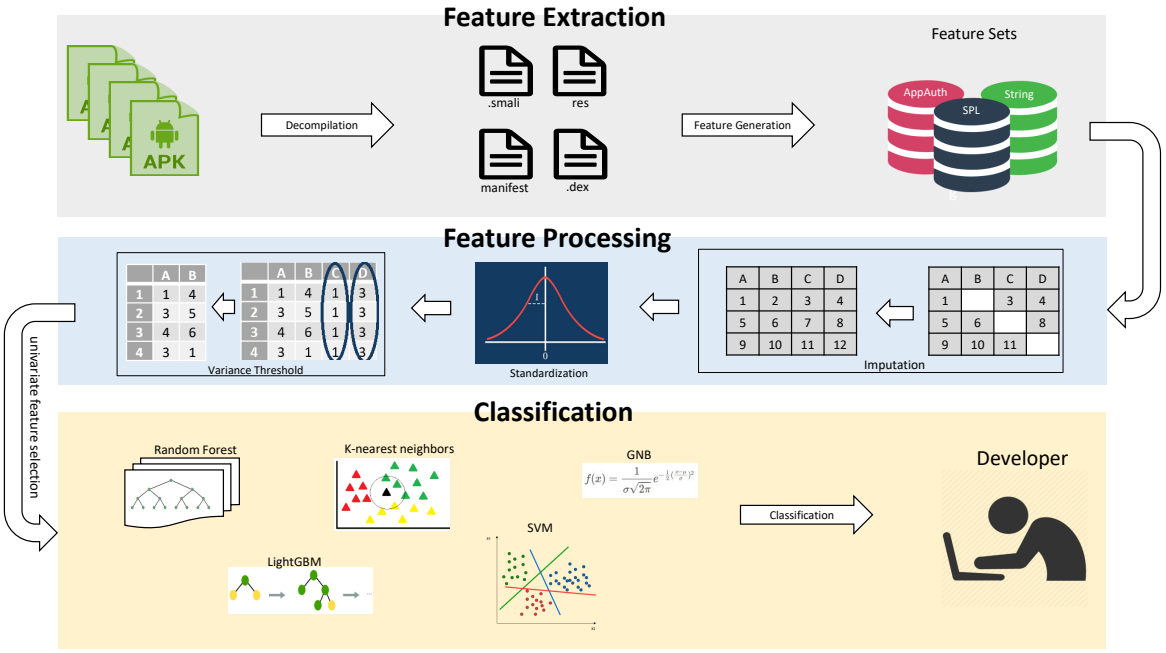


Figure 5.1 Overview of Model

Two-phase dimensional reduction is used before the classification algorithms are executed. First, the features with zero variance were eliminated from the dataset. Then, univariate feature selection was applied, which selected the best features based on the univariate statistical tests. Five different classification algorithms are applied using the scikit-learn library [32]. Then, stratified 10-fold cross-validation is employed. Because stratified 10-fold cross-validation ensures that the proportion of positive to negative examples from the original distribution is preserved in all folds, it is especially useful when the dataset is unbalanced [33]. Therefore, it was used five times, and the average of these five runs is presented in the results.

5.2. Dataset

In this study, two different datasets were mainly constructed: market and malware datasets, consisting of benign and malicious applications. To construct the market dataset, a Web scraper was implemented using the Scrapy framework [34] to collect benign applications from various alternative markets. In addition, applications from other studies [14, 15, 35, 36] were included in this dataset.

The benign dataset contains binaries collected from different alternative Android markets, namely Apkpure [37], Apkmirror [21], Onemobile [38] and Aptoide [20] between January 2020 and August 2020. However, it also includes applications written much earlier than 2020 because of the availability of early versions of contemporary applications. The distribution of the benign datasets over the years is presented in Table 5.1. Initially, 200,000 applications were downloaded. Then, developers with fewer than ten applications are eliminated because earlier studies use developers with at least ten applications [14, 16]. Moreover, we must have sufficient applications for each developer to generate a good model that can differentiate between developers.

Table 5.1 Year distribution of dataset

Year	**Percentage (%)**
2014	4.9
2015	5.4
2016	8.6
2017	39.7
2018	41.4

The malware dataset contains malicious binaries belonging to malware families, namely Ransomware, Adware, and SMS [39], and some binaries from datasets introduced and used in security-related studies, namely Rmvdroid [40], Drebin [35], Genome [15] and Koodous [14].

The number of authors and applications collected from different repositories in the dataset is shown in Fig. 5.2. Due to the elimination of authors with fewer than

ten applications, the dataset consists mainly applications that are mostly written by multiple developers or companies. Fig. 5.3 displays a histogram representing the distribution of *APK* sizes, measured in megabytes (MB), for the market and malware datasets.

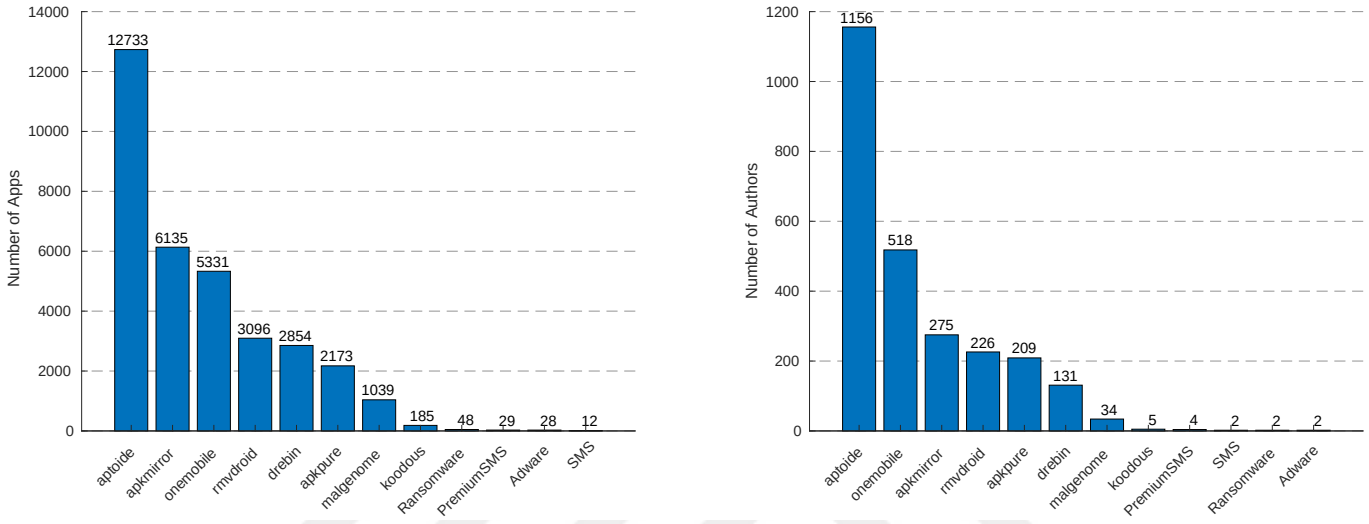


Figure 5.2 Summary of dataset

Authorship attribution is primarily based on the hypothesis that matching serial numbers in certificates are indicative of the same author, excluding cases involving public or leaked certificates. Each application must be signed with a developer certificate installed on an Android device. Therefore, applications that share the same signature are assigned to the same developer, and each application is grouped according to its related signature, as in [14]. Signatures were extracted from the *APK* files using the *print-apk-signature* tool [41].

Some authors were eliminated in the feature extraction step because of errors. If an application exists in more than one dataset, only one of them is randomly chosen and kept in the dataset. If an application has several versions, they are all kept in the dataset. As a result, the dataset given in Table 5.2 was used in all experiments other than version analysis, in which the effect of versions on AA was analyzed, and metadata analysis, in which features extracted from application metadata were analyzed on AA.

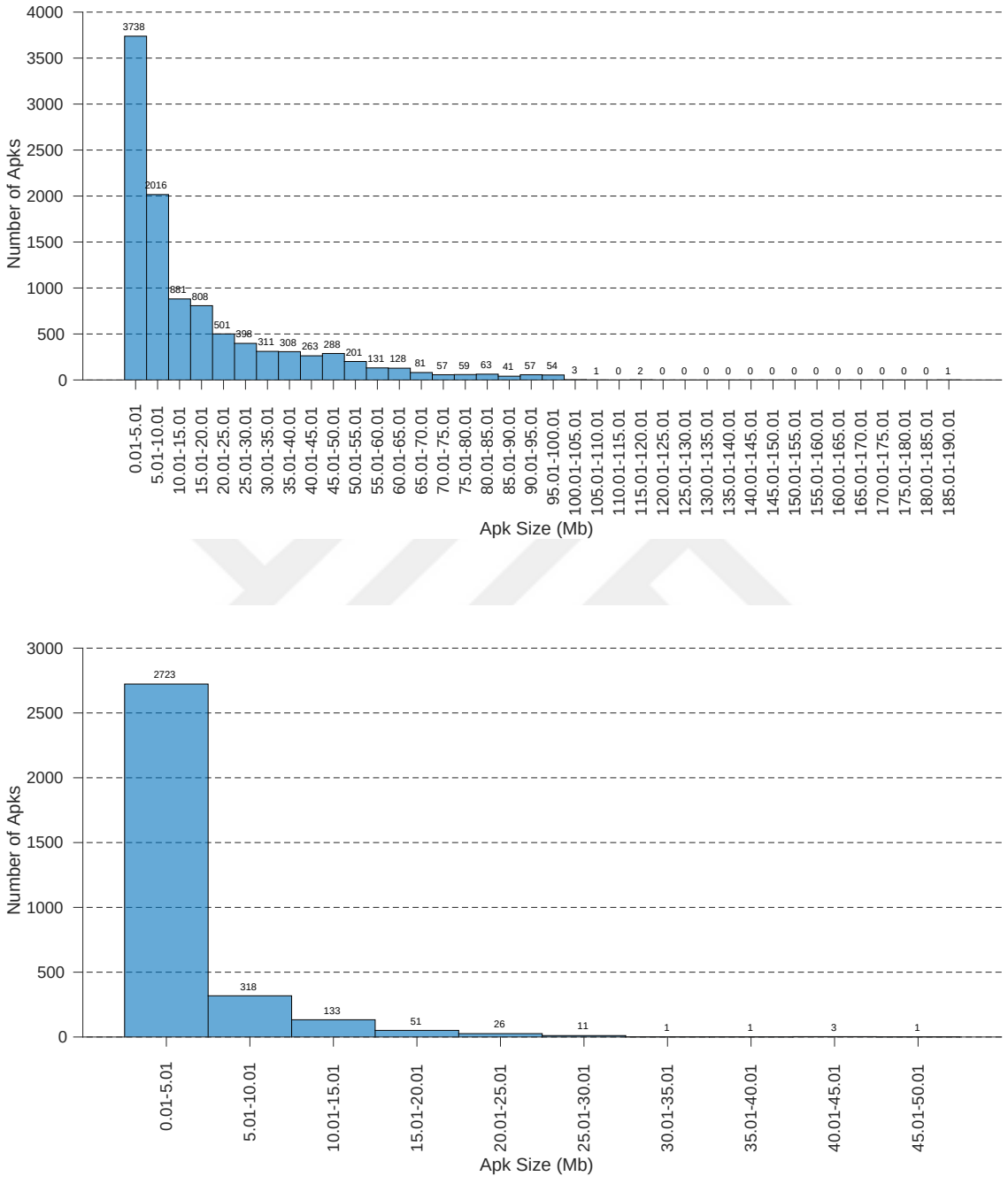


Figure 5.3 APK size distribution (Market at the top, Malware at the bottom)

Since not all applications have metadata information and different versions, a subset of the dataset is used for version and metadata analysis.

Table 5.2 Dataset information

Dataset	# of app	# of author
Market	10,385	488
Malware	3,268	153
Genome	1,530	39

We employed an alternative dataset to show the effect of metadata-based features due to challenges in acquiring applications' descriptions. The dataset size of descriptions can be found in Table 5.3.

Table 5.3 Metadata analysis dataset info

Dataset	# of app	# of author
Rmvdroid	1,216	67
Market	2,359	133

5.3. Feature Extraction

The proposed model employs feature extraction processes, starting with the decompilation of *APK* files using "apktool". This step enables the extraction of smali files, *.dex* files, resource files, manifest files, XML files etc. Three primary feature categories are extracted using all these files:

- **AppAuth Features**: Discriminating features for AA from the *.dex* file, manifest file, and resource files of applications. AppAuth uses the configuration features collected from the manifest file, as listed in Table 5.4. It also uses features related to resource files, such as the number of files/folders in the *res* directory. The use of the dex features is on analyzing the usage of methods, classes, fields, and data structures in *.dex* files, which also encompass annotations, interfaces, and debug information. To address the variability in data structure features caused by developers releasing apps of different sizes, ratio values are used instead of absolute numerical values to avoid size-related biases.

Table 5.4 Feature set descriptions

Category (# of features)	Description
Source Code-Based (18)	# of average character per line
	# of average character per local variable
	# of average character per global variable
	# of average character per function name
	# of average character per function parameter name
	Ratio of global variables to lines of code
	Ratio of local variables to lines of code
	# of average lines of code per function
	Ratio of variables to lines of code
	Ratio of if to all codes
	# of average lines of code per class or interface
	# of average number of functions per class or interface
	Ratio of invoke to all codes
	Ratio of move to all codes
	Percentage of function names starting with an uppercase letter
	Percentage of int function definitions to all
	Percentage of void function definitions to all
	Percentage of identifiers beginning with an uppercase character
DEX Code (14)	Ratio of abstract classes
	Ratio of classes containing annotations
	Ratio of direct methods
	Ratio of virtual methods
	Ratio of methods containing try and catch
	Ratio of methods containing debug information
	Ratio of static fields
	Ratio of classes containing interfaces
	Ratio of mathematical instructions to functional instructions
	Ratio of **aget** instructions to **aput** instructions
	Average of the length of all the arrays
	Median of the length of all the arrays
	Standard deviation of the length of all the arrays
	Ratio of arrays with constant length
Configuration (9)	Number of activity
	Number of service
	Number of receiver
	Number of provider
	Number of intent-filters
	Number of meta-data
	Number of uses-permission
	Number of sensitive uses-permission
	Number of uses-feature
Resource (11)	Number of directories in the **res** directory
	Number of directories whose name containing **drawable** in the **res** directory
	Number of files in the directories whose name containing **drawable**
	Number of directories whose name containing **layout** in the **res** directory
	Number of files in the directories whose name containing **layout**
	Number of directories whose name containing **values** in the **res** directory
	Number of files in the directories whose name containing **values**
	Number of files in the **assets** directory
	Number of files in the **lib** directory
	Number of .so files in the **assets/lib** directory
	Number of **XML** files in the **res** directory
Permission (158)	Android permissions
Library (500)	A whitelist of TPL
Metadata (10,000)	Application descriptions placed in Android markets
String/n-gram (10,000)	DEX - Strings present in the *.dex* file
	Application - Strings extracted from the strings.xml file
	All strings (DEX + Application)

- **StringAA Features**: Derived from the string XML and *.dex* files of the applications. In [14], preliminary experiments were first conducted to determine the optimal n value for n-gram analysis, which was selected to be 3. Therefore, in this study, 3-grams are also used for string-based features. Following the approach in [14], to prevent the omission of lines containing one or two words, we prepend and append a tag to each string. This ensures that only lines without any content are removed. As a result, line-bounded 3-grams were obtained for each string type. Table 5.5 shows the 3-gram features for the two different types of strings. Due to the extensive volume of 3-grams extracted from our dataset, the HashingVectorizer technique was employed to reduce 3-gram features. Consequently, this process allowed for the selection of the most effective 3-grams.

Table 5.5 Sample 3-gram features

Type	**String**	**3-gram**
app	Enable Google Play services	<LB>Enable Google Enable Google Play Google Play services Play services <LB>
DEX	mContainer=	<LB>mContainer= <LB>

- **SPL Features**: Four feature groups were introduced, such as source code-based, permissions, TPL, and metadata-based features. With the exception of those related to TPL, these features are exclusive to the Android platform. The extraction of source code-level features from smali files represents a novel approach within the realm of Android AA. Permissions are pulled from the manifest file, while TPL features are sourced from directories within the decompiled *APK* file. Furthermore, metadata features are collected from the application's marketplace page.

  In this study, smali files, which closely resemble the original source code, were utilized to extract features based on the source code. The source code of software

can offer valuable attributes for determining the authorship of unidentified software. One such attribute is the coding style of the authors. Although some salient features, such as the usage of comments, loops, and braces in the source code, are lost during the compilation process, as shown in [66], the coding style still remains in the binary code. Based on this assumption, we inherit some source code-based features listed in [63] because of the nature of the Dalvik bytecode, which preserves some stylistic features of AA. All 18 source code-based features used in this study are listed in Table 5.4. The features listed in [63] are generally proposed for languages such as Java and C/C++; therefore, features compatible with smali codes from that list are selected. In the future, more source code-based features, such as comments, annotations, bracket positions, indentation styles, etc., could be included using the original source code of applications.

The decompiled Java code can also be extracted from the *APK* files. To obtain Java files from *APK* files, *.dex* files must first be generated from *APK* files, and Java files must then be extracted from these *.dex* files using decompiler tools such as JADX [86]. The same source code-based features extracted from the smali files can be extracted from decompiled Java files. However, because the Java language does not contain some smali-specific instructions such as move and invoke, two features related to these instructions are not considered in Java codes. Note that the transformation process from *APK* files to Java files is prone to errors, such as the inability to convert some classes, methods, or variables correctly. Owing to such errors encountered in the conversion, we were able to extract features from Java codes from 80% of the samples in the market dataset. Note that only the source code-based features extracted from the decompiled Java files are used to show and compare the effect of the source code-based features in the decompiled Java files and the smali files.

While AppAuth [16] used the number of permissions in the manifest file as a single feature, in this study, the existence of each feature is considered a separate

feature. Therefore, 158 binary features corresponding to the 158 permissions are added to the feature set.

TPL are also included in the feature dataset. To extract library features, a whitelist of the library was built. First, we downloaded all library names from the mvnrepository website [87]. Then, we try to determine which common libraries are used in both malware and benign datasets using the downloaded library name list. When an *APK* file is decompiled using apktool, all smali codes of the libraries used in the application are placed in the src directory according to its package name. For example, if the jsoup library is used, all smali codes related to jsoup are placed in the src/org/jsoup directory because the jsoup package name is "org.jsoup". Hence, we can find the directories of libraries used by their package names and exclude them. This enables us to only obtain custom smali codes written by the author of the application. Some libraries can be obfuscated; therefore, their names consist of arbitrary character sequences rather than meaningful words. We also eliminated these libraries. Libraries used in very few applications were also not considered in this study. For each application, the existence of each library in the list was used as a feature. In a recently proposed study [17], TPL were used for Android AA. However, unlike our approach, the names of TPL were used as string features.

The application descriptions generated by the developers were analyzed for the problem. Because it is textual information, it might help solve AA. The same method from [14] was employed here to extract n-grams. Unlike the textual data used for AA in the literature, the application descriptions are short texts. For example, application descriptions are limited to 4,000 characters on Google Play.

In conclusion, the main objective of this study is to find effective features to solve the AA problem on Android. Therefore, new features based on source code-based AA extracted from smali files are presented. Furthermore, new features unique to the Android system have been analyzed, such as the usage of permissions and

metadata-based features that have been increasingly used in Android security in recent years [88]. It has been shown that many applications use a large number of TPL [89] and on average, 60% of the code belongs to these libraries [25, 26]. As TPL create noise, they are shown to affect repackage and malware detection on Android [90]. This could also affect Android AA because source code-based features were employed in this study. Therefore, here, the effects of extracting source code-based features only from custom code on the Android AA are investigated. Moreover, the effects of TPL as features are explored. The proposed features are compared with those in the literature, which are based on binary AA [14, 16]. We did not include Gonzalez et al.'s [15] work in the comparison because AppAuth [16] also used the same features used in that study [15]. All features considered in this study are listed in Table 5.6, and their explanations are listed in Table 5.4. The features first explored in this study are indicated in red. In the following sections, we first introduce the features previously identified in the literature, followed by a presentation of the features newly proposed in this study.

Table 5.6 Features proposed for Android AA in the literature

Features	**Abbr.**	**Our Study**	**AppAuth[16]**	**StringAA[14]**
Configuration	conf	✓	✓	
DEX Code-Based	dex	✓	✓	
Resource	rsrc	✓	✓	
String-Based	str	✓		✓
Source Code-Based	src	✓		
Permission	perm	✓		
Third-Party Library	lib	✓		
Metadata	meta	✓		

5.3.1. Feature Set Descriptions

The features covered in this study are categorized to offer a comprehensive insight into various aspects of app development, structure, and behavior. Each feature has been designed to capture specific elements of the code of Android applications. This categorization not only aids in a holistic understanding of application architecture but also plays a crucial role in tasks such as AA, security analysis, and functionality

assessment. The features are grouped into distinct categories, each representing a different dimension of the application. Within each category, individual features are explained to offer clarity on their significance and the type of information they provide.

1. **Source Code-Based (18 features)** Features based on source code, extracted from decompiled Smali files. Many of these features are applicable in other languages, such as Java, but some are specific to a particular language.

    - Number of average characters per line: This measures the average number of characters in each line of code, which can indicate coding style and complexity.
    - Number of average characters per local variable: It calculates the average character count for local variable names, offering insights into naming conventions.
    - Number of average characters per global variable: Similar to local variables, this assesses the average length of global variable names.
    - Number of average characters per function name: This reflects the average length of function names, which can vary significantly among different programmers.
    - Number of average characters per function parameter name: It measures the average length of function parameter names, contributing to understanding the coder's naming style.
    - Ratio of global variables to lines of code: This ratio provides an insight into how frequently global variables are used relative to the overall code length.
    - Ratio of local variables to lines of code: It indicates the density of local variable usage in the code, which can vary based on coding practices.
    - Number of average lines of code per function: This shows the average size of functions.

- Ratio of variables to lines of code: This overall ratio gives an idea about variable usage in relation to the total code length.
- Ratio of "if" to all codes: It measures how frequently conditional statements are used, which can vary based on programming style and application logic.
- Number of average lines of code per class or interface: This indicates the average size of classes or interfaces, reflecting on the complexity and structure of the code.
- Number of average number of functions per class or interface: It provides insights into the class/interface complexity by showing how many functions they typically contain.
- Ratio of "invoke" to all codes: This ratio indicates the frequency of function or method calls within the code.
- Ratio of "move" to all codes: It measures the proportion of data movement-related operations in the code, like variable assignments and data manipulation.
- Percentage of function names starting with an uppercase letter: This statistic reflects on naming conventions, particularly in function naming.
- Percentage of "int" function definitions to all: It shows how often functions return an integer value, which can indicate the nature of operations performed.
- Percentage of "void" function definitions to all: This measures the frequency of functions that don't return a value, often used for their side effects or operations.
- Percentage of identifiers beginning with an uppercase character: This is another metric reflecting naming conventions, particularly in how identifiers are capitalized.

2. **DEX Code (14 features)** These features are extracted from the *.dex* file itself.

- Ratio of abstract classes: This measures how often abstract classes are used, indicating a certain level of code abstraction and design patterns.
- Ratio of classes containing annotations: It indicates the use of annotations in classes, which can be significant in modern Java programming.
- Ratio of direct methods: This reflects the use of direct methods in the code, which are methods directly invoked by the caller.
- Ratio of virtual methods: It measures the use of virtual methods, indicating object-oriented programming practices.
- Ratio of methods containing try and catch: This shows how often exception handling is used, which is essential for robust and error-free code.
- Ratio of methods containing debug information: It indicates the prevalence of debugging information in methods useful for code maintenance and troubleshooting.
- Ratio of static fields: Measures the use of static fields, which can indicate a certain style of programming or application design.
- Ratio of classes containing interfaces: Shows how often interfaces are used, indicative of design principles like abstraction.
- Ratio of mathematical instructions to functional instructions: This compares the use of mathematical operations to other functional instructions, giving insights into the nature of the application.
- Ratio of “aget” instructions to “aput” instructions: Measures the balance between array get and array put operations, reflecting on data manipulation patterns.
- Average length of all the arrays: Indicates the typical size of arrays used, which can relate to the nature of data handling in the application.
- Median length of all the arrays: Provides a central tendency measure of array sizes, complementing the average length information.

- Standard deviation of the length of all the arrays: Shows the variability in array sizes, which can indicate the diversity of data structures.
- Ratio of arrays with constant length: Measures how often arrays of fixed length are used, which can reflect on the nature of data structures and algorithms used.

3. **Configuration (9 features)**Each app includes a manifest file embedded in the *APK* file, providing key information such as package name, app ID, components, requested permissions, and device compatibility. Upon decompiling the app, this encoded manifest can be decoded into an AndroidManifest.xml file, from which configuration features can be extracted.

    - Number of activities: Counts the total activities in the application, indicative of its complexity and user interface components.
    - Number of services: Reflects the number of service components, which are essential for background processing and operations.
    - Number of receivers: Indicates the number of broadcast receivers, showing how the app interacts with system or application events.
    - Number of providers: Counts the data providers, revealing how the app shares or manages data.
    - Number of intent-filters: Shows the number of ways the application can be triggered or communicated with, reflecting its interactivity.
    - Number of meta-data: Counts metadata elements, which can provide additional context or configuration information about the application.
    - Number of uses-permission: Indicates the number of permissions the app requests, reflecting its access requirements.
    - Number of sensitive uses-permission: Specifically counts permissions that access sensitive data or system features, important for security analysis.

- Number of uses-feature: Shows the number of hardware or software features the app declares to use, providing insights into its functional scope.

4. **Resource (11 features)**Upon decompiling apps, three additional directories named 'assets', 'lib', and 'res' are created, containing all essential resource files such as icons, pictures, sounds, XML files, compressed files, native library files, and language files. Resource features are extracted from these files. While developers may change these files for apps with different functionalities, the structure of these resources and the quantity of files can also indicate patterns in the app development process.

    - Number of directories in the "res" directory: Counts the resource directories, giving an idea of the variety of resources used.
    - Number of directories whose name contains "drawable" in the "res" directory: Counts drawable resource directories, important for understanding the visual elements of the app.
    - Number of files in directories with "drawable": Shows the number of drawable files, indicating the extent of visual resources.
    - Number of directories whose name contains "layout" in the "res" directory: Counts layout directories, reflecting on the UI complexity.
    - Number of files in directories with "layout": Indicates the number of layout files, correlating with the UI design complexity.
    - Number of directories whose name contains "values" in the "res" directory: Counts directories with value resources, like strings and dimensions.
    - Number of files in directories with "values": Shows the number of value resource files, which are crucial for app localization and configuration.
    - Number of files in the "assets" directory: Counts the files in the assets directory, which can include various types of raw application resources.

- Number of files in the “lib” directory: Indicates the number of library files, which can reveal the use of native code or TPL.
- Number of .so files in the 'assets/lib' directory: Specifically, it counts the shared object files, often used in apps with native code components.
- Number of “XML” files in the “res” directory: Counts the XML files in resources, important for configuration and UI design.

5. **Permission (158 features)**

- Permissions in Android apps are indicators of the resources and data the app intends to access. This can range from access to the device's camera and location to contacts and personal information.

  The permissions an app requests can reveal a lot about the app's intended functionality. For example, it is logical and expected for a camera app to request access to the device's camera and storage permissions. However, if the same app requests access to contacts or call logs, it may raise questions about its functionality and the necessity of such permissions [91]. The permission system can also be used by malicious applications by requesting unnecessary permissions or using permissions in unintended ways.

  Dangerous permissions are permissions that provide access to sensitive user data or system features. They are more critical than regular permissions and usually require explicit user consent at runtime, and malicious applications often request this type of permissions.

6. **Library (500 features)**

- A whitelist of TPL: Identifies known TPL used in the app, which can indicate the app's dependencies and potential external integrations.

  Third-party libraries may offer patterns or coding styles that are not originally owned by the primary author, potentially complicating the attribution process. They make it difficult to isolate and analyze the unique

coding style of the person or team responsible for underlying development. Therefore, when attributing authorship, it is important to distinguish between original code written by the author and sections of code coming from these third-party libraries.

7. **Metadata (10,000 n-gram features)**

   - Application descriptions placed in Android markets: Analyzes a vast array of metadata descriptions provided in app stores, which can offer insights into the app's purpose, features, and intended audience. Metadata may also contain security-related information, for example descriptions are expected to contain information about dangerous permissions [92].

8. **String/n-gram (10,000 features)**

   - DEX - Strings present in the *.dex* file: Examines a large set of strings within the *.dex* file, providing clues about the app's functionality and structure.
   - Application - Strings extracted from the strings.xml file: It focuses on strings defined in the application's resource files, which are important for understanding the UI and user-facing features.
   - All strings (DEX + Application): Combines string analysis from both the *.dex* file and application resources, offering a comprehensive view of the app's textual content.

5.4. Feature Processing

Feature Processing phase involves steps aimed at handling the extracted features for optimal utilization in subsequent analysis.

- **Feature Vector Generation**: The combination of various features extracted from different sources results in the creation of feature vectors for each individual

application. These vectors encapsulate a comprehensive set of attributes that contribute to the overall characterization of the application in terms of authorship.

- **Handling Missing Values**: Feature extraction may result in situations where certain features get null or NAN (missing) values. To eliminate these features from feature vectors, the SimpleImputer function from the scikit-learn library is employed. This function replaces missing values across each column by using a descriptive statistic (e.g., mean, median, or most frequent) or by using a constant value.

- **Feature Standardization**: After SimpleImputer[30], the feature vectors undergo standardization for consistency and enhanced model performance. The StandardScaler function from the scikit-learn library is applied to standardize the features. This standardization process involves centering the data by removing the mean and scaling it to unit variance. This standardization step is for ensuring that all features contribute equally to the analysis, preventing any particular feature from dominating due to its scale or magnitude.

- **Dimension Reduction**: In classification tasks, it is essential to reduce the dimensionality of the feature vectors to improve model efficiency and performance. A two-phase dimensional reduction process is applied to the dataset:

  - **Features with Zero Variance Elimination:** Features with zero variance are those that have the same value for all samples in the dataset. These features do not provide any discriminatory information. Hence, they are not informative for the classification task. Removing features with zero variance is a common preprocessing step to simplify the feature set and enhance model performance. The process of eliminating zero-variance features typically involves calculating the variance for each feature, and if

the variance is zero (meaning all values are the same), the feature is removed from the dataset.

– **Univariate Feature Selection based on Statistical Tests:** After removing zero-variance features, the next phase involves selecting the most relevant features using univariate statistical tests. Univariate feature selection helps identify the features that have the most significant impact on the target variable, allowing the model to focus on the most relevant attributes and potentially improving classification accuracy. Common statistical tests used for univariate feature selection include the chi-squared test, ANOVA (Analysis of Variance), mutual information, and more. We employ the ANOVA test in this study. These tests assess the relationship between each feature and the target variable. Depending on the chosen statistical test, a certain number of top-performing features are selected. The number of features to retain is often determined through experimentation or cross-validation. Univariate feature selection helps reduce the dimensionality of the dataset by retaining only the most informative features, which can be particularly beneficial when dealing with high-dimensional data.

By applying these two phases of dimensional reduction, the model aims to streamline the feature set, eliminating irrelevant or redundant attributes while retaining those that are most discriminative for AA. This can lead to improved model performance and reduced computational complexity.

5.5. Machine Learning (ML) Model Development and Optimization

Classification algorithms are applied to predict authorship within the Android application domain. The following classification algorithms are explored in this study:

- Random Forest

- K-Nearest Neighbors
- SVM
- Gaussian Naive Bayes
- LightGBM

To ensure robust model evaluation, stratified 10-fold cross-validation is employed. This method preserves the proportion of positive to negative examples across folds, a crucial strategy for handling unbalanced datasets. The entire process is executed five times, and the results are averaged to provide an evaluation.

To achieve the best possible model performance, we utilize the 'GridSearchCV' technique, which stands for Grid Search Cross-Validation. 'GridSearchCV' is a method provided by the scikit-learn library in Python that allows an exhaustive search over a specified parameter grid to determine the best possible combination of hyperparameters for a given estimator. The initial parameters for each ML algorithm and their values can be seen in Table 5.7

Table 5.7 Hyperparameters of classification algorithms

SVM	{'C': [0.1, 0.25, 0.5, 1, 10, 100, 1000, 10000], 'kernel': ['linear']}
	{'C': [0.1, 0.25, 0.5, 1, 10, 100, 1000, 10000], 'gamma': [10, 1, 0.1, 0.01, 0.001, 0.0001], 'kernel': ['rbf']}
KNN	'n_neighbors': [3, 5, 11, 19],
	'weights': ['uniform', 'distance'],
	'p': [1, 2, 3],
	'metric': ['euclidean', 'manhattan', 'minkowski'],
	'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
	'leaf_size': [10, 20, 30, 40]
RF	'bootstrap': [True, False],
	'max_depth': [10, 30, 50, 70, 90, None],
	'max_features': ['auto', 'sqrt'],
	'min_samples_leaf': [1, 2, 4],
	'min_samples_split': [2, 5, 10],
	'n_estimators': [200, 600, 1000, 1400, 1800]
GaussianNB	'var_smoothing': np.logspace(0, -9, num=100)

By combining 'GridSearchCV' with Stratified K-Fold cross-validation technique, we ensure that the selected hyperparameters are perform well across different subsets of the dataset. This approach improves the generalization of our machine learning models.

Utilizing ‘GridSearchCV‘ in the context of cross-validation is an integral part of our research methodology, as it helps in fine-tuning the models and enhancing their predictive power.

6. EXPERIMENTAL RESULTS

In this study, we conducted a series of experiments to evaluate our framework and analyze the effects of newly introduced features. The performance of the proposed approach is also explored for different types of applications, such as malicious, benign, and obfuscated applications. Various experiments were performed to analyze the performance of the proposed approach. For each experiment, 10-fold cross-validation and five epochs were employed, and an average of 50 results were obtained. All experiments were run on a CentOS 7.7 server with 128 GB RAM and Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz.

We attempt find answers to the following research questions (RQs) on how our model identifies the author of Android applications, that are either benign or malicious:

- **RQ1** - *What is the performance of classification algorithms in solving the AA problem?*
- **RQ2** - *What is the ideal set size for n-gram features?*
- **RQ3** - *Does the use of TPL bring improvements in solving the authorship problem?*
- **RQ4** - *How effective is the proposed approach in identifying the developer of applications?*
- **RQ5** - *Does metadata of applications help to attribute the developer of applications?*
- **RQ6** - *What are the most important features for attributing applications to their developers?*
- **RQ7** - *Can we reduce the number of features without decreasing the performance of the model?*

- **RQ8** - *Does the number of applications per developer affect classification performance?*
- **RQ9** - *Does the proposed model successfully identify different versions of applications developed by the same author?*
- **RQ10** - *What is the effect of obfuscation on Android AA?*
- **RQ11** - *Are there any clone applications in the datasets? How do they affect the performance on Android AA?*

6.1. RQ1 - Performance of classification algorithms on Android AA

Motivation. Machine learning algorithms may exhibit different performances for different problems. With this motivation, we aim to compare the performance of machine learning algorithms on the problem at hand so that the algorithm that shows the best performance can be used in subsequent experiments.

Method. Random Forest, K-Nearest Neighbors, Support Vector Machines, Gaussian Naive Bayes, and LightGBM algorithms are used in the experiment. These algorithms are highlighted in the literature for their effectiveness in code AA tasks [27]. To optimize the performance of these algorithms on both the market and malware datasets, the GridSearchCV function from the scikit-learn library is utilized. GridSearchCV conducts a comprehensive search over specified parameter ranges to find the best settings for the models.

In this experiment, all available feature groups are incorporated, including those used in the AppAuth [16] and String Analysis [14], with the exception of metadata features. The exclusion of metadata features is due to the lack of descriptions for some applications, which leads to imbalance in the dataset.

The algorithms are evaluated on both benign and malware datasets, with accuracy and F1-score chosen as the performance metrics. These metrics are selected because an efficient model should have good precision and high recall. Additionally, the classification time for each algorithm is measured.

Result. The comparative analysis of machine learning algorithms for AA is summarized in Table 6.1. Among the algorithms, Random Forest (RF) emerges as a right choice due to its high accuracy and F1-scores alongside a reasonable classification time. This factors led to the decision to employ the RF algorithm in subsequent experiments.

Table 6.1 Comparison of classification algorithms

	Market			**Malware**		
	acc	**f1**	**time(s)**	**acc**	**f1**	**time(s)**
Random Forest	82.4%	80.3%	124.46	95.4%	94.5%	14.18
KNeighbors	64.8%	62.0%	76.30	82.6%	79.1%	9.40
Support Vector	51.3%	50.5%	549.83	79.7%	76.6%	26.09
Gaussian Naive Bayes	49.5%	47.6%	45.97	60.3%	59.0%	8.23
LightGBM	**86.4**%	**85.1**%	1563.35	**97.5**%	**97.0**%	370.81

In addition, it is observed that the default parameters for the RF algorithm yield performance very close to those achieved with its optimized parameters. This suggests that RF is a robust choice for AA, capable of delivering good performance without extensive parameter tuning.

While LightGBM demonstrates superior accuracy compared to RF, its longer classification times and the high variability of its results across different parameter sets are considered drawbacks. These factors, coupled with RF's overall solid performance, influenced the decision to prioritize RF for further exploration. However, the high accuracy of LightGBM indicates its potential as a valuable algorithm for AA in future investigations.

6.2. RQ2 - The ideal set size for n-gram

Motivation. As the number of applications increases, the number of 3-grams requiring processing increases correspondingly. In the market dataset, this could amount to tens of millions of 3-grams, posing significant computational and storage issues. To mitigate these issues and enhance the model's performance, it is crucial to identify the maximum number of n-grams from which meaningful results can be extracted.

By focusing on the most informative n-grams, it's possible to reduce the feature size without significantly compromising the model's accuracy or its ability to generalize from the data.

Method. As suggested in [14], 3-grams are selected for use in this study based on the finding that system performance improved with an increase in n-gram size from 1 to 3, but deteriorated with further increases. This finding underlines the importance of choosing an optimal n-gram size to capture sufficient contextual information.

To mitigate memory consumption associated with the potentially vast number of n-grams, a *HashingVectorizer* is employed. *HashingVectorizer* converts a collection of text documents into a numerical format, specifically a sparse matrix, that represents the occurrence of tokens within those documents. In our case, the vectorization process is applied using 3-grams as the tokens. This means that instead of considering individual words as tokens, the process considers sequences of three consecutive words as single tokens. This approach can capture more contextual information, potentially leading to more nuanced representations of the text documents in the matrix.

The use of *HashingVectorizer* not only addresses memory constraints but also facilitates efficient computation by reducing the dimensionality of the feature space without significant loss of information. This strategy is particularly valuable in scenarios where the dataset encompasses tens of millions of n-grams, as it enables the processing of large-scale data while keeping resource usage in check.

Result. The relationship between the number of 3-grams utilized in classification and its impact on both classification accuracy and training time is elucidated in Figures 6.1a and 6.1b, respectively. These figures highlight that the training time experiences a notable increase when the classification process incorporates more than 10,000 3-grams. However, this augmentation in the number of 3-grams does not correspondingly enhance classification accuracy to a significant extent.

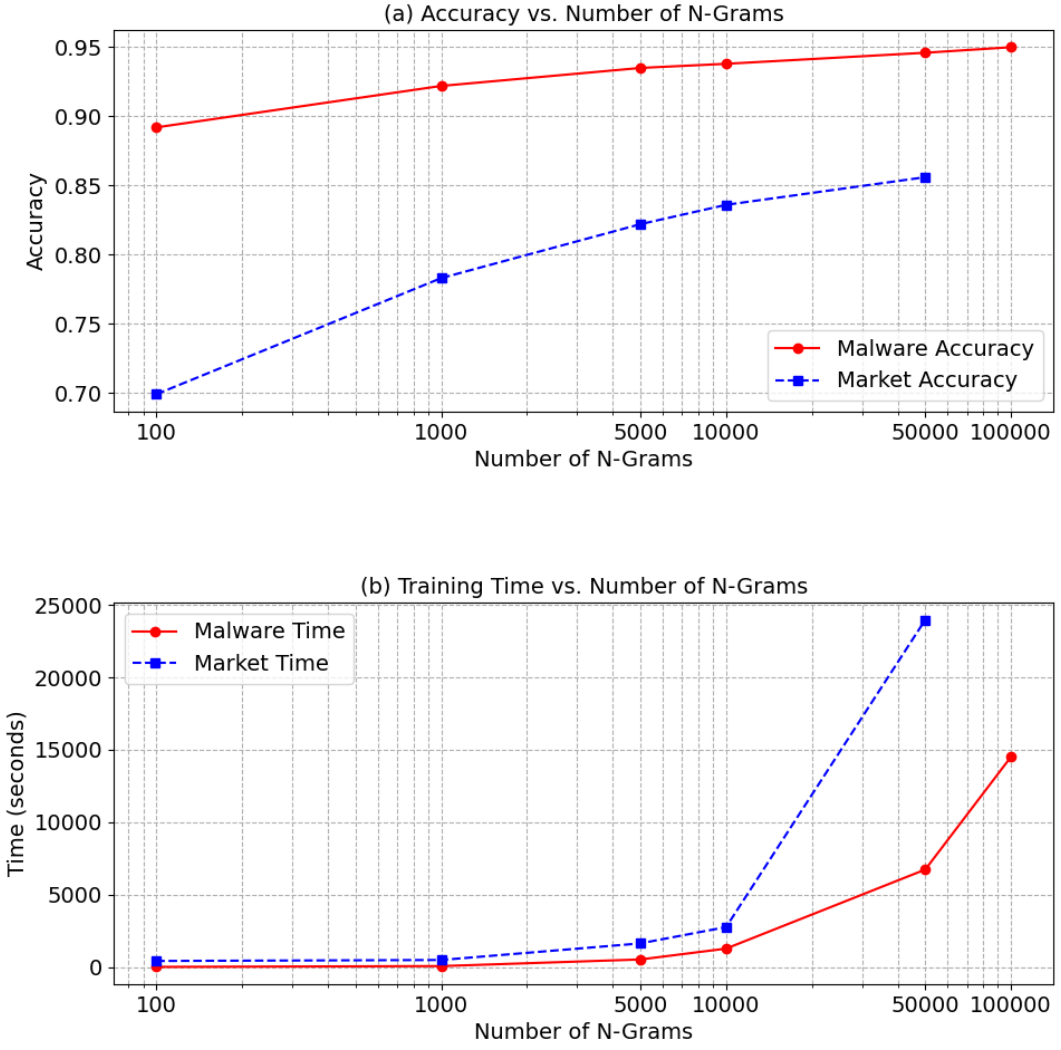


Figure 6.1 Effect of the number of 3-grams

The constraints imposed by the market dataset, notably its extensive size and the consequent generation of a massive amount of 3-grams, limit the use of more than 50,000 3-grams in the classification process. This limitation underscores the importance

of optimizing the selection of 3-grams to ensure that the computational resources are directed towards processing the most impactful features, thereby achieving an efficient and effective classification without unnecessarily extending the training time.

6.3. RQ3 - Custom code vs. all code including TPL

Motivation. Developers often exhibit specific coding habits and preferences, such as opting for a *while* loop over a *for* loop or favoring object-oriented modularization over procedural programming [93]. These habits and stylistic choices are typically preserved in the custom code that developers write, serving as distinguishing features that can aid in AA.

The use of Third-Party Libraries (TPLs) is a common practice in application development. While custom code reflects the developer's unique characteristics, TPL code can introduce noise into the AA process, potentially obscuring the fingerprint that identifies an author's style. Despite this, the consistent use of specific TPLs by developers across their applications could contribute positively to AA. Developers often exhibit loyalty to certain libraries, frequently reusing them without updating to newer versions, even when the older versions contain known security vulnerabilities [94].

This tendency not to update TPLs can inadvertently serve as a developer's fingerprint, thereby assisting in AA. It's essential to discern how much the presence of TPL code can help or hinder the identification of the true author of a piece of software. This involves examining whether the benefits of using TPLs as features of developer habits outweigh the potential noise they introduce into the attribution process.

Method. To elucidate the impact of Third-Party Libraries (TPL) on Authorship Attribution (AA), the study employs two distinct settings:

- Custom Code Only: In this setting, source code-based features are extracted exclusively from the custom code written by the developers. This approach

focuses on identifying the unique coding habits and styles intrinsic to the developer, excluding any influence from external libraries.

- Including TPL (All Codes): Contrarily, in the second setting, features are extracted from the entire application codebase, incorporating both the custom code and the TPL. This comprehensive approach is intended to evaluate how the inclusion of TPL affects the AA process.

Result #1. As shown in Table 6.2, including TPL when extracting source code-based features yields much better performance (custom src vs. all src). Then, the custom code is enriched with other features presented in Table 5.6 and compared with all code in Table 6.2. The results clearly show that, rather than extracting source code-based features from TPL, the existence of TPL is sufficient for AA and produces much better results than including TPL's code. Therefore, in the subsequent experiments, the source code-based features are only extracted from the custom code and are used with the library features (custom src+lib).

Table 6.2 Accuracy results of custom source code enriched with other feature groups

Dataset	Custom Src	Custom Src+Lib	All Src	Custom SPL	All Src +Perm	Custom SPL +AppAuth	All Src+Perm +AppAuth
Market	42.3%	70.9%	66.8%	75.1%	71.8%	**78.7%**	76.1%
Malware	80.5%	87.1%	87.5%	91.5%	91.1%	**93.4%**	93.3%
Genome	80.7%	97.1%	95.0%	97.9%	96.4%	**98.0%**	97.4%

Result #2. The confusion matrix is presented in Fig. 6.2, where authors with at least 40 applications in the market dataset are included to fit the matrix. The figure illustrates that certain applications from developer auth9 are incorrectly matched with those of developer auth11. Further analysis of these authors is conducted using SimiDroid [43], which reveals the similarity between mismatched applications of developer auth9 and all applications of developer auth11. A high similarity between such applications is shown in the results, owing to the SmaliHook library, which is not eliminated because it does not appear on our TPL list. Analysis of the feature vectors for these applications through cosine similarity reveals extremely high similarities

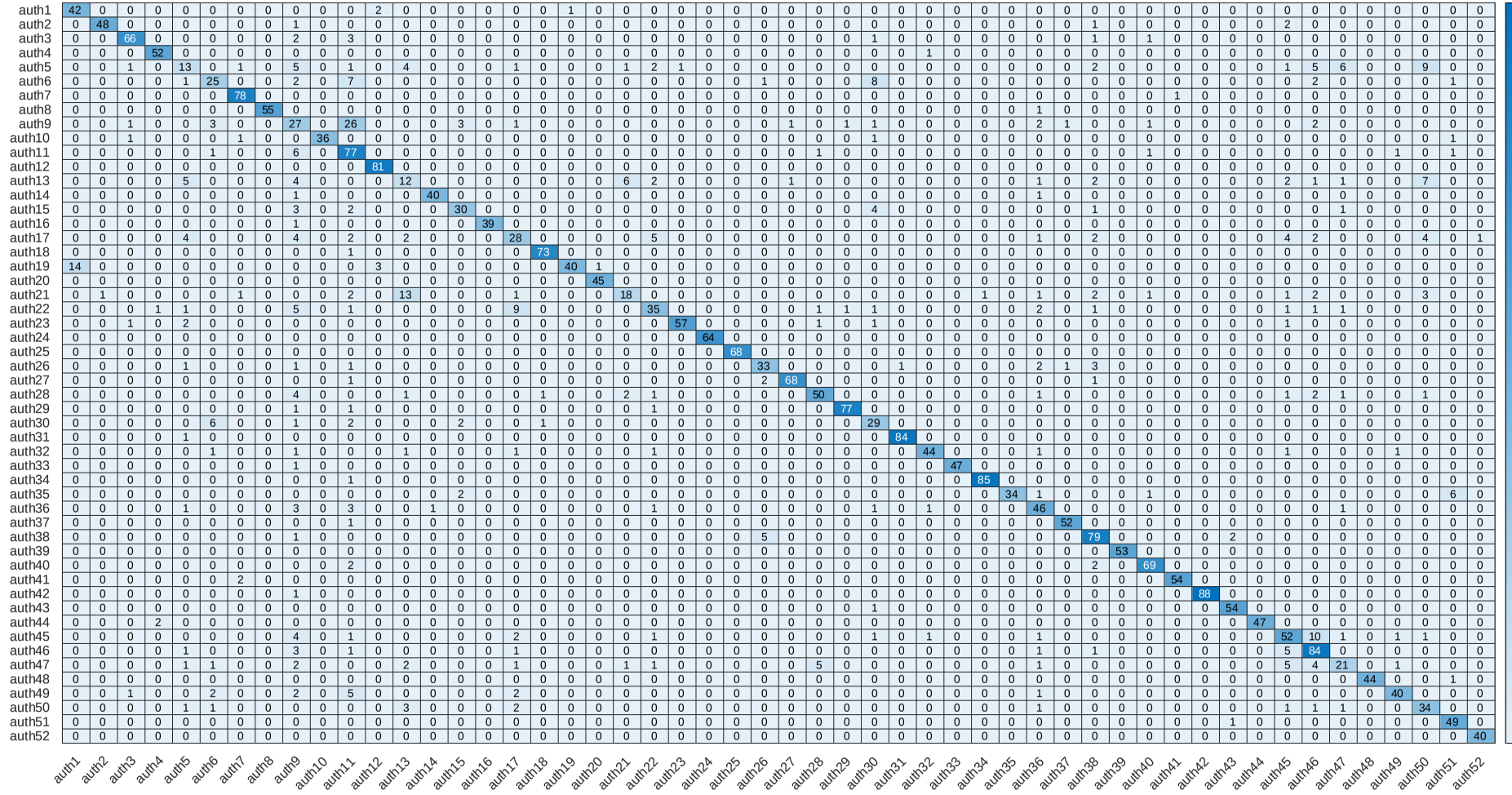


Figure 6.2 Confusion Matrix for the 52 developers who have at least 40 applications

(98%). The results are severely affected due to the impact of the overlooked TPLs on the source code-based features. This finding indicates the necessity of broadening the TPL scope within the library feature set to enhance our model's accuracy. Beyond the current whitelist method, future enhancements could incorporate techniques [24, 94, 95] in the literature to identify additional libraries, including those that are obfuscated.

6.4. RQ4 - Effectiveness of the proposed approach

Motivation. The primary goal of this study is to pinpoint effective features for Android Authorship Attribution (AA), thereby enhancing the ability to accurately identify the authors of applications. To achieve this, the features proposed in this study are benchmarked against other feature sets previously suggested in the literature, specifically those detailed in [16] and [14]. This comparative analysis is designed to assess the effectiveness of different feature sets in AA.

The effectiveness of the proposed approach in identifying authors is presented in a comparative manner, taking into account performance metrics such as accuracy and F1 score. This allows for a nuanced understanding of how well each feature set can distinguish between different authors.

Method. To facilitate a fair and accurate comparison, the study involves a re-implementation of the code necessary to extract the string-related features as described in [14].

Additionally, the authors of AppAuth demonstrated the research community's cooperative nature by making their source code accessible for others to extract the features they had identified. This generosity allows for the direct application of their code in this study, ensuring that the AppAuth features are extracted in a manner fully consistent with their original implementation. Such an approach not only underscores the importance of reproducibility and transparency in research but also significantly enhances the reliability of the comparative analysis conducted.

Result #1. As shown in Table 6.3, the newly proposed features perform better than AppAuth, whereas the proposed features produce less accuracy than string analysis [14]. However, as stated above, extracting all 3-gram features may not be applicable from a classification time point of view. Therefore, only 10,000 3-grams are employed here. As a relatively small dataset, all 3-grams are extracted from the Genome dataset, which is the dataset used for evaluation in [14]. In the Genome dataset, the proposed features show slightly better performance than string-related features. When all the features are included, the best performance is obtained in the market and malware datasets.

Table 6.3 Comparison with AppAuth and StringAA - Random Forest

Dataset	SPL+AppAuth+ StringAA		SPL+AppAuth		SPL		AppAuth		StringAA	
	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1
Market	**82.5**%	**80.4**%	79.0%	76.6%	75.0%	72.4%	73.9%	71.2%	81.7%	79.9%
Malware	**95.6**%	**94.5**%	93.9%	92.5%	92.1%	90.7%	90.6%	89.4%	95.1%	94.2%
Genome	97.0%	96.7%	**98.2**%	**98.0**%	98.1%	97.9%	97.0%	96.9%	96.9%	96.7%

We also calculate the time required to extract each set of features. Table 6.4 shows that n-gram features require at least one and a half times more time than other feature sets. The extraction times of SPL and AppAuth features are relatively close. Although a feature extractor should be performed only once, the feature extraction process, when generating a considerable volume of n-grams, can be notably memory-intensive. This can result in memory-related challenges, particularly for systems with limited RAM availability. In [42], it is highlighted that a primary limitation of the n-gram approach is its tendency for exponential n-gram growth as the text size increases. This complexity often renders the method unviable for systems with limited computational capabilities.

Table 6.4 Feature extraction time in minutes

	SPL	AppAuth	StringAA (10,000)
Market	125.02	95.93	219.87
Malware	11.02	11.61	13.45
Genome	30.04	27.30	40.47

Result #2. The effect of each feature group on Android AA is also explored. The results are presented in Table 6.5. In addition to the n-grams in the code, the source code-based and resource-based features lead to the highest accuracy. Because n-grams are extracted from strings, such as the names of variables or methods in the application code, they can include unique identifiers that distinguish developers. However, as shown in Table 6.4, the time required to extract and process such features is considerably high. Moreover, such features might not be resilient to obfuscation techniques such as renaming and encryption.

Table 6.5 Effect of feature sets on Android AA

Dataset	SPL						AppAuth						StringAA	
	src		perm		lib		conf		dex		rsrc		str	
	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1
Market	**66.8**%	**63.5**%	47.6%	42.4%	60.5%	57.6%	59.6%	55.8%	63.6%	60.0%	66.7%	63.4%	**81.7**%	**79.7**%
Malware	**87.3**%	**85.9**%	80.6%	76.5%	51.2%	46.2%	82.3%	79.3%	85.1%	82.3%	83.7%	80.7%	**95.1**%	**94.2**%
Genome	95.0%	94.7%	83.0%	80.6%	88.1%	86.8%	95.0%	94.6%	88.2%	87.7%	**96.2**%	**95.8**%	**96.8**%	**96.4**%

In general, all feature sets other than library features produce higher accuracy on the malware dataset, which is a smaller dataset. The average number of libraries per application varies significantly between the two datasets, as shown in Table 6.6. While applications collected from the market include 9.37 TPL on average, this number is only 2.72 for malware applications. This may be the result of the use of obfuscation in malicious applications. In this study, a whitelist approach is used to extract the TPL used in the *APK* package. Common libraries used in the market and malware datasets are selected and used as features. However, because of obfuscation, the names of some libraries are simply random words, and such libraries are eliminated in the whitelist approach. In the future, recent studies such as LibID [95] and LibRadar [24] can be used to find obfuscated libraries in the code. Then, the effect of the library features can be re-evaluated on the malware dataset.

The effects of permissions on the two datasets are also significantly different. When the average number of all 158 permissions is analyzed in both datasets, it is found that the use of different permissions is higher in the malware dataset (all permissions:

Table 6.6 Avg # of Library and Permission per application

	Market	Malware
Library	9.37	2.72
Permissions (158)	8.05	9.99
Dangerous Permissions (26)	3	4.5

9.99, dangerous permissions: 4.5) than in the market dataset (all permissions: 8.05, dangerous permissions: 3).

Result #3. To show the statistical significance of the improvements, each approach (namely SPL+AppAuth+StringAA and StringAA) is run five times using 10-fold cross-validation. Thus, 50 results are obtained for each dataset. A *t-test*, with an alpha value of 0.05, is then applied, and the results are shown in Table 6.7. α value represents the significance level, which is the probability of rejecting the null hypothesis when it is true. Table 6.7 shows that the p-values for the market and malware datasets are lower than α value of 0.05, which implies that the difference is statistically significant. Figure 6.3 and 6.4 below show the box plot of the accuracy results of 50 runs represented in Table 6.7.

Table 6.7 Results of t-test

	p Value
Market	3.54622E-14
Malware	0.000562331
Genome	0.09986412

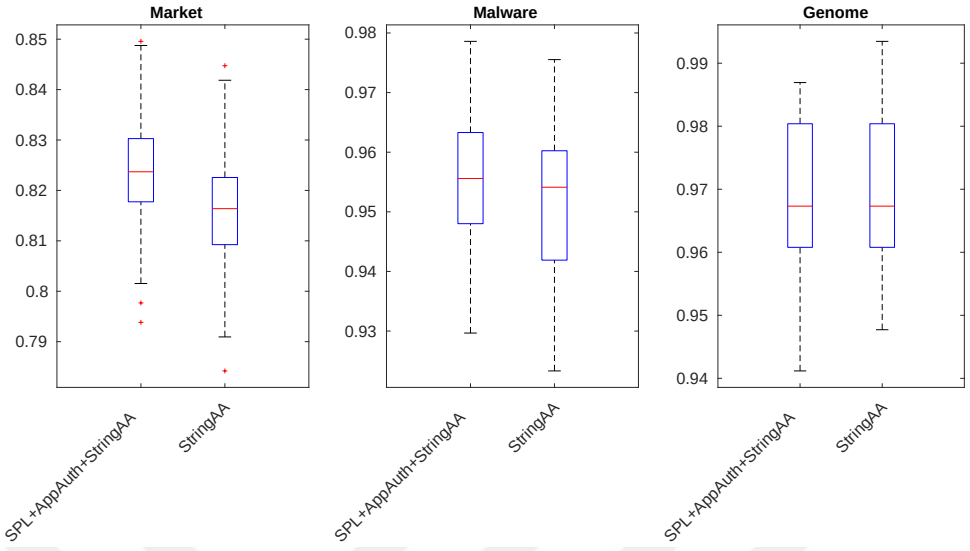


Figure 6.3 Accuracy results used in comparison

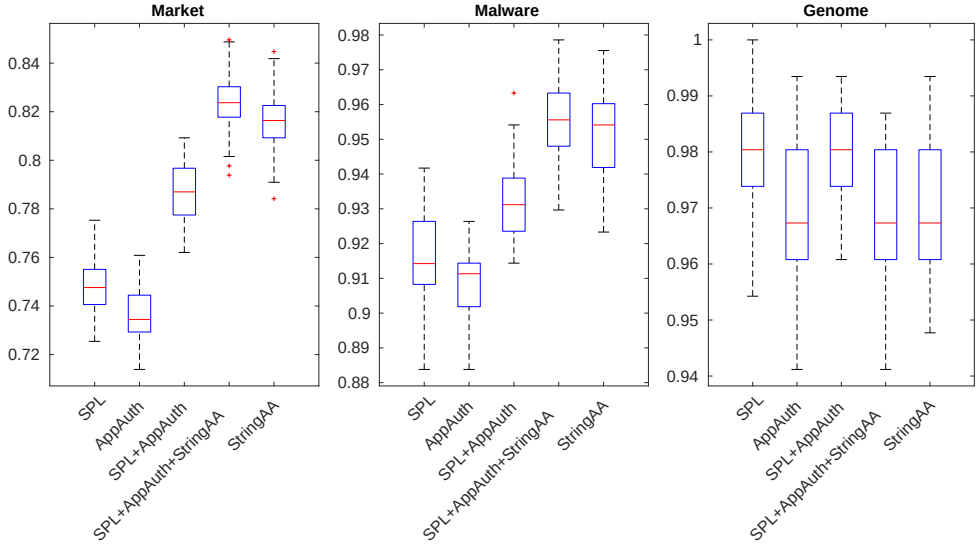


Figure 6.4 Accuracy results of all features used in comparison

Result #4. To our knowledge, this is the first study to explore the use of source code-based features on smali codes for Android AA. As shown in Table 6.5, such features perform much better than the other features proposed in the literature, except for n-grams. Source code-based features can also be obtained from Java files. In the literature, it has been shown that more than 94% of Java classes can be successfully decompiled [96]. Therefore, the same source code-based features listed in Table 5.4

are collected from Java files and are compared in Table 6.8. However, only 80% of the applications are successfully decompiled. Recall that the following features specific to smali code are not available in Java files: the ratio of invoke to all code (the ratio of the number of invoking instructions to the number of all other instructions) and the ratio of move to all code. The comparison results show that the source code-based features collected from the smali files are more effective. However, it is worth mentioning that the decompilation of *APK* files into Java files could be erroneous. Here, the jadx decompiler [86] is used to convert *APK* files to Java files; however, we encounter some unsuccessful decompilations.

Table 6.8 Accuracy results of Smali vs. Java: source code-based features

Dataset	Smali	Java
Market	**66.2**%	52.9%
Malware	**87.0**%	86.8%

6.5. RQ5 - Metadata Features

Motivation. As humans have different writing styles, text data are used largely for AA in different domains, such as social media, e-mail, and literature [97–99]. Android markets provide metadata such as descriptions, version numbers, and download counts. This information can also shed light on new findings. Therefore, the effect of metadata is investigated for the first time for Android AA in this study.

Method. First, we investigate the optimal n value and the number of n-grams. To extract n-grams, each n consecutive word in the descriptions is extracted, each time moving one word to the right. Preliminary experiments are carried out with different n-gram sizes (from 1 to 5) to obtain the optimal value of n. Subsequently, the number of n-grams required to obtain the best results is determined. Second, we combine the metadata features with other feature sets to determine whether the metadata features improve accuracy.

Result #1. The entire market dataset and only Rmvdroid [40] among the malware datasets are used because these datasets contain application descriptions. Fig. 6.5a shows the accuracy results for different n values. We set the number of n-grams to 10,000. It can be observed that 1-grams produce better results than the other n-grams. After determining the value of n, we investigate the optimal number of 1-gram features in further experiments. Fig. 6.5b shows the accuracy results obtained using only metadata features under different numbers of 1-grams. As shown in Fig. 6.5b, the number of 1-grams shows an important effect on the results. Because n-gram analysis requires a significant amount of system resources, we could not obtain results after 100,000 1-grams. If 100,000 1-grams are included, accuracies of approximately 84% and 74% are obtained for the market and malware datasets, respectively. This clearly shows that the application descriptions are beneficial to Android AA. However, it can be seen that the results of 10,000 1-grams are also quite sufficient, and the time required is much less than 100,000 1-grams. Therefore, we chose 10,000 1-grams for upcoming experiments in the metadata analysis.

Result #2. Table 6.9 shows the effects of metadata features in the market and malware datasets, respectively. Because not all applications have descriptions, the dataset used in this experiment is smaller than the original dataset. Application descriptions can be written in any language. Our datasets also include different languages in the application descriptions, such as English, Chinese, and French. It is observed that accuracy increases significantly when we use descriptions in English only. Therefore, descriptions other than English are eliminated from both datasets. In the market dataset, metadata features produce approximately 6% better accuracy when combined with the proposed SPL features, whereas an increase of approximately 3% is observed for other feature sets. In the malware dataset, metadata features show a clear positive effect, with an increase of 3% in the proposed SPL features. In contrast, the increase is not significant for the other feature sets.

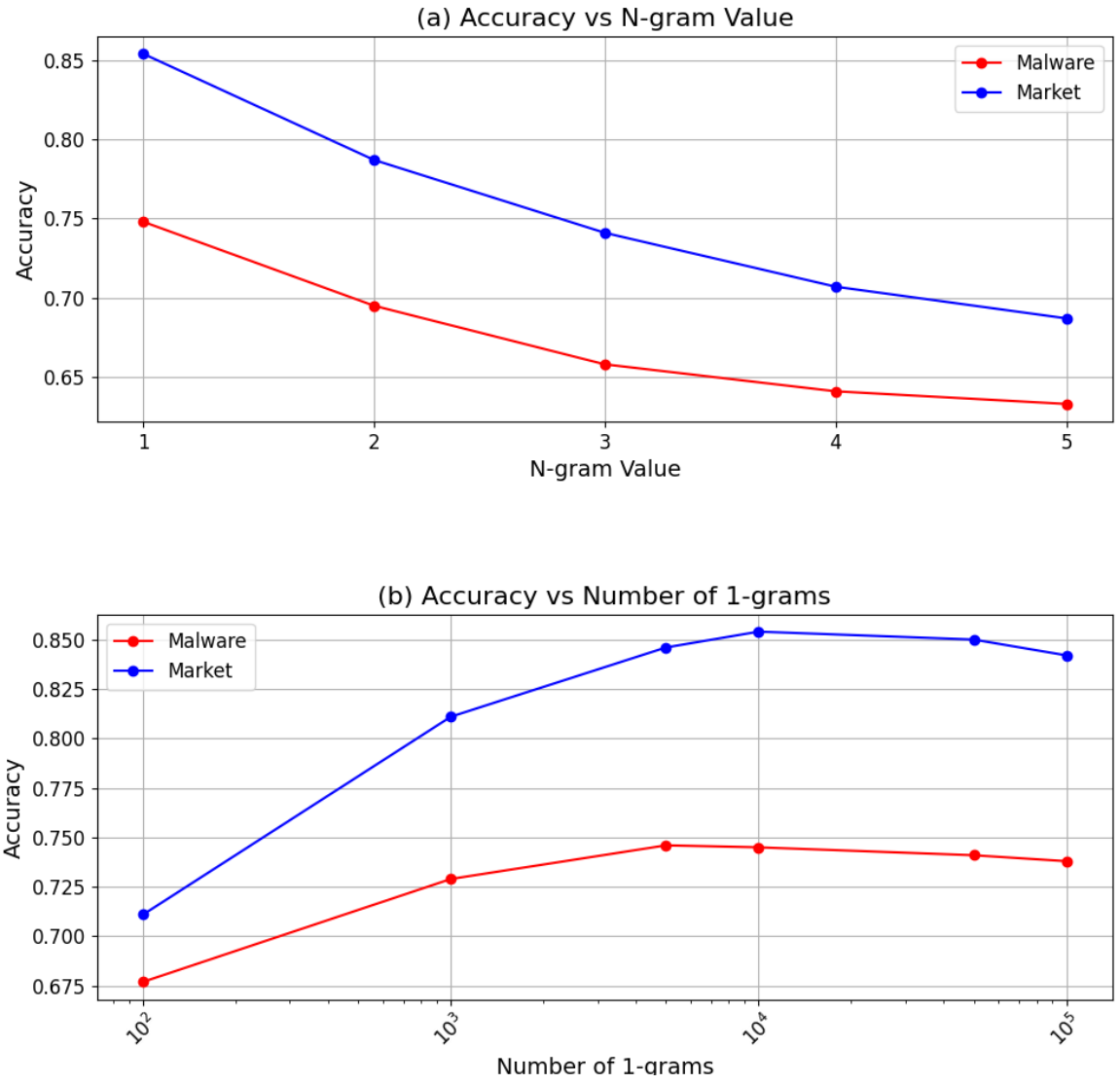


Figure 6.5 Effect of n-grams on metadata features

Table 6.9 Metadata analysis

Dataset	SPL		SPL+Meta		StringAA		StringAA+Meta		AppAuth		AppAuth+Meta	
	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1	acc	f1
Market	81.4%	78.4%	87.0%	85.0%	86.6%	84.5%	89.0%	87.0%	82.6%	80.1%	85.6%	83.3%
Malware	89.1%	87.1%	92.0%	90.3%	94.7%	93.5%	95.0%	93.8%	89.0%	86.7%	89.1%	87.0%

Dataset	SPL+AppAuth +StringAA		SPL+AppAuth+ StringAA+Meta		SPL+AppAuth		SPL+AppAuth +Meta	
	acc	f1	acc	f1	acc	f1	acc	f1
Market	86.8%	84.7%	89.0%	87.1%	84.5%	82.3%	88.4%	86.6%
Malware	94.9%	93.7%	95.4%	94.3%	91.9%	90.5%	93.6%	92.3%

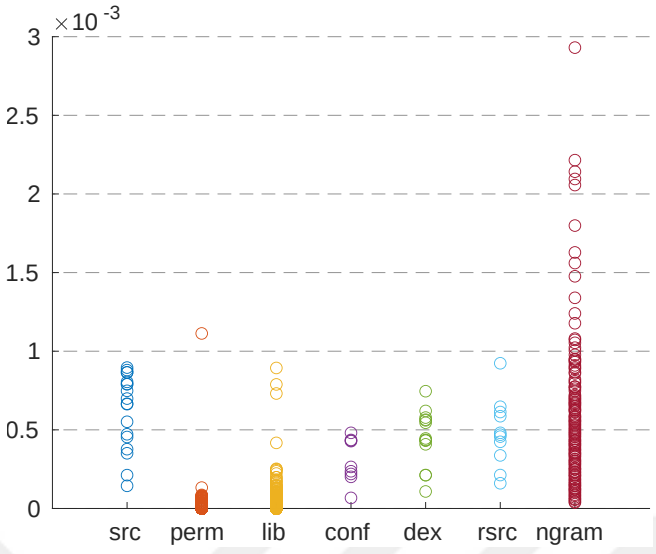
6.6. RQ6 - Most Effective Features on Android AA

Motivation. In addition to knowing why our prediction is high or low, we also want to know which features contribute more and which are irrelevant to improving our prediction. Therefore, in this experiment, we investigate the features that had the greatest influence on the results.

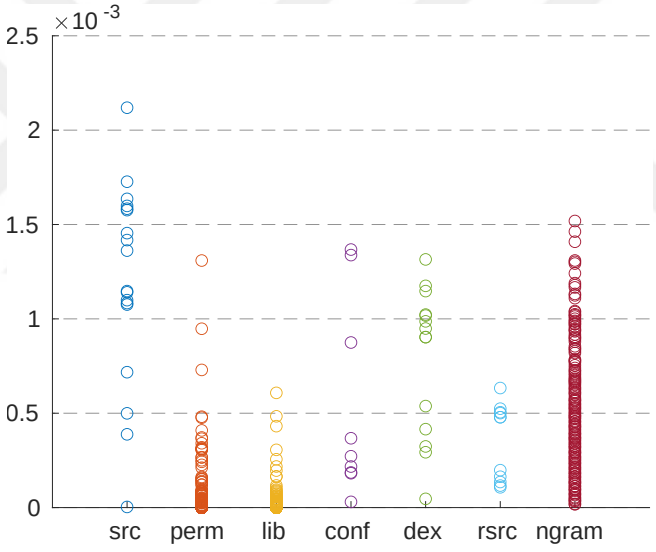
Method. The methodology employed in our experiment involves utilizing all available features, with the exception of metadata, and applying the Random Forest algorithm to achieve our results. Following the application of this algorithm, we leverage the scikit-learn library to extract the importance values of all features. These values are crucial in understanding the contribution of each feature to the predictive model's performance. The extracted feature importance values are then visually represented in two figures: Figure 6.6a illustrates the feature importance for the market dataset, while Figure 6.6b displays the importance for the malware dataset.

Result. The analysis reveals that source code-based features and string/n-gram features have a bigger effect on the results for both the market dataset and the malware dataset than other types of features. Specifically, the features utilized in the AppAuth (conf, dex, rsrc) demonstrate a notably positive effect, particularly within the context of the malware dataset. This distinction underscores the relevance of these features in distinguishing malicious applications from benign ones.

In a detailed breakdown presented in Table 6.10, the distribution of features within the top 50, ranked by importance, is showcased for each dataset. This table effectively quantifies the contribution of different feature sets to the model's prediction, highlighting the prominence of certain feature types over others.



(a) Market



(b) Malware

Figure 6.6 Importance values of each feature

Table 6.10 Distribution of Top 50 features per feature set

Dataset	src	perm	lib	conf	dex	rsrc	str
Market	8	1	2	0	0	1	38
Malware	14	1	0	2	6	0	27

Moreover, the ordering of the 50 most critical features is cataloged in Table 6.11, providing a granular view of which specific features hold the most weight in the

predictive process. The findings from this table further reinforce the significant impact of source code-based features in the malware dataset, indicating their critical role in identifying malicious software. Conversely, the market dataset saw a predominant influence from n-gram features, suggesting their effectiveness in applications more typical of general market trends.

Table 6.11 The most important 50 features per dataset

Market		**Malware**	
Name	**Type**	**Name**	**Type**
interstitial is already	ngram	avgCharPerGlobalVar	src
1s kjrer ikk	ngram	avgLinePerClass	src
lb itt lb	ngram	avgFuncPerClass	src
saved state of	ngram	ratioInvokeToAllCodes	src
elle sera bient	ngram	ratioVoidToAllFunc	src
metadata tag in	ngram	ratioVarToAllCodes	src
lb samplerate lb	ngram	lb getruntime lb	ngram
lb cannot find	ngram	or zero length	ngram
invalid ad size	ngram	lb replace lb	ngram
lb rgb lb	ngram	ratioIfToAllCodes	src
lb isinterface lb	ngram	avgLinePerFunc	src
freeing fragment index	ngram	dapatkan perkhidmatan google	ngram
change.component.enabled.state	perm	susesPermissionNum	conf
container view with	ngram	ratioGlobalVarToAllCodes	src
lb krko lb	ngram	usesPermissionNum	conf
lb score lb	ngram	vtlMethodRatio	dex
lb readbyte lb	ngram	lb plusclient must	ngram
lb pair lb	ngram	receive.mms	perm
lb aan lb	ngram	lb marketsearchqpnamecomgoogle lb	ngram
be null instead	ngram	this message lb	ngram
lb zzafm lb	ngram	lb initializing adview	ngram
lb onanimationend lb	ngram	permissions are not	ngram
lb zc lb	ngram	lb settextalign lb	ngram
lb landroidviewsurface lb	ngram	antClassRatio	dex
lb zzti lb	ngram	lb ljavalangcharsequence lb	ngram
cannot call this	ngram	lb lapp non	ngram
resDrawableFileNum	rsrc	statFieldRatio	dex
como 1s lb	ngram	ratioMoveToAllCodes	src
share via lb	ngram	ratioLocalVarToAllCodes	src
avgCharPerLocalVar	src	lb writedouble lb	ngram
com/appyet	lib	lb iil lb	ngram
lb failure lb	ngram	lb row lb	ngram
ratioInvokeToAllCodes	src	avgCharPerLocalVar	src
vi cc dch	ngram	avgCharPerFuncName	src
avgCharPerFuncName	src	ratioIntToAllFunc	src
avgLinePerClass	src	lb getconfig lb	ngram
key cannot be	ngram	lb landroidgraphicsbitmapconfig lb	ngram
lb multiply lb	ngram	lb stopplayback lb	ngram
not forward oncreate	ngram	dbiMethodRatio	dex
ratioVarToAllCodes	src	lb zllil lb	ngram
giving up on	ngram	agetRatio	dex
ratioGlobalVarToAllCodes	src	the main ui	ngram
avgCharPerGlobalVar	src	lb iso88591 lb	ngram
com/doapps	lib	lb packagename lb	ngram
or out of	ngram	lb getsnippet lb	ngram
lb case_insensitive_order lb	ngram	lb module without	ngram
at least one	ngram	lb zu lb	ngram
lb ce lb	ngram	drtMethodRatio	dex
lb landroidappalarmmanager lb	ngram	lb getlastpathsegment lb	ngram
ratioMoveToAllCodes	src	lipsesc de pe	ngram

By analyzing these importance values, we could make informed decisions about feature selection, focusing on those that offer the most value to our predictive accuracy and potentially discarding or de-emphasizing those with minimal impact. This analysis is essential for refining the model, enhancing its efficiency, and ensuring its robustness in accurately identifying malware and its origins.

6.7. RQ7 - The Effect of the Number of Features

Motivation. When dealing with a large dataset comprising over 10,000 different attributes, as in the case of this study, it is not uncommon to encounter features that may not contribute positively to the performance of a classifier. Some features might be redundant, contain NaN (Not a Number) values, or exhibit zero variance, meaning that they do not vary across different instances and hence provide no useful information for classification.

The presence of such features can negatively impact the efficiency of machine learning models. Redundant features can increase the computational complexity of model training and prediction, while features with NaN values or zero variance can dilute the model's ability to learn meaningful patterns from the data.

The effect of the number of features on the classifier's performance is explored to understand how different feature subsets impact the model's accuracy. This involves experimenting with feature selection technique to identify the most informative attributes and assess their contribution to the classification task.

Method. The selection of the best subsets of features from the entire feature set is conducted through the computation of ANOVA F-values using the *f_classif()* function in scikit-learn, a process aimed at enhancing the model's performance by focusing on the most informative features. ANOVA (Analysis of Variance) is a widely used parametric statistical hypothesis test in feature selection to determine whether there are any statistically significant differences between the means of three or more independent (unrelated) groups.

To explore the impact of reducing the number of features on the performance of our classifier, we progressively decrease the set of features under consideration. Starting with the 100% of features based on their F-values, we gradually include less—80%, 60%, 40%, and eventually 20% of the features. This stepwise approach allows us to closely monitor how simplifying our model by reducing its complexity affects its accuracy. By comparing the model's performance across these different feature subsets, it is possible to identify the optimal balance between the number of features and the model's predictive capabilities.

Result. The performance of each feature group is shown in Fig. 6.7. Please note that zero-impact features are also present. Because the malware dataset has more zero-impact features (435) than the market dataset (174), there was not much improvement in the last percentile. This figure also shows that if time and resources matter, fewer features (for example, 60%) can be used as replacements for all features.

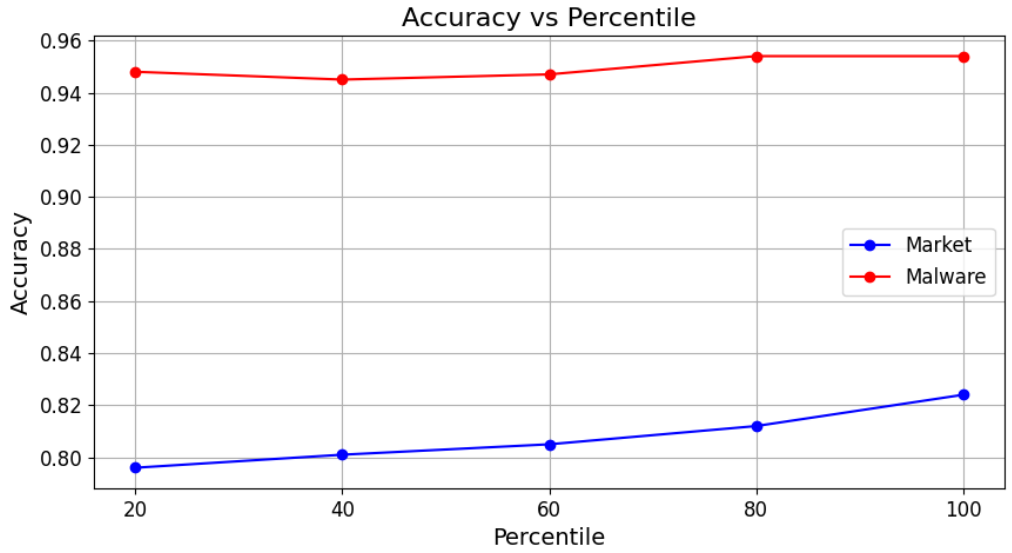


Figure 6.7 Impact of different percentiles of features

6.8. RQ8 - The Effect of the Number of Applications per Developer

Motivation. Supervised machine learning requires prior information on developers to yield accurate results. Therefore, the quantity of applications each developer has contributed to can significantly influence our prediction model's performance. To illustrate this impact, we conducted an experiment demonstrating the relationship between the number of applications per developer and the efficacy of our prediction model.

Method. Our dataset predominantly comprises applications developed by multiple developers, as individual developers with more than ten applications on the market are rare. This leads to an imbalance in our dataset, with a variance in the number of applications contributed by different developers. To mitigate this imbalance and standardize our analysis, we initially identified developers who have contributed to at least 40 applications across both market and malware datasets, resulting in 37 and 19 developers, respectively. For each developer, we then randomly selected subsets of 10, 20, 30, and 40 applications, repeating this process ten times. This approach ensures that each iteration involves different combinations of applications, allowing for a comprehensive assessment of how the number of applications per developer influences our analysis.

Result. Fig. 6.8 shows that when the number of applications per developer is increased, the accuracy of Android AA is also increased. Due to being a smaller dataset with fewer authors, the results on the malware dataset are very high, even when trained with 20 applications per developer. Therefore, the differences among the models trained using 20, 30, and 40 applications per developer in the malware dataset are similar.

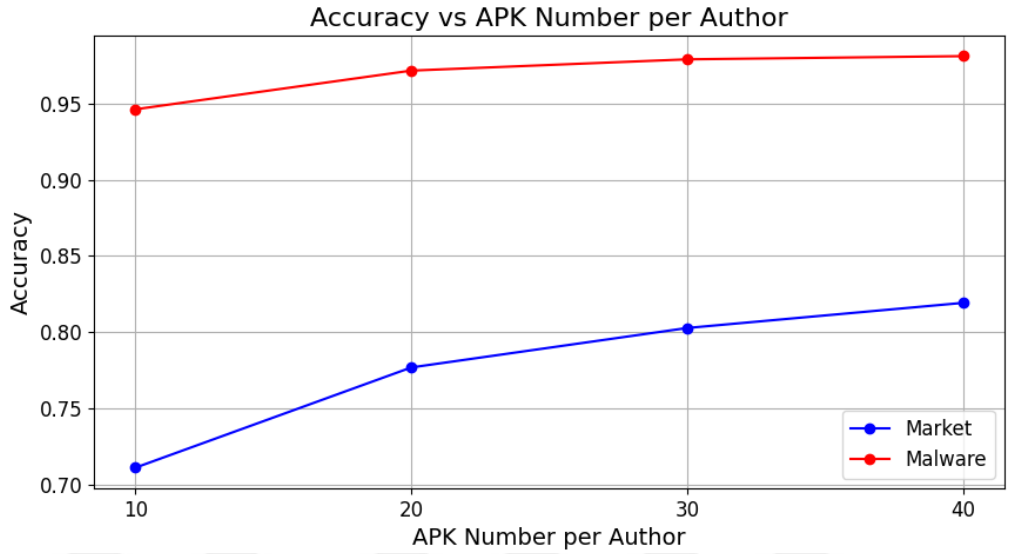


Figure 6.8 Impact of different # of applications per developer

6.9. RQ9 - Effect of Application Versions

Motivation. Developers often update their applications due to bug fixes, security patches, adding new functionalities, and the like. Therefore, there can be multiple versions of an application on the developer's market page. As our market dataset also consists of different versions of some applications, it is worth investigating whether the versions of the applications developed by the same author can be identified by the proposed method.

Method. Therefore, two datasets are constructed in this study. First, if applications have more than one version in the market dataset, duplicate applications are eliminated, and only the first available version of the application is left in the market dataset. This training dataset is called the no-version dataset; their versions are put into another dataset called the testing set. Therefore, 1,193 training and 1,183 testing applications implemented by 42 developers are used. All developers in the no-version dataset have at least ten applications, as in the previous experiments.

Result. We first obtain the result by applying 10-fold cross-validation to the no-version set and achieved 80.7% accuracy. As shown in Table 6.3, if all versions

are included, the accuracy is slightly higher (82.6%). Then, to answer the question "if a version of an application is included in the training, could new versions of this application be detected with the proposed approach?", the model is trained using a no-version dataset and evaluated on the testing dataset. Here, high accuracy (91.7%) is obtained. Similarities between the different versions of the applications are obtained using SimiDroid [43]. However, there is no significant correlation between the unidentified versions of applications and their similarities to the first available versions in training. Although similarities to the first available version generally decrease proportionally to the version numbers, there are exceptional cases in the testing set. Although the similarity between a version of the application and the first available version in the training set is low, the proposed approach can successfully identify the authors of such versions. These results show that the developer's signature is preserved, even if the similarity between different versions of the same application is low.

6.10. RQ10 - Effect of Obfuscation

Motivation. Because both benign and malicious applications apply obfuscation techniques, Android AA has been evaluated in obfuscated applications in the literature [14, 17]. In [17], obfuscated applications were obtained using ProGuard [100], which provides simple obfuscation techniques such as method, class, and identifier renaming, along with code shrinking and code optimization. It is shown that the authors of some applications (7%) could not be identified when they were obfuscated. AppAuth [16] also claims that its features are not robust against encryption and shell package obfuscation. Kalgutkar et al. [14] analyzed obfuscated applications using three different obfuscation tools: ProGuard, Allatori [101], and DashO [102]. They employed different types of obfuscation techniques, such as string obfuscation, string encryption, and control flow obfuscation to the source code of applications. However, they worked on a very small dataset with 96 applications from nine different authors. Their results showed that the

accuracy results of AA in applications obfuscated by ProGuard are unexpectedly 6% better than those of the original applications. String features are expected to be less robust against string obfuscation and encryption techniques; however, it is shown that there is no significant change in the results when applications are obfuscated using the other two obfuscation tools.

Method. In this study, to better understand the performance of Android AA in obfuscated applications, the Obfuscapk [44] tool is used in applications (smali files) in the market and Genome datasets. Obfuscapk [44] works in a black-box fashion, supports advanced obfuscation features, and has a modular architecture that is easily extensible with new techniques. Note that because of some errors encountered during the application of obfuscation techniques, the number of applications used in this experiment is much lower than that of the original dataset (≈40%). However, it is a much larger dataset (6055 applications from 320 authors) than [14]. The six different obfuscation techniques listed in Table 6.12 are used. These obfuscation techniques can be grouped into two categories: encryption and renaming.

Table 6.12 Obfuscation abbreviations

Techniques	**Abbrv.**	**Description**
ConstStringEncryption	CSE	Encrypt constant strings in code
LibEncryption	LE	Encrypt native libs
ResStringEncryption	RSE	Encrypt strings in resources (only those called inside code)
ClassRename	CR	Change the package name and rename classes (even in the manifest file)
MethodRename	MR	Rename methods
FieldRename	FR	Rename fields

Result. The effects of different obfuscation techniques are shown in Table 6.13. Although encryption obfuscation techniques do not affect the results, the newly proposed source code-based feature sets are susceptible to renaming obfuscation techniques, as shown in Table 6.13, because they are extracted from the variable, method, and class names of applications. String features [14] are also affected by renaming techniques, but less than the newly proposed feature sets because string features use all strings placed in the *APK* files, not only strings extracted from the

source codes.

Table 6.13 Obfuscation results

Dataset	Technique	SPL+AppAuth+StringAA	SPL+AppAuth	SPL	AppAuth	StringAA
Genome (1332 applications)	**Original**	97.0%	**97.9**%	97.7%	97.2%	96.8%
	CR+MR+FR	95.9%	**97.6**%	96.5%	97.0%	96.0%
	CSE+LE+RSE	96.9%	**98.1**%	97.8%	97.3%	96.7%
	CSE+LE+RSE+CR+MR+FR	96.3%	**97.7**%	96.5%	97.4%	95.9%
Market (6055 applications)	**Original**	**82.8**%	78.4%	75.3%	73.7%	82.3%
	CR+MR+FR	**81.4**%	75.3%	66.2%	73.5%	80.9%
	CSE+LE+RSE	**82.3**%	78.5%	75.2%	73.9%	81.6%
	CSE+LE+RSE+CR+MR+FR	**81.5**%	75.4%	66.4%	73.6%	80.8%

6.11. RQ11 - Analysis of Clone Applications

Motivation. To ensure data quality, we checked whether our dataset contained application clones and identical descriptions.

Method. The Romadroid tool [45] is used to detect application clones in the dataset. Romadroid creates a string from each manifest file of two applications to be compared and measures the similarity between the two strings using the LCS algorithm. The authors of Romadroid compared their tool with SimiDroid [43] and claimed that Romadroid performed better than SimiDroid over a 60% threshold. Their results showed that Simidroid produced a much lower recall value at their best threshold values than did Romadroid (60.64% vs. 98.38%) in the same dataset.

We use the 70% and 90% threshold values in our experiments, so we calculate the similarity scores of applications of 488, 153, and 39 developers in the market, malware, and Genome datasets using Romadroid. As a result, $n * (n-1)/2$ similarity results are obtained by pairwise comparison of applications. Here, n denotes the total number of applications. Because we encounter errors in some applications, we could not obtain similarity scores for each pair. Many studies typically consider an application pair as an application clone if their similarity scores exceed either %70 or %90 [43, 45, 46]. In our approach, we exclude all applications that exhibited a similarity score above %70 with another.

Result. As shown in Table 6.14, 20.6% of the market and 22.3% of the malware datasets have similar applications above the 70% similarity rate. The effects of clone apps on the results are given in Tables 6.15 and 6.16 for the market and malware datasets, respectively. The results indicate that removing similar applications improves the accuracy and F1 scores, especially for the market dataset. As similar applications from different developers can mislead the model during training, removing app clones has a positive effect on the results.

Table 6.14 Ratio of clone applications in the datasets

similarity threshold	**>%70**	**>%80**	**>%90**
Market	20.6%	14.0%	8.5%
Malware	22.3%	13.0%	9.0%
Genome	10.6%	11.0%	0.3%

Table 6.15 Differences on accuracy and f1 score when clone apps are removed from the market dataset

	customsmali	**perm**	**lib**	**allsmali**	**conf**	**dex**	**res**	**ngram**
acc	-0.572	6.966	5.647	1.033	2.807	1.012	1.233	2.729
f1	-0.527	7.803	6.346	1.116	3.086	1.152	1.516	3.161

	SPL + AppAuth + StringAA	**SPL + AppAuth**	**SPL**	**AppAuth**	**StringAA**
acc	2.626	2.443	3.211	1.608	2.729
f1	3.053	2.681	3.608	1.860	3.161

Table 6.16 Differences (%) on accuracy and f1 score when clone apps are removed from the malware dataset

	customsmali	**perm**	**lib**	**allsmali**	**conf**	**dex**	**res**	**ngram**
acc	1.022	1.410	3.513	0.031	1.925	-0.145	1.183	0.477
f1	1.027	1.702	3.625	-0.057	1.924	-0.250	1.356	0.476

	SPL + AppAuth + StringAA	**SPL + AppAuth**	**SPL**	**AppAuth**	**StringAA**
acc	0.554	0.278	0.246	0.337	0.477
f1	0.573	0.207	0.207	0.322	0.476

7. GENERAL DISCUSSION

In this study, we explore the use of source code-based features and compare different feature sets in the literature on Android AA. However, this study had a few limitations, detailed below:

7.1. Usage of Native Code

Today, developers are increasingly using native code within Android application packages, where they co-exist and interact with DEX bytecode through the Java Native Interface [103]. In our approach, only the non-native code parts of applications are used to extract features, as in [16]. On the other hand, developers can use Android NDK, which can help developers reuse code libraries to embed in Android apps; therefore, native code written in C/C++ could carry developers' fingerprints. Because our focus is on the use of source code-based features for Android AA in this study, the features that could be extracted from smali files are used. However, native codes could be included in the Android AA in the future.

7.2. Obfuscation

As indicated in Table 6.13, the proposed source code-based features are susceptible to renaming obfuscation techniques. Therefore, different groups of features, such as metadata and string features, can be combined to identify authors.

A whitelist approach was adapted to find TPL used in an application. However, this approach cannot detect obfuscated TPL. When such libraries cannot be identified correctly, they are considered a developer's custom code, which causes an increase in the similarities between applications developed by different authors that use the same libraries. Moreover, such libraries cannot be used as features to distinguish developers. Therefore, recent studies such as LibID [95] and LibRadar [24] can be used to find obfuscated libraries in the future.

7.3. Dataset

The size of the dataset is important for any machine-learning-based approach. Even though the dataset in this study is larger and more comprehensive than those in other studies in the literature, except [15], as shown in Table 7.1, further studies could extend it. One limitation is finding developers with sufficient applications so that the model can learn their styles. Another limitation is the need to find applications with proper descriptions. In this study, we considered only applications that have descriptions written in English. The size of the dataset can be increased by including other languages.

Table 7.1 Dataset information

Dataset	# of app	# of author
Our paper	15,183	680
AppAuth [16]	3,871	273
String Analysis [14]	1,917	59

7.4. Clone/Repackaged Applications

The current study assumes that if an application is signed by a developer, it is written by that developer. This is the same approach taken in previous studies [14, 16]. However, as shown above, there might be clone applications in the application stores; hence, in our dataset, their existence might have affected our results. In this study, a simple approach based on code similarity was employed to detect app clones. Many researchers have been working on detecting app clones or repackaged applications effectively [46]. It is stated that [46] while there is widespread recognition of the app repackaging issue in both academic and industrial circles, there's a notable lack of datasets to aid research. Creating a comprehensive set of repackaging pairs that serves as the ground truth requires significant effort. With the introduction of a new, large dataset in [46], we expect such studies to accelerate. For these reasons, the investigation of the effects of app clones on Android AA is left for future work.

7.5. Applications Developed by Multiple Authors

Kalgutkar et al. [27] highlighted that many studies operate under the assumption that individual applications in their dataset are authored by a singular developer. This is in contrast to real-world scenarios where software applications often involve contributions from multiple developers. Gong et al. further expressed this point, indicating that between 10% and 60% of source files can contain contributions from unattributed authors, and as many as 75.4% of source files present multiple authorships [104]. For this study, we assume that a single developer is responsible for a given application. However, it is imperative to acknowledge that many applications, particularly expansive commercial software, are the collective efforts of multiple developers. This multiplicity adds layers of complexity to the issue at hand. Notably, when developers adhere to institutionalized guidelines for software development, the overarching organization can be treated as a singular entity or developer within the scope of this study. For instance, even on Github, some developers specify a set of rules or guidelines that contributors need to follow to contribute to a project; this is typically called a CONTRIBUTING guide or CONTRIBUTING.md file. Companies can also implement regular code review processes. This ensures that all codes are checked for quality and adherence to standards before they are merged into the main codebase. The unified coding standards establish a set of coding standards that all developers in the company must follow. Therefore, if some of the codes are written by multiple developers, the total code of the application can be seen as the work of a single developer. This ensures consistency in the codebase, thus making it easier to read, maintain, and debug.

8. CONCLUSION

This study investigates the use of various features for Android AA. The proposed model uses a feature extraction process that starts with compiling *APK* files using "apktool". This step allows the extraction of various files, including small files (which provide Android-specific low-level, assembly-like code), *.dex* files, resource files, manifest files, and XML files. Three main categories of features are extracted from these files:

SPL Features: These include source code-based, which are primarily extracted from smali files, permissions, third-party libraries (TPL), and metadata-based features. The study utilizes 18 source code-based features, adapting them for compatibility with smali codes. Additionally, the model considers each Android permission in the manifest file as a separate feature, adding 158 binary features. TPL features are included by building a whitelist of common libraries and identifying their usage in applications. The model also analyzes application descriptions provided by developers, using a similar method to that used for extracting string features, recognizing that these descriptions can be beneficial in authorship attribution (AA).

AppAuth Features[16]: They include configuration features from the manifest file, resource features from the files/folders in the "res" directory, and dex features related with methods, classes, fields, and data structures in *.dex* files.

StringAA Features[14]: These are extracted from strings contained in XML and *.dex* files of applications. This study uses 3-grams for string-based features, with a tag inserted at the beginning and end of each string to preserve all lines.

To evaluate the effectiveness of these features, experiments were conducted on datasets containing benign, malicious, and obfuscated applications obtained from various platforms. The dataset contains 10,385 benign Android apps from 488 unique developers, each contributing more than 10 apps, collected from five different Android markets. There are also over 3,000 malicious apps from 153 developers, obtained from various sources. Additionally, approximately 6,000 obfuscated applications created

using the “Obfuscapk” tool and more than 1,000 applications are available for version analysis; both subsets are derived from the benign dataset.

Authorship attributions in the literature are based on the hypothesis that applications signed with the same private (signing) key belong to a unique developer. Keys can be extracted from *APK* files using the “print-apk-signature” tool. In cases where multiple developers have the same or similar application, only a single unique example was selected.

The feature processing phase involves generating feature vectors for each application, handling missing values, and standardizing features. The SimpleImputer function from the scikit-learn library is used to replace missing values across each column. The StandardScaler function is applied to standardize the features, allowing features with different scales or units to be directly compared and combined in models. To improve model efficiency and performance, a two-phase dimensional reduction process is applied to the dataset. The first phase involves eliminating features with zero variance, which do not provide discriminatory information and might lead to overfitting. In the second phase, the most relevant features are selected using univariate statistical tests (ANOVA). This helps reduce the dimensionality of the dataset by retaining only the most informative features, which can be particularly beneficial when dealing with high-dimensional data.

This study uses various classification algorithms to predict authorship in the Android domain. The algorithms include Random Forest, K-Nearest Neighbors, SVM, Gaussian Naive Bayes, and LightGBM. The GridSearchCV (Grid Search Cross-Validation) technique is used to determine the optimal values for a given model. Models are then evaluated using these optimal parameters through stratified 10-fold cross-validation, which preserves positive and negative examples across folds.

This study addresses several research questions (RQs) related to Android Authorship Attribution. Our findings can be summarized as follows:

- We investigate which classification algorithm is best for attributing applications to authors. The Random Forest algorithm emerges as the most efficient in terms of both accuracy and time. LightGBM has superior accuracy; however, its classification time is a bit longer.

- After that, we determine both the best "n" value and the number of n-gram features. Here, a set size of 10,000 3-grams is identified as ideal, balancing accuracy and training time. Even though accuracy improves when we increase the number of 3-grams, classification time exponentially increases after 10,000 3-grams.

- After compiling the *APK* file, the TPL codes used in the development of the *APK* file are converted into small files as well as user-defined codes. As developers tend to use the same TPL for their different applications, TPL codes may be insightful in distinguishing authors from each other. However, if two developers create similar applications with the same functionality, they mostly use the same TPLs. Therefore, most of their codes are similar, making it harder to find distinct features between them. As a result, we demonstrate the differences between when we include TPl and when we do not include TPL. The results show that using TPL features as a distinct feature set is more effective than getting results by incorporating TPL into source code-based features.

- The findings indicate that the feature sets introduced in this study achieve high levels of accuracy in Android AA, rivaling that of string-based features, which require more computational resources. Moreover, these newly proposed features surpass other features in the context of AA. While string features are generally superior, practicality issues arise in large datasets. A combination of all features is recommended if feasible.

- Application metadata, like descriptions, aids in attributing developers due to varying writing styles. However, multiple languages in descriptions can hinder classification. Notably, it is observed that metadata features contribute to

improved accuracy, particularly when combined with source code-based features. Of particular significance, this is the first study to explore the impact of metadata information, which describes applications, on Android AA.

- Using only 60% of the features can still yield satisfactory results, reducing resource expenditures.
- A balanced dataset with sufficient samples per developer enhances model performance.
- The model effectively identifies different versions of the same applications.
- All features, especially source code strings, are vulnerable to obfuscation techniques, notably renaming.

Overall, the present work provides a rigorous analysis of Android AA and is compared with state-of-the-art techniques. Future research should investigate the use of this approach for clone applications, particularly those with malicious intent. It is important to first find similarities between malicious software from different sources to predict how malware will evolve; hence, we could develop new protection techniques against malware that will appear in the future. The features related to the author of malware can be useful to achieve this goal. Hence, new applications in the market can be analyzed from their authors point of view.

REFERENCES

[1] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who wrote this code? identifying the authors of program binaries. In Vijay Atluri and Claudia Diaz, editors, *Computer Security – ESORICS 2011*, pages 172–189. Springer Berlin Heidelberg, Berlin, Heidelberg, **2011**. ISBN 978-3-642-23822-2.

[2] Saed Alrabaee, Paria Shirani, Mourad Debbabi, and Lingyu Wang. On the feasibility of malware authorship attribution. In Frédéric Cuppens, Lingyu Wang, Nora Cuppens-Boulahia, Nadia Tawbi, and Joaquin Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 256–272. Springer International Publishing, Cham, **2017**. ISBN 978-3-319-51966-1.

[3] Robert Layton and Ahmad Azab. Authorship analysis of the zeus botnet source code. In *2014 Fifth Cybercrime and Trustworthy Computing Conference*, pages 38–43. **2014**. doi:10.1109/CTC.2014.14.

[4] Mamoun Alazab, Robert Layton, Roderic Broadhurst, and Brigitte Bouhours. Malicious spam emails developments and authorship attribution. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 58–68. **2013**. doi:10.1109/CTC.2013.16.

[5] Robert Layton, Stephen McCombie, and Paul Watters. Authorship attribution of irc messages using inverse author frequency. In *2012 Third Cybercrime and Trustworthy Computing Workshop*, pages 7–13. **2012**. doi:10.1109/CTC.2012.11.

[6] Steven Burrows and Seyed MM Tahaghoghi. Source code authorship attribution using n-grams. In *Proceedings of the twelth Australasian document computing symposium, Melbourne, Australia, RMIT University*, pages 32–39. Citeseer, **2007**.

[7] Rong Chen, Lina Hong, Chunyan Chunyan Lü, and Wu Deng. Author identification of software source code with program dependence graphs. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 281–286. **2010**. doi:10.1109/COMPSACW.2010.56.

[8] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Source code author identification based on n-gram author profiles. In Ilias Maglogiannis, Kostas Karpouzis, and Max Bramer, editors, *Artificial Intelligence Applications and Innovations*, pages 508–515. Springer US, Boston, MA, **2006**. ISBN 978-0-387-34224-5.

[9] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. A probabilistic approach to source code authorship identification. In *Fourth International Conference on Information Technology (ITNG'07)*, pages 243–248. **2007**. doi:10.1109/ITNG.2007.17.

[10] Ivan Krsul and Eugene H. Spafford. Authorship analysis: identifying the author of a program. *Computers & Security*, 16(3):233–257, **1997**. ISSN 0167-4048. doi:https://doi.org/10.1016/S0167-4048(97)00005-9.

[11] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 255–270. USENIX Association, USA, **2015**. ISBN 9781931971232.

[12] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. Oba2: An onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103, **2014**. ISSN 1742-2876. doi:https://doi.org/10.1016/j.diin.2014.03.012. Proceedings of the First Annual DFRWS Europe.

[13] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). *Digital Investigation*, 18:S11–S22, **2016**. ISSN 1742-2876. doi:https://doi.org/10.1016/j.diin.2016.04.002.

[14] Vaibhavi Kalgutkar, Natalia Stakhanova, Paul Cook, and Alina Matyukhina. Android authorship attribution through string analysis. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES '18. Association for Computing Machinery, New York, NY, USA, **2018**. ISBN 9781450364485. doi:10.1145/3230833.3230849.

[15] Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. Authorship attribution of android apps. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, page 277–286. Association for Computing Machinery, New York, NY, USA, **2018**. ISBN 9781450356329. doi:10.1145/3176258.3176322.

[16] Guoai Xu, Chengpeng Zhang, Bowen Sun, Xinyu Yang, Yanhui Guo, Chengze Li, and Haoyu Wang. Appauth: Authorship attribution for android app clones. *IEEE Access*, 7:141850–141867, **2019**. doi:10.1109/ACCESS.2019.2944684.

[17] Wei Wang, Guozhu Meng, Haoyu Wang, Kai Chen, Weimin Ge, and Xiaohong Li. A3ident: A two-phased approach to identify the leading authors of android apps. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 617–628. **2020**. doi:10.1109/ICSME46990.2020.00064.

[18] StatCounter. Mobile operating system market share worldwide. `https://gs.statcounter.com/os-market-share/mobile/worldwide`, **2023**. Accessed: 2023-12-12.

[19] Google. Android apps on google play. `https://play.google.com/store/apps`, **2023**. Accessed: 2023-12-12.

[20] Aptoide. Aptoide - the alternative android app store. `https://en.aptoide.com`, **2023**. Accessed: 2023-12-12.

[21] APKMirror. Apkmirror - free apk downloads - free and safe android apk downloads. `https://www.apkmirror.com`, **2023**. Accessed: 2023-12-12.

[22] Android Developers. String resources | android developers. `https://developer.android.com/guide/topics/resources/string-resource`, **2023**. Accessed: 2023-12-12.

[23] Haoyu Wang, Yao Guo, Zihao Tang, Guangdong Bai, and Xiangqun Chen. Reevaluating android permission gaps with static and dynamic analysis. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. **2015**. doi:10.1109/GLOCOM.2015.7417621.

[24] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 653–656. **2016**.

[25] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 71–82. Association for Computing Machinery, New York, NY, USA, **2015**. ISBN 9781450336208. doi:10.1145/2771783.2771795.

[26] Haoyu Wang and Yao Guo. Understanding third-party libraries in mobile app analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 515–516. **2017**. doi:10.1109/ICSE-C.2017.161.

[27] Vaibhavi Kalgutkar, Ratinder Kaur, Hugo Gonzalez, Natalia Stakhanova, and Alina Matyukhina. Code authorship attribution: Methods and challenges. *ACM Comput. Surv.*, 52(1), **2019**. ISSN 0360-0300. doi:10.1145/3292577.

[28] Hugo F. Gonzalez Robledo. *An automatic authorship attribution technique for Android applications*. Ph.D. thesis, The University of New Brunswick, **2017**.

[29] Connor Tumbleson. Apktool - a tool for reverse engineering android apk files. `https://ibotpeaches.github.io/Apktool/`, **2023**. Accessed: 2023-12-12.

[30] scikit learn. sklearn.impute.simpleimputer - scikit-learn 1.3.2 documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html`, **2024**. Accessed: 2024.

[31] scikit learn. sklearn.preprocessing.standardscaler - scikit-learn 1.3.2 documentation. `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`, **2024**. Accessed: 2024.

[32] scikit-learn developers. scikit-learn: machine learning in python — scikit-learn 1.3.2 documentation. `https://scikit-learn.org/stable/index.html`, **2023**. Accessed: 2023-12-12.

[33] Haibo He and Yunqian Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications*, pages i–xi. John Wiley & Sons, Ltd, **2013**. ISBN 9781118646106. doi:https://doi.org/10.1002/9781118646106.

[34] Scrapy. Scrapy | a fast and powerful scraping and web crawling framework. `https://scrapy.org`, **2023**. Accessed: 2023-12-12.

[35] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. **2014**. doi:10.14722/ndss.2014.23247.

[36] PRALab. Android praguard dataset. `http://pralab.diee.unica.it/en/AndroidPRAGuardDataset`, **2021**. Accessed: 2021.

[37] APKPure. Download apk on android with free online apk downloader - apkpure. `https://apkpure.com`, **2023**. Accessed: 2023-12-12.

[38] 1Mobile-Market. 1mobile market - best google android apps market. `http://market.1mobile.com`, **2018**. Accessed: 2018.

[39] Canadian Institute for Cybersecurity. Datasets | research | canadian institute for cybersecurity | unb. `https://www.unb.ca/cic/datasets/index.html`, **2023**. Accessed: 2023-12-12.

[40] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. Rmvdroid: Towards a reliable android malware dataset with app metadata. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 404–408. **2019**. doi:10.1109/MSR.2019.00067.

[41] warren-bank. Github - warren-bank/print-apk-signature: print information about the certificate used to sign an android apk file. `https://github.com/warren-bank/print-apk-signature`, **2023**. Accessed: 2023-12-12.

[42] Gaurav Goel, Harsh Bhardwaj, Ishika Hooda, and Shailender Kumar. Optimal n-gram subset extraction for accelerating evaluation using genetic algorithm. In *2020 International Conference for Emerging Technology (INCET)*, pages 1–5. **2020**. doi:10.1109/INCET49848.2020.9154083.

[43] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 136–143. **2017**. doi:10.1109/Trustcom/BigDataSE/ICESS.2017.230.

[44] Claudiu Georgiu. Github - claudiugeorgiu/obfuscapk: An automatic obfuscation tool for android apps that works in a black-box fashion, supports advanced obfuscation features and has a modular architecture easily extensible with new techniques. `https://github.com/ClaudiuGeorgiu/Obfuscapk`, **2023**. Accessed: 2023-12-12.

[45] Byoungchul Kim, Kyeonghwan Lim, Seong-Je Cho, and Minkyu Park. Romadroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app's manifest file. *IEEE Access*, 7:72182–72196, **2019**. doi:10.1109/ACCESS.2019.2920314.

[46] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 47(4):676–693, **2021**. doi:10.1109/TSE.2019.2901679.

[47] Kaspersky. Google play malware clocks up more than 600 million downloads in 2023. `https://www.kaspersky.com/blog/malware-in-google-play-2023/49579/`, **2023**. Accessed: 2023-12-12.

[48] AV-TEST. Malware statistics & trends report. `https://www.av-test.org/en/statistics/malware/`, **2023**. Accessed: 2023-12-12.

[49] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109. **2012**. doi:10.1109/SP.2012.16.

[50] Kursat Aktas and Sevil Sen. Updroid: Updated android malware and its familial classification. In Nils Gruschka, editor, *Secure IT Systems*, pages 352–368. Springer International Publishing, Cham, **2018**. ISBN 978-3-030-03638-6.

[51] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, **2021**. doi:10.18653/v1/2021.emnlp-main.685.

[52] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, **2021**.

[53] Rosa María Coyotl-Morales, Luis Villaseñor-Pineda, Manuel Montes-y Gómez, and Paolo Rosso. Authorship attribution using word sequences. In José Francisco Martínez-Trinidad, Jesús Ariel Carrasco Ochoa, and Josef Kittler, editors, *Progress in Pattern Recognition, Image Analysis and Applications*, pages 844–853. Springer Berlin Heidelberg, Berlin, Heidelberg, **2006**. ISBN 978-3-540-46557-7.

[54] Patrick Juola. Authorship attribution. *Found. Trends Inf. Retr.*, 1(3):233–334, **2006**. ISSN 1554-0669. doi:10.1561/1500000005.

[55] Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 932–944. Association for Computing Machinery, New York, NY, USA, **2021**. ISBN 9781450385626. doi:10.1145/3468264.3468606.

[56] O. de Vel, A. Anderson, M. Corney, and G. Mohay. Mining e-mail content for author identification forensics. *SIGMOD Rec.*, 30(4):55–64, **2001**. ISSN 0163-5808. doi:10.1145/604264.604272.

[57] Jason Gray, Daniele Sgandurra, and Lorenzo Cavallaro. Identifying authorship style in malicious binaries: Techniques, challenges & datasets, **2021**.

[58] Sangwoo Lee and Jungwon Cho. Malware authorship attribution model using runtime modules based on automated analysis. *International Journal on Informatics Visualization*, 6(1-2), **2022**. doi:10.30630/joiv.6.1-2.941.

[59] Frank Smallenburg Emanuele Boattini, Michel Ram and Laura Filion. Neural-network-based order parameters for classification of binary hard-sphere crystal structures. *Molecular Physics*, 116(21-22):3066–3075, **2018**. doi:10.1080/00268976.2018.1483537.

[60] Xinyu Yang, Guoai Xu, Qi Li, Yanhui Guo, and Miao Zhang. Authorship attribution of source code by using back propagation neural network based on particle swarm optimization. *PLOS ONE*, 12(11):1–18, **2017**. doi:10.1371/journal.pone.0187204.

[61] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. Source code authorship attribution using long short-term memory based networks. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 65–82. Springer International Publishing, Cham, **2017**. ISBN 978-3-319-66402-6.

[62] E. Dauber, A. Caliskan, R. Harang, and R. Greenstadt. Poster: Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. In *2018 IEEE/ACM 40th International Conference on Software*

Engineering: Companion (ICSE-Companion), pages 356–357. **2018**. ISSN 2574-1934.

[63] Steven Burrows, Alexandra L. Uitdenbogerd, and Andrew Turpin. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience*, 44(1):1–32, **2014**. doi:https://doi.org/10.1002/spe.2146.

[64] Georgia Frantzeskou, Stephen G. MacDonell, and Efstathios Stamatatos. *Source Code Authorship Analysis For Supporting the Cybercrime Investigation Process*, pages 470–495. IGI Global, **2010**. doi:10.4018/978-1-60566-836-9.ch020.

[65] Rong Zheng, Yi Qin, Zan Huang, and Hsinchun Chen. Authorship analysis in cybercrime investigation. In Hsinchun Chen, Richard Miranda, Daniel D. Zeng, Chris Demchak, Jenny Schroeder, and Therani Madhusudan, editors, *Intelligence and Security Informatics*, pages 59–73. Springer Berlin Heidelberg, Berlin, Heidelberg, **2003**. ISBN 978-3-540-44853-2.

[66] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proceedings 2018 Network and Distributed System Security Symposium*, NDSS 2018. Internet Society, **2018**. doi:10.14722/ndss.2018.23304.

[67] Sangwoo Lee and Jungwon Cho. Malware authorship attribution model using runtime modules based on automated analysis. *JOIV : International Journal on Informatics Visualization*, 6:214, **2022**. doi:10.30630/joiv.6.1-2.941.

[68] Anthony Desnos. Android: Static analysis using similarity distance. In *2012 45th Hawaii International Conference on System Sciences*, pages 5394–5403. **2012**. doi:10.1109/HICSS.2012.114.

[69] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: Fast detection of repackaged applications. In Vijay Atluri and Günther Pernul, editors, *Data and Applications Security and Privacy XXVIII*, pages 130–145. Springer Berlin Heidelberg, Berlin, Heidelberg, **2014**. ISBN 978-3-662-43936-4.

[70] Google Play. Use play app signing. `https://support.google.com/googleplay/android-developer/answer/9842756`, **2023**. Accessed: 2023-12-12.

[71] Kanae Yoshida, Hironori Imai, Nana Serizawa, Tatsuya Mori, and Akira Kanaoka. Understanding the origins of weak cryptographic algorithms used for signing android apps. *Journal of Information Processing*, 27:593–602, **2019**. doi:10.2197/ipsjjip.27.593.

[72] Huawei. Huawei appgallery - huawei türkiye. `https://consumer.huawei.com/tr/mobileservices/appgallery/`, **2024**. Accessed: 2024.

[73] Tencent. Tencent appstore. `https://appstore.tencent.com/`, **2024**. Accessed: 2024.

[74] Kaspar Hageman, Álvaro Feal, Julien Gamba, Aniketh Girish, Jakob Bleier, Martina Lindorfer, Juan Tapiador, and Narseo Vallina-Rodriguez. Mixed signals: Analyzing software attribution challenges in the android ecosystem. *IEEE Transactions on Software Engineering*, 49(4):2964–2979, **2023**. doi:10.1109/TSE.2023.3236582.

[75] apkmonk. Download android app apks free. `https://www.apkmonk.com/`, **2024**. Accessed: 2024.

[76] Baidu. https://shouji.baidu.com/, **2024**. Accessed: 2024.

[77] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, **2001**. ISSN 1573-0565. doi:10.1023/A:1010933404324.

[78] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer New York, NY, 1 edition, **1995**. ISBN 978-1-4757-2440-0. doi:10.1007/978-1-4757-2440-0.

[79] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, **2008**. ISSN 0219-3116. doi:10.1007/s10115-007-0114-2.

[80] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *IEEE Transactions on Neural Networks*, 13(2):415–425, **2002**. doi:10.1109/72.991427.

[81] Harry Zhang. The optimality of naive bayes. In *The Florida AI Research Society*, volume 2. **2004**.

[82] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3149–3157. Curran Associates Inc., Red Hook, NY, USA, **2017**. ISBN 9781510860964.

[83] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review*, 54(3):1937–1967, **2021**. ISSN 1573-7462. doi:10.1007/s10462-020-09896-5.

[84] scikit-learn developers. sklearn.model_selection.gridsearchcv — scikit-learn 1.3.2 documentation. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html, **2023**. Accessed: 2023-12-12.

[85] scikit-learn developers. 3.2. tuning the hyper-parameters of an estimator — scikit-learn 1.3.2 documentation. https://scikit-learn.org/stable/modules/grid_search.html, **2023**. Accessed: 2023-12-12.

[86] skylot. Github - skylot/jadx: Dex to java decompiler. https://github.com/skylot/jadx, **2023**. Accessed: 2023-12-12.

[87] MVNRepository. Maven repository: Central. https://mvnrepository.com/repos/central, **2023**. Accessed: 2023-12-12.

[88] Sevil Sen and Burcu Can. Android security using nlp techniques: A review, **2021**.

[89] PrivacyGrade. Privacygrade:grading the privacy of smartphone apps. http://privacygrade.org, **2021**. Accessed: 2021.

[90] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414. **2016**. doi:10.1109/SANER.2016.52.

[91] Georgia Kapitsaki and Modestos Ioannou. Examining the privacy vulnerability level of android applications. In *Proceedings of the 15th International Conference on Web Information Systems and Technologies*, WEBIST 2019, page 34–45. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, **2019**. ISBN 9789897583865. doi:10.5220/0007955100340045.

[92] Huseyin Alecakir, Burcu Can, and Sevil Sen. Attention: there is an inconsistency between android permissions and application metadata! *International Journal of Information Security*, 20(6):797–815, **2021**. ISSN 1615-5270. doi:10.1007/s10207-020-00536-1.

[93] Saed Alrabaee, Paria Shirani, Lingyu Wang, Mourad Debbabi, and Aiman Hanna. Decoupling coding habits from functionality for effective binary authorship attribution. *Journal of Computer Security*, 27:1–36, **2019**. doi:10.3233/JCS-191292.

[94] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 356–367. Association for Computing Machinery, New York, NY, USA, **2016**. ISBN 9781450341394. doi:10.1145/2976749.2978333.

[95] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. Libid: Reliable identification of obfuscated third-party android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 55–65. Association for Computing Machinery, New York, NY, USA, **2019**. ISBN 9781450362245. doi:10.1145/3293882.3330563.

[96] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 21. USENIX Association, USA, **2011**.

[97] Danah Boyd, Scott Golder, and Gilad Lotan. Tweet, tweet, retweet: Conversational aspects of retweeting on twitter. In *2010 43rd Hawaii International Conference on System Sciences*, pages 1–10. **2010**. doi:10.1109/HICSS.2010.412.

[98] Na Cheng, R. Chandramouli, and K.P. Subbalakshmi. Author gender identification from text. *Digital Investigation*, 8(1):78–88, **2011**. ISSN 1742-2876. doi:https://doi.org/10.1016/j.diin.2011.04.002.

[99] Ying Zhao and Justin Zobel. Searching with style: Authorship attribution in classic literature. In *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ACSC '07, page 59–68. Australian Computer Society, Inc., AUS, **2007**. ISBN 1920682430.

[100] Android Developers. Shrink, obfuscate, and optimize your app | android studio | android developers. `https://developer.android.com/studio/build/shrink-code`, **2023**. Accessed: 2023-12-12.

[101] Smardec Inc. Allatori java obfuscator - professional java obfuscation. `http://www.allatori.com`, **2023**. Accessed: 2023-12-12.

[102] PreEmptive Solutions. Dasho | preemptive. `https://www.preemptive.com/products/dasho`, **2023**. Accessed: 2023-12-12.

[103] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1232–1244. Association for Computing Machinery, New York, NY, USA, **2022**. ISBN 9781450392211. doi:10.1145/3510003.3512766.

[104] Siyi Gong and Hao Zhong. Code authors hidden in file revision histories: An empirical study. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 71–82. **2021**. doi:10.1109/ICPC52881.2021.00016.