5. Shared Code Analysis: Paylaşılan Kod Analiz

Benzerlik analizi olarak da adlandırılan paylaşılan kod analizi, paylaştıkları ön derleme kaynak kodunun yüzdesini tahmin ederek iki kötü amaçlı yazılım örneğini karşılaştırdığımız süreçtir.  Kötü amaçlı yazılım örneklerini harici özniteliklerine (örneğin kullandıkları masaüstü simgeleri veya çağırdıkları sunucular) göre karşılaştıran paylaşılan öznitelik analizinden farklıdır.

Yeni bir örnek verildiğinde, paylaşılan kod tahmini, muhtemelen hangi örnekleri paylaştığı ve bu örnekler hakkında bildiklerimizi saniyeler içinde görmemizi sağlar. Bu örnekte, çok benzer bir örneğin bilinen bir APT'den veya gelişmiş kalıcı tehditten geldiğini ortaya koyuyor ve böylece bu yeni kötü amaçlı yazılım için anında bağlam sağlıyor.Ayrıca, Bölüm 4'te öğrendiğiniz ağ görselleştirmeyi kullanarak örnek paylaşılan kod ilişkilerini görselleştirebiliriz.

Görselleştirmeden de görebileceğiniz gibi, otomatik paylaşılan kod analizi teknikleri, manuel analiz yoluyla keşfedilmesi günler veya haftalar süren kötü amaçlı yazılım ailelerinin varlığını hızla ortaya çıkarabilir. Bu bölümde, aşağıdakileri yapmak için bu teknikleri kullanmayı öğreneceksiniz:

• Aynı kötü amaçlı yazılım araç setlerinden gelen veya aynı saldırganlar tarafından yazılmış yeni kötü amaçlı yazılım ailelerini belirleyin.

• Yeni bir örnek ile daha önce görülen örnekler arasındaki kod benzerliğini belirleyin.

• Kötü amaçlı yazılım örnekleri arasındaki kod paylaşım modellerini daha iyi anlamak ve sonuçlarınızı başkalarına iletmek için kötü amaçlı yazılım ilişkilerini görselleştirin.

• Bu kitap için oluşturduğum, bu fikirleri uygulayan ve kötü amaçlı yazılım paylaşılan kod ilişkilerini görmenize olanak tanıyan iki kavram kanıtı aracını kullanın.

Bu bölümde, deneylerimizi yapmak için önemli miktarda kodu birbiriyle paylaşan gerçek dünyadaki kötü amaçlı yazılım ailelerini kullanıyoruz. Bu veri kümeleri, bu örnekleri küratörlüğünü yapan ve araştırma topluluğunun kullanımına sunan Mandiant ve Mila Parkour sayesinde elde edildi. Ancak gerçekte, bir kötü amaçlı yazılım örneğinin hangi aileye ait olduğunu veya yeni kötü amaçlı yazılım örneklerinizin daha önce görülen örneklere ne ölçüde benzediğini bilmiyor olabilirsiniz. Ancak bildiğimiz örneklerden geçmek iyi bir uygulama olacaktır, çünkü otomatikleştirilmiş örnek benzerliği çıkarımlarımızın hangi örneklerin gerçekte aynı gruba ait olduğuna dair bilgilerimizle aynı çizgide olduğunu doğrulamamıza olanak tanır., Kaspersky'nin yüksek güvenle algıladığı yedi kötü amaçlı yazılım sınıfı seçtim. Bunlar özellikle dapato, pasta, skor, vbna, webprefix, xtoober ve zango ailelerini içerir.

Şekil 5-2: Kötü amaçlı yazılım kod paylaşım analizi için "özellik çantası" modelinin bir resmi

Dinamik kötü amaçlı yazılım analizi durumunda, örnekleri yalnızca paylaştıkları davranışlara değil, aynı zamanda davranışları ifade ettikleri sıraya veya davranış dizileri olarak adlandırdığımıza göre karşılaştırmak isteyebiliriz. Sıralı bilgileri kötü amaçlı yazılım örnek karşılaştırmalarına dahil etmenin yaygın bir yolu, özellikler çantası modelini N-gram kullanarak sıralı verileri barındıracak şekilde genişletmektir.

->What are N-Grams? nedir?:

Bir N-gram, bazı daha büyük olaylar dizisinin belirli bir uzunluğuna (N) sahip bir olaylar alt dizisidir. Sıralı verilerin üzerine bir pencere kaydırarak bu alt diziyi daha büyük bir diziden çıkarıyoruz.tamsayı dizisi (1,2,3,4,5,6,7), 3 uzunluğunun beş farklı alt dizisine çevrilmiştir: (1,2,3), (2,3,4) , (3,4,5), (4,5,6), (5,6,7).

->Using the Jaccard Index to Quantify Similarity:Benzerliği Ölçmek için Jaccard Endeksini Kullanma

Bir kötü amaçlı yazılım örneğini bir dizi özellik olarak temsil ettikten sonra, söz konusu örneğin özellik çantası ile başka bir örneğin özellik çantası arasındaki benzerlik derecesini ölçmeniz gerekir. İki kötü amaçlı yazılım örneği arasındaki kod paylaşımının kapsamını tahmin etmek için aşağıdaki özelliklere sahip olması gereken bir benzerlik işlevi kullanıyoruz:

* Kötü amaçlı yazılım örnekleri çiftleri arasındaki tüm benzerlik karşılaştırmalarının ortak bir ölçeğe yerleştirilebileceği şekilde normalleştirilmiş bir değer verir. Geleneksel olarak, işlev 0 (kod paylaşımı yok) ile 1 (örnekler kodlarının yüzde 100'ünü paylaşır) arasında değişen bir değer vermelidir.
* İşlev, iki örnek arasındaki kod paylaşımının doğru tahminlerini yapmamıza yardımcı olmalıdır (bunu deney yoluyla deneysel olarak belirleyebiliriz).
* İşlev modellerinin benzerliği neden iyi kodladığını kolayca anlayabilmeliyiz (anlamak veya açıklamak için çok çaba gerektiren karmaşık bir matematiksel kara kutu olmamalıdır).

Jaccard indeksi, bu özelliklere sahip basit bir fonksiyondur. Aslında, kod benzerlik tahminine yönelik diğer matematiksel yaklaşımlar güvenlik araştırma topluluğunda denenmiş olsa da (örneğin, kosinüs mesafesi, L1 mesafesi, Öklid [L2] mesafesi vb.), Jaccard indeksi en yaygın olarak ortaya çıkmıştır.İki grup kötü amaçlı yazılım özelliği arasındaki örtüşme derecesini basit ve sezgisel olarak ifade ederek, her iki grupta da var olan benzersiz özelliklerin yüzdesine göre normalleştirilmiş iki kümede ortak olan benzersiz özelliklerin yüzdesini verir.

Şekil 5-4, Jaccard indeks değerlerinin örneklerini göstermektedir.

Bu, dört çift kötü amaçlı yazılım örneğinden çıkarılan dört çift kötü amaçlı yazılım özelliğini gösterir. Her görüntü, iki küme arasında paylaşılan özellikleri, iki küme arasında paylaşılmayan özellikleri ve verilen kötü amaçlı yazılım örnekleri çifti ve ilişkili özellikler için ortaya çıkan Jaccard indeksini gösterir. Örnekler arasındaki Jaccard indeksinin, örnekler arasında paylaşılan özellik sayısının Venn diyagramında çizilen toplam özellik sayısına bölünmesiyle elde edildiğini görebilirsiniz.

->Using Similarity Matrices to Evaluate Malware Shared Code Estimation Methods: Kötü Amaçlı Yazılım Paylaşılan Kod Tahmin Yöntemlerini Değerlendirmek için Benzerlik Matrislerini Kullanma

iki malware örneğinin aynı aileden gelip gelmediğini belirlemek için dört yönteme bakalım:komut dizisi tabanlı benzerlik, dizelere dayalı benzerlik, İçe Aktarma Adres Tablosu Import Address Table tabanlı benzerlik ve dinamik API çağrısı dynamic api call tabanlı benzerlik. Bu dört yöntemi karşılaştırmak için bir benzerlik matrisi görselleştirme tekniği kullanacağız.Buradaki amacımız, örnekler arasındaki paylaşılan kod ilişkilerini aydınlatma yeteneği açısından her yöntemin göreceli güçlü ve zayıf yönlerini karşılaştırmak olacaktır. Başlamak için, benzerlik matrisi kavramının üzerinden geçelim.

Bu matris, tüm örnekler arasındaki benzerlik ilişkisini görmenizi sağlar.Bu matriste bir miktar alanın boşa gittiğini görebilirsiniz.

Şekilde gösterilen çok sayıda kötü amaçlı yazılım örneği nedeniyle, her benzerlik değerinin gölgeli bir pikselle temsil edildiğini unutmayın. Her numunenin ismini vermek yerine, her numunenin aile isimlerini yatay ve dikey eksenler boyunca veririz. Mükemmel bir benzerlik matrisi, her bir aileyi temsil eden satırlar ve sütunlar bir arada gruplandırıldığından ve belirli bir ailenin tüm üyelerinin birbirine benzer olmasını beklediğimizden, sol üstten sağ alta çapraz olarak uzanan bir beyaz kareler zinciri gibi görünecektir. , ancak diğer ailelerden örnekler değil.

->Instruction Sequence-Based Similarity:Talimat Sırasına Dayalı Benzerlik:

İki kötü amaçlı yazılım ikili dosyasını, paylaştıkları kod miktarı açısından karşılaştırmanın en sezgisel yolu, x86 birleştirme talimatları dizilerini karşılaştırmaktır, çünkü komut dizilerini paylaşan örnekler, derlemeden önce gerçek kaynak kodunu paylaşmış olabilir. malwareleri disassemly ile bakmamız gerekir. Ardından, kötü amaçlı yazılım dosyasının .text bölümünde göründükleri sırayla talimat dizilerini çıkarmak için daha önce tartıştığım N gram çıkarma yaklaşımını kullanabiliriz. . Son olarak, ne kadar kod paylaştıklarını düşündüğümüzü tahmin etmek için örnekler arasındaki Jaccard indekslerini hesaplamak için N-gram talimatını kullanabiliriz.N büyük olursa yanlışlar çoğalır karşılaştırmada.N'yi küçültürsende ince benzerlikler artar bu sefer yanlış pozitifler çoğalır. N'yi 5 ayarlamışlar

Birçok ailenin benzerlik ilişkilerinin kaçtığını görüyoruz.Gördüğünüz gibi, komut alt dizisi paylaşılan kod analizinin bir sınırlaması, örnekler arasındaki birçok kod paylaşım ilişkisini gözden kaçırabilmesidir.bunun nedeni malwarelerin paketlenmiş olabilmesidir. PĞaketten çıkarsak bile kaynak kıodu derleme işlemenin getirmiş olduğu farklılıklardan dolayı benzerlik ilişkilerinde sıkıntı çıkarıyor.Derleyiciler aynı kaynak kodunu farklı assembly talimatları dizileri halinde derleyebilirler.Aynı derleyicide olsa derleyiciyi derlemek için vermiş oldığumuz kod farklılıklarından dolayı da farklar oluşuyor.MEsela hızlşı olması için  –O3  bayrağı ile derlediğin kod ile diğeri arasında fark olur.

Birleştirme talimatlarına baktığımızda çok farklı görünen özdeş C ve C ++ kodu sorununun ötesinde, ikili dosyaları derleme kodlarına göre karşılaştırdığımızda ortaya çıkan ek bir sorun var: birçok kötü amaçlı yazılım ikili dosyası artık yüksek seviyeli dillerde yazılıyor C # gibi. Bu ikili dosyalar, bu üst düzey dillerin bayt kodunu basitçe yorumlayan standart ortak metin derleme kodunu içerir. Bu nedenle, aynı yüksek seviyeli dilde yazılan ikili dosyalar çok benzer x86 talimatlarını paylaşsalar da, gerçek bayt kodları çok farklı kaynak kodlarından geldikleri gerçeğini yansıtabilir.

->Strings-Based Similarity:Dizelere Dayalı Benzerlik:

Örneklerdeki tüm bitişik yazdırılabilir karakter dizilerini çıkararak ve ardından paylaşılan dize ilişkilerine dayalı olarak tüm kötü amaçlı yazılım örneği çiftleri arasındaki Jaccard dizinini hesaplayarak dizelere dayalı kötü amaçlı yazılım benzerliğini hesaplayabiliriz.

Şekil 5-8: Dize özellikleri kullanılarak oluşturulan benzerlik matrisi

Diğer yönteme göre daha başarılıdır ve yedi ailenin tümü için benzerlik ilişkilerinin çoğunu doğru bir şekilde kurtarır. Ancak, yönerge benzerliği yönteminden farklı olarak, xtoober ve dapato'nun bazı kod düzeylerini paylaştığını yanlış tahmin ettiği için birkaç yanlış pozitif vardır. Ayrıca, bu yöntemin bazı ailelerde örnekler arasında benzerlikler tespit etmediğini, özellikle zango, skor ve dapato ailelerinde kötü performans gösterdiğini belirtmek gerekir.

->Import Address Table–Based Similarity:Adres Tablosu Tabanlı Benzerliği İçe Aktar:

Kötü amaçlı yazılım ikili dosyaları tarafından yapılan DLL içe aktarmalarını karşılaştırarak, "Adres Tablosuna dayalı İçe Aktarma benzerliği" dediğim şeyi hesaplayabiliriz. Bu yaklaşımın arkasındaki fikir, kötü amaçlı yazılım yazarı talimatları yeniden sıralasa, kötü amaçlı yazılım ikili programının başlatılmış veri bölümünü karartsa ve hata ayıklayıcı ve anti-VM anti-analiz tekniklerini uygulamış olsa bile, aynı içe aktarma bildirimlerini yerinde bırakmış olabileceğidir. . İçe Aktarma Adres Tablosu yönteminin sonuçları Şekil 5-9'da gösterilmektedir.

Şekil, İçe Aktarma Adres Tablosu yönteminin, webprefix ve xtoober örnekleri arasındaki benzerlik ilişkilerini tahmin etmede önceki yöntemlerden daha iyi olduğunu ve skor, dapato ve vbna ilişkilerinin birçoğunu kaçırmasına rağmen genel olarak çok iyi olduğunu göstermektedir. Ayrıca bu yöntemin deneysel veri kümemizde çok az yanlış pozitif vermesi de dikkate değerdir.

->Dynamic API Call–Based Similarity:Dinamik API Çağrısına Dayalı Benzerlik:

Bu bölümde tanıttığım son karşılaştırma yöntemi dinamik kötü amaçlı yazılım benzerliğidir. Dinamik dizileri karşılaştırmanın avantajı, kötü amaçlı yazılım örnekleri aşırı derecede karıştırılmış veya paketlenmiş olsa bile, aynı koddan türetildikleri veya birbirlerinden kod ödünç aldıkları sürece korumalı alandaki bir sanal makinede benzer eylem dizileri gerçekleştirme eğiliminde olmalarıdır. Bu yaklaşımı uygulamak için, bir sanal alanda kötü amaçlı yazılım örnekleri çalıştırmanız ve yaptıkları API çağrılarını kaydetmeniz, dinamik günlüklerden N-gram API çağrılarını çıkarmanız ve son olarak, paketleri arasında Jaccard indeksini alarak örnekleri karşılaştırmanız gerekir. N-gram. Şekil 5-10, dinamik N-gram benzerlik yaklaşımının çoğu durumda içe aktarma ve dize yöntemlerinin yanı sıra yaklaşık olarak işe yaradığını göstermektedir.

Buradaki kusurlu sonuçlar, bu yöntemin her derde deva olmadığını göstermektedir. Bir sanal alanda basitçe kötü amaçlı yazılım çalıştırmak, birçok davranışını tetiklemek için yeterli değildir. Örneğin, bir komut satırı kötü amaçlı yazılım aracının varyasyonları, önemli bir kod modülünü etkinleştirebilir veya etkinleştirmeyebilir ve bu nedenle kodlarının çoğunu paylaşsalar bile farklı davranış dizileri yürütebilirler.

Diğer bir sorun da, bazı örneklerin korumalı alanda çalıştıklarını algılaması ve ardından yürütmeden derhal çıkması ve karşılaştırma yapmamız için bize çok az bilgi bırakmasıdır. Özetle, özetlediğim diğer benzerlik yaklaşımları gibi, dinamik API çağrı dizisi benzerliği mükemmel değildir, ancak örnekler arasındaki benzerlikler hakkında etkileyici bilgiler sağlayabilir.

->Building a Similarity Graph: Benzerlik Grafiği Oluşturma

bu analizi malware veri kümesi üzerinden gerçekleştiren basit bir sistem oluşturalım. Öncelikle kullanmak istediğimiz özellikleri çıkararak örneklerin paylaştığı kod miktarını tahmin etmemiz gerekiyor. Bunlar, Adres Tablosunu İçe Aktarma tabanlı işlevler, dizeler, N-gram talimat veya N-gram dinamik davranış gibi daha önce açıklanan özelliklerden herhangi biri olabilir. Burada, iyi performans gösterdikleri ve çıkarılması ve anlaşılması kolay oldukları için yazdırılabilir dize özelliklerini kullanacağız.

Dize özelliklerini çıkardıktan sonra, özelliklerini Jaccard indeksini kullanarak karşılaştırarak her kötü amaçlı yazılım örneği çiftini yinelemeliyiz. Ardından bir kod paylaşım grafiği oluşturmamız gerekiyor. Bunu yapmak için, önce iki örneğin ne kadar kod paylaştığını tanımlayan bir eşik üzerinde karar vermemiz gerekiyor - araştırmamda kullandığım standart değer 0,8. Belirli bir kötü amaçlı yazılım örneği çifti için Jaccard endeksi bu değerin üzerindeyse, görselleştirme için aralarında bir bağlantı oluştururuz. Son adım, hangi örneklerin paylaşılan kod ilişkileriyle birbirine bağlı olduğunu görmek için grafiği incelemektir.

ilk olarak kütüphaneleri ekleyip  iki örneğin özellik seti arasındaki Jaccard indeksini hesaplayan jaccard () işlevini bildirir.

Daha sonra, Liste 5-3'te, iki ek yardımcı program işlevi açıklıyoruz: analiz edeceğimiz kötü amaçlı yazılım dosyalarındaki yazdırılabilir dizi dizilerini bulan getstrings () ve hedef dosyaların gerçekten Windows olmasını sağlayan pecheck () PE dosyaları. Bu işlevleri daha sonra hedef kötü amaçlı yazılım ikili dosyalarında özellik çıkarma işlemi gerçekleştirirken kullanacağız.

Ardından, kullanıcımızın komut satırı argümanlarını ayrıştırıyoruz. Bu argümanlar, analiz edeceğimiz kötü amaçlı yazılımın bulunduğu hedef dizini, oluşturduğumuz paylaşılan kod ağını yazacağımız çıktı ".dot" dosyasını ve Jaccard indeksinin ne kadar yüksek olması gerektiğini belirleyen Jaccard indeks eşiğini içerir. programın birbirleriyle ortak bir kod tabanını paylaştıklarına karar vermesi için iki örnek arasında olun.

Ardından, programın ana işini yapmak için daha önce belirttiğimiz yardımcı işlevleri kullanıyoruz: hedef dizinde PE ikili dosyaları bulmak, bunlardan özellikleri çıkarmak ve benzerliği ifade etmek için kullanacağımız bir ağı başlatmak ikili dosyalar arasındaki ilişkiler.

Liste 5-5: Hedef dizindeki PE dosyalarından özelliklerin çıkarılması ve paylaşılan kod ağının başlatılması

Hedef örneklerimiz den özellikleri çıkardıktan sonra, Jaccard indeksini kullanarak özelliklerini karşılaştırarak her kötü amaçlı yazılım örneği çiftini yinelemeliyiz. Bunu Liste 5 6'da yapıyoruz. Jaccard indeksi kullanıcı tanımlı bir eşiğin üzerindeyse, örneklerin birbirine bağlandığı bir kod paylaşım grafiği de oluşturuyoruz. Araştırmamda işe yaradığını bulduğum eşik 0,8.

->Scaling Similarity Comparisons:Benzerlik Karşılaştırmalarını Ölçeklendirme

yukarıdaki kod örneğimiz  küçük kötü amaçlı yazılım veri kümeleri için iyi çalışsa da, çok sayıda kötü amaçlı yazılım örneğinde iyi çalışmaz. Bunun nedeni, bir veri kümesindeki tüm kötü amaçlı yazılım örneği çiftlerinin karşılaştırılmasının örnek sayısı ile ikinci dereceden artmasıdır.

Kötü amaçlı yazılım benzerlik karşılaştırmalarını ölçeklendirmek için, rastgele karşılaştırma yaklaşım algoritmaları kullanmamız gerekir. Temel fikir, hesaplama süresinde bir azalma karşılığında karşılaştırma hesaplamamızda bazı hatalara izin vermektir. Amaçlarımız açısından minhash olarak bilinen yaklaşık bir karşılaştırma yaklaşımı bu amaca güzel bir şekilde hizmet etmektedir. Minhash yöntemi, milyonlarca örnek arasındaki paylaşılan kod ilişkilerini analiz edebilmemiz için önceden tanımlanmış bazı benzerlik eşiğinin altındaki benzer olmayan kötü amaçlı yazılım örnekleri arasındaki hesaplama benzerliklerini önlemek için yaklaşımı kullanarak Jaccard endeksini hesaplamamıza olanak tanır.

->Minhash in Nutshell: Özetle Minhash

Minhash, bir kötü amaçlı yazılım örneğinin özelliklerini alır ve bunları k hash işlevleriyle hash hale getirir. Her bir karma işlevi için, tüm özellikler üzerinden hesaplanan karma değerlerinin yalnızca minimum değerini saklarız, böylece kötü amaçlı yazılım özellikleri kümesi, minhash olarak adlandırdığımız sabit boyutlu k tamsayı dizisine indirgenir. Minhash dizilerine göre iki örnek arasındaki yaklaşık Jaccard indeksini hesaplamak için, şimdi sadece k minhash'lerin kaçının eşleştiğini kontrol etmeniz ve bunu k'ye bölmeniz gerekir.

Sihirli bir şekilde, bu hesaplamalardan düşen sayı, herhangi iki örnek arasındaki gerçek Jaccard indeksinin yakın bir tahmini. Jaccard indeksinin birebir hesaplanması yerine minhash kullanmanın yararı, hesaplamanın çok daha hızlı olmasıdır.

Aslında, minhash'i bir veritabanındaki kötü amaçlı yazılımları akıllıca dizine index  eklemek için bile kullanabiliriz, böylece yalnızca benzer olma olasılığı bulunan kötü amaçlı yazılım örnekleri arasındaki karşılaştırmaları hesaplamamız gerekir, çünkü bunların karmalarından en az biri eşleşir ve böylece kötü amaçlı yazılım veri kümelerindeki benzerliklerin hesaplanmasını önemli ölçüde hızlandırır. .

->Building a Persistent Malware Similarity Search System:  Kalıcı Bir Kötü Amaçlı Yazılım Benzerlik Arama Sistemi Oluşturma

kötü amaçlı yazılım örneklerini dize özelliklerine göre indekslediğim basit bir sistem örneğini gösterir.

Başlangıç olarak programımız için gerekli olan Python paketlerini içe aktarır.burada murmur , shelve ve sim\_graph gibi paketleri ithal ediyorum. Örneğin murmur üfürüm , az önce bahsettiğim minhash algoritmasını hesaplamak için kullandığımız  bir hashing kitaplığıdır.

Python standart kitaplığında bulunan basit bir veritabanı modülü olan shelve'yi, benzerlikleri hesaplamak için kullandığımız örnekler ve minhash'leri hakkında bilgi depolamak için kullanıyoruz. Ayrıca Liste 5-7'de iki sabit beyan ederiz: NUM\_MINHASHES ve SKETCH\_RATIO. Bunlar, her bir örnek için hesapladığımız minhash sayısına ve minhash'lerin  minhashes skeçlere  sketches oranına karşılık gelir. Ne kadar çok mini çizgi ve eskiz kullanırsak, benzerlik hesaplamalarımızın o kadar doğru olduğunu hatırlayın. Örneğin, 256 minhash ve 8: 1 (32 çizim) oranı, düşük bir hesaplama maliyetiyle kabul edilebilir doğruluk sağlamak için yeterlidir.

Liste 5-8, kötü amaçlı yazılım örnek bilgilerini depolamak için kullandığımız raf  shelve veritabanını başlatmak, erişmek ve silmek için kullandığımız veritabanı işlevselliğini uygular.Depoladığımız örnek bilgileri silmek ve baştan başlamak istememiz durumunda programımızın veritabanını silmek için wipe\_database () (1) tanımlarız. Daha sonra veritabanımızı açmak için get\_database () (2) 'yi tanımlarız, henüz yoksa veritabanımızı oluştururuz ve ardından kötü amaçlı yazılım örneklerimiz hakkındaki verileri depolamamıza ve almamıza izin veren bir veritabanı nesnesi döndürürüz.

Liste 5-9, paylaşılan kod analizimiz için kodun temel bir parçasını uygular: minhash.

NUM\_MINHASHES kez (1) döngü yapıyoruz ve bir minhash değeri ekliyoruz. Her bir minhash değeri, tüm özelliklere hashing uygulanarak ve ardından minimum hash değeri alınarak hesaplanır. Bu hesaplamayı gerçekleştirmek için, özellikleri hash etmek için üfürüm murmur paketinin string\_hash () işlevini kullanırız ve ardından Python’un min () işlevini (2) çağırarak karma listesinin minimum değerini alırız.

String\_hash'in ikinci argümanı, karma işlevin tohumun değerine bağlı olarak farklı karmalarla eşleşmesine neden olan bir tohum değeridir.. Her bir minhash değeri, 256 dakikalık hash değerlerimizin tümü aynı olmayacak şekilde benzersiz bir hash işlevi gerektirdiğinden, her yinelemede string\_hash işlevini sayaç değerimiz i ile çekirdeklendiririz, bu da özelliklerin her yinelemede farklı karmalarla eşleşmesine neden olur. çekirdeklendirme iteration  Ardından, hesapladığımız mini karmalar üzerinde döngü oluşturur ve çizimleri  sketches hesaplamak için mini karmaları kullanırız (3).

Eskizlerin sketch , kötü amaçlı yazılım örneklerimizin veritabanını indekslemek için kullandığımız, veritabanını sorgulayarak birbirine benzemesi muhtemel örnekleri hızlı bir şekilde alabilmemiz için kullandığımız birden fazla minhash'in karmaları olduğunu hatırlayın.Bir sonraki kod listesinde, skeçlerimizi elde etmeye giderken her bir karma parçasını karma haline getirerek, örneklemimizin tüm mini karmalarını SKETCH\_RATIO adım boyutunda döngüye alıyoruz. Son olarak, minhash'leri birlikte hash hale getirmek için üfürüm murmur paketinin string\_hash işlevini kullanırız (4).

Liste 5-10, Örnekleri sistemimizin veritabanına endeksleyen bir işlev oluşturmak için Liste 5-8'den get\_database (), içe aktardığımız sim\_graph modülünden getstrings () işlevi ve Liste 5-9'daki minhash () işlevini kullanır.

Get\_database () (1), getstrings () (2) ve minhash () (3) 'ü çağırıyoruz ve sonra (4)' ten başlayarak örneğimizin eskizlerini yineliyoruz. Daha sonra, örneklerimizi veritabanında indekslemek için, örnekleri bir kimlik yerine sketh değerlerine göre saklamamıza izin veren ters indeksleme inverted indexing, olarak bilinen bir teknik kullanıyoruz. Daha spesifik olarak, bir numunenin 32 eskiz değerinin her biri için, o eskizin kayıtlarını veritabanında ararız ve örneğimizin kimliğini bu çizimle ilişkili örnekler listesine ekleriz. Burada kimliği olarak bir örneğin dosya sistemi yolunu kullanıyoruz.

Bunun kodda nasıl uygulandığını görebilirsiniz: bir örnek (4) için hesapladığımız eskizlerin üzerinde döngü yapıyoruz, zaten yoksa çizim için bir kayıt oluşturuyoruz (örneğimizi çizimle ilişkilendirirken biz Anladım) (5) ve son olarak, eskizin kaydı mevcutsa (6) çizimin ilişkili örnek yolları setine örnek yolunu ekliyoruz.

Liste 5-11, iki önemli işlevin bildirimini gösterir: comment\_sample () ve search\_sample ().

Beklendiği gibi comment\_sample () (1), bir örneğin veritabanı kaydına kullanıcı tanımlı bir yorum kaydı ekler. Bu işlevsellik kullanışlıdır, çünkü programın kullanıcılarının tersine mühendislikten elde ettikleri içgörüleri veritabanına bir örnek olarak dahil etmelerine olanak tanır, öyle ki örneklere benzer yeni bir örnek gördüklerinde yorumları daha hızlı anlamak için bu yorumlardan faydalanabilirler. ve yeni numunenin amacı.

Daha sonra search\_sample () (2), bir sorgu örneğine benzer örnekler bulmak için minhash'tan yararlanır. Bunu yapmak için, ilk olarak sorgu örneğinden dize unsurlarını, küçük kareleri ve çizimleri çıkarıyoruz. Daha sonra, aynı taslağa (3) sahip olan veritabanında depolanan örnekleri arayarak, numunenin eskizlerini yineleriz. Sorgu örneğiyle bir taslak paylaşan her örnek için, minhash'leri (4) kullanarak yaklaşık Jaccard indeksini hesaplıyoruz. Son olarak, sorgu örneğine en benzer örnekleri, veritabanında depolanan bu örneklerle ilişkili yorumlarla birlikte kullanıcıya bildiriyoruz (5).

Burada, kullanıcıların veritabanına kötü amaçlı yazılım örnekleri yüklemelerine izin veriyoruz, böylece kullanıcılar veritabanında benzer örnekleri aradığında bu örnekler yeni kötü amaçlı yazılım örnekleriyle karşılaştırılacak (1). Daha sonra, kullanıcıların (2) 'de geçirdiği örneğe benzer numuneler aramasına ve sonuçları terminale yazdırmasına izin veriyoruz. Ayrıca kullanıcının veritabanında bulunan numuneler hakkında yorum yapmasına da izin veriyoruz (3). Son olarak, kullanıcının mevcut veritabanını (4) silmesine izin veriyoruz.

->Running the Similarity Search System:  Benzerlik Arama Sistemini Çalıştırmak

Bu kodu uyguladıktan sonra, dört basit işlemden oluşan benzerlik arama sistemini çalıştırabilirsiniz:

Yükle Load : Örnekleri sisteme yüklemek, bunları gelecekteki kod paylaşım aramaları için sistem veritabanında depolar. Örnekleri tek tek yükleyebilir veya sistemin PE dosyaları için özyinelemeli olarak arayacağı ve bunları veritabanına yükleyeceği bir dizin belirtebilirsiniz. Bu bölümün kod dizininde çalıştırılan aşağıdaki komutu kullanarak örnekleri veritabanına yükleyebilirsiniz:

python listings\_5\_7\_to\_5\_12.py –l <path to directory or individual malware

sample>

Comment: Yorum Bir örnek hakkında yorum yapmak yararlıdır çünkü o örnek hakkında bilgi depolamanıza izin verir. Ayrıca, bu örneğe benzer yeni örnekler gördüğünüzde, bu örnekler üzerinde bir benzerlik araştırması daha eski, benzer örnek üzerinde yaptığınız yorumları ortaya çıkaracak ve böylece iş akışınızı hızlandıracaktır. Aşağıdaki komutla bir kötü amaçlı yazılım örneği hakkında yorum yapabilirsiniz:

python listings\_5\_7\_to\_5\_12.py –c <path to malware sample>

Search :  Arama Tek bir kötü amaçlı yazılım örneği verildiğinde, arama, veritabanındaki tüm benzer örnekleri tanımlar ve bunları azalan benzerlik sırasına göre yazdırır. Ayrıca, bu örnekler hakkında yapmış olabileceğiniz yorumlar da yazdırılır. Aşağıdaki komutu kullanarak belirli bir örneğe benzer kötü amaçlı yazılım örneklerini arayabilirsiniz:

python listings\_5\_7\_to\_5\_12.py –s <path to malware sample>

Wipe:  Silme Veritabanını silmek, aşağıdaki komutla yapabileceğiniz sistem veritabanındaki tüm kayıtları temizler:

python listings\_5\_7\_to\_5\_12.py –w

Liste 5-13, APT1 örneklerini sisteme yüklediğimizde nasıl göründüğünü gösterir.

Liste 5-13: Bu bölümde uygulanan benzerlik arama sistemine yükleme verilerinden örnek çıktı

Ve Liste 5-14, benzerlik araması yaptığımızda nasıl göründüğünü gösterir.

Liste 5-14: Bu bölümde uygulanan benzerlik arama sisteminden örnek çıktı

Sistemimizin, sorgu örneğinin (bir "greencat" örneği) kodu diğer greencat örnekleriyle paylaştığını doğru şekilde belirlediğini unutmayın. Bu numunelerin greencat ailesinin üyeleri olduğunu bilme lüksüne sahip olmasaydık, sistemimiz bizi bir ton tersine mühendislik işinden kurtarırdı.

Bu benzerlik arama sistemi, bir üretim benzerliği arama sisteminde nelerin uygulanacağına dair sadece küçük bir örnektir. Ancak, sisteme görselleştirme yetenekleri eklemek ve birden çok benzerlik arama yöntemini desteklemek için genişletmek için şimdiye kadar öğrendiklerinizi kullanmakta sorun yaşamazsınız.

ÖZET:

Bu bölümde, kötü amaçlı yazılım örnekleri arasındaki paylaşılan kod ilişkilerini nasıl belirleyeceğinizi, yeni kötü amaçlı yazılım ailelerini belirlemek için binlerce kötü amaçlı yazılım örneğinde kod paylaşım benzerliğini nasıl hesaplayacağınızı, yeni bir kötü amaçlı yazılım örneğinin daha önce görülen binlerce kötü amaçlı yazılım örneğiyle kod benzerliğini nasıl belirleyeceğinizi ve kötü amaçlı yazılım ilişkilerini nasıl görselleştireceğinizi öğrendiniz. kod paylaşımı modellerini anlamak için.

Artık kötü amaçlı yazılım analizi araç kutunuza paylaşılan kod analizi ekleyerek kendinizi rahat hissetmelisiniz; bu, büyük miktarda kötü amaçlı yazılım üzerinde hızlı istihbarat elde etmenizi ve kötü amaçlı yazılım analizi iş akışınızı hızlandırmanızı sağlar.

Bölüm 6, 7 ve 8'de kötü amaçlı yazılımları tespit etmek için makine öğrenimi sistemleri oluşturmayı öğreneceksiniz. Bu tespit tekniklerini öğrendiklerinizle birleştirmek, diğer araçların gözden kaçırdığı gelişmiş kötü amaçlı yazılımları yakalamanıza ve kötü amaçlı yazılımı kimin dağıttığı ve hedeflerinin ne olduğu hakkında ipuçları elde etmek için diğer bilinen kötü amaçlı yazılımlarla ilişkilerini analiz etmenize yardımcı olacaktır.

i