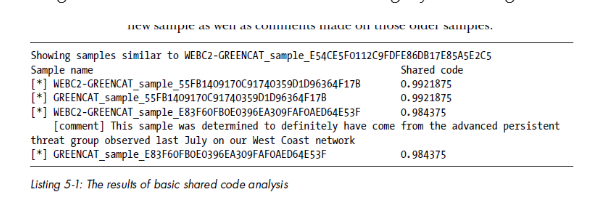
**5. Shared Code Analysis: Paylaşılan Kod Analizi**

Ağınızda yeni bir kötü amaçlı yazılım örneği keşfettiğinizi varsayalım. Onu analiz etmeye nasıl başlayacaksınız? Hangi kötü amaçlı yazılım ailesine ait olduğunu öğrenmek için bunu VirusTotal gibi çok motorlu bir antivirüs tarayıcısına gönderebilirsiniz. Ancak, motorlar kötü amaçlı yazılımı genellikle hiçbir şey ifade etmeyen "aracı agent " gibi genel terimlerle etiketlediğinden, bu tür sonuçlar genellikle belirsiz ve belirsizdir.Kötü amaçlı yazılım örneğinin geri arama sunucuları ve davranışları hakkında sınırlı bir rapor almak için örneği CuckooBox veya başka bir kötü amaçlı yazılım korumalı alanı aracılığıyla da çalıştırabilirsiniz. Bu yaklaşımlar yeterli bilgi sağlamadığında, örneğe tersine mühendislik uygulamanız gerekebilir. Bu aşamada, paylaşılan kod analizi iş akışınızı önemli ölçüde iyileştirebilir. Yeni kötü amaçlı yazılım örneğinin daha önce analiz edilen örnekleri ortaya çıkararak ve böylece paylaştıkları kodu ortaya çıkararak, paylaşılan kod analizi, yeni kötü amaçlı yazılım üzerinde önceki analizlerinizi yeniden kullanmanıza olanak tanır, böylece sıfırdan başlamazsınız. Daha önce görülen bu kötü amaçlı yazılımın nereden geldiğini anlamak, kötü amaçlı yazılımı kimin dağıtmış olabileceğini anlamanıza da yardımcı olabilir.

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. Tersine mühendislikte, paylaşılan kod analizi, birlikte analiz edilebilecek örnekleri belirlemeye yardımcı olur (çünkü bunlar aynı kötü amaçlı yazılım araç setinden üretilmiştir veya aynı kötü amaçlı yazılım ailesinin farklı sürümleri olabilir), bu da aynı geliştiricilerin bir gruptan sorumlu olup olmadığını belirleyebilir. kötü amaçlı yazılım örnekleri. Kötü amaçlı yazılım paylaşılan kod analizinin değerini göstermek için bu bölümde daha sonra oluşturacağınız bir programdan gelen Liste 5-1'de gösterilen çıktıyı düşünün. Yeni örnekle kodu paylaşabilecek daha önce görülen örnekleri ve bu eski örneklerle ilgili yorumları gösterir.



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. Örneğin, Şekil 5-1, gelişmiş bir kalıcı tehdit veri kümesindeki örnekler arasındaki paylaşılan kod ilişkileri ağını gösterir.

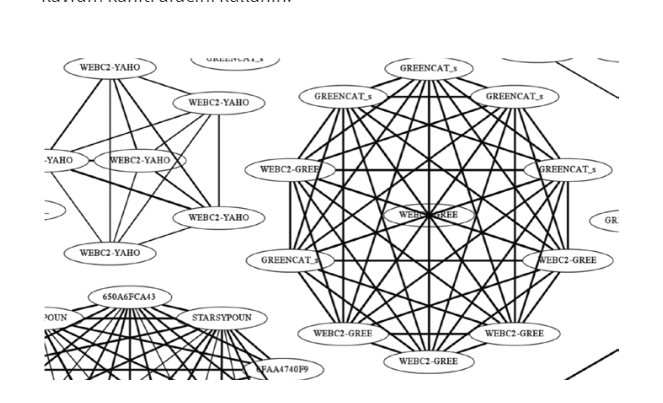
Görselleştirmeden de görebileceğiniz gibi, otomatik paylaşılan kod analizi teknikleri, manuel analiz yoluyla keşfedilmesi günler veya haftalar süren kötü amaçlı yazılım ailelerinin varlığını hızla ortaya çıkarabilir. Bu bölümde, aşağıdakileri yapmak için bu teknikleri kullanmayı öğreneceksiniz:

• 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.



Şekil 5-1: Bu bölümde oluşturmayı öğreneceğiniz görselleştirme türünün bir örneği, bazı APT1 örnekleri arasındaki paylaşılan kod ilişkilerini gösterir.

İlk olarak, Bölüm 4'teki PLA APT1 örnekleri ve çeşitli suç yazılımı örnekleri olan bu bölümde kullanacağınız test kötü amaçlı yazılım örneklerini tanıtıyorum. Ardından, matematiksel benzerlik karşılaştırması ve ortak özellikleri açısından kötü amaçlı yazılım örneklerini karşılaştırmak için bir dizi teorik yöntem olan Jaccard indeksi kavramını öğrenirsiniz. Ardından, iki kötü amaçlı yazılım örneğinin paylaştığı kod miktarını yaklaşık olarak tahmin etmek için bunları Jaccard indeksi ile birlikte nasıl kullanabileceğinizi gösteren özellikler kavramını tanıtacağım. Ayrıca, kötü amaçlı yazılım özelliklerini yararlılıkları açısından nasıl değerlendireceğinizi de öğrenirsiniz. Son olarak, Bölüm 4'teki ağ görselleştirme bilginizden yararlanarak, Şekil 5-1'de gösterildiği gibi, çeşitli ölçeklerde kötü amaçlı yazılım kod paylaşımının görselleştirmelerini oluşturuyoruz.

-> Bu Bölümde Kullanılan Kötü Amaçlı Yazılım Örnekleri:

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. İlk örnekler, paylaşılan kaynak analizini göstermek için Bölüm 4'te kullandığımız APT1 veri kümesinden gelir. Diğer örnekler, suçlular tarafından insanların kredi kartlarını çalmak, bilgisayarlarını botnet'lere bağlanmış zombi ana bilgisayarlara dönüştürmek için geliştirilen binlerce kötü amaçlı yazılım örneğinden oluşur. Bunlar, tehdit istihbaratı araştırmacıları için ücretli bir hizmet olarak sağlanan ticari bir kötü amaçlı yazılım beslemesinden elde edilen gerçek dünya örnekleridir. Aile adlarını belirlemek için her örneği Kaspersky antivirüs motoruna girdim. Kaspersky, bu örneklerden 30.104'ünü sağlam hiyerarşik sınıflandırmalarla (jorik.win32.jorik.skor.akr gibi) sınıflandırabildi, 41.830 örneğe "bilinmeyen" bir sınıf atadı ve jenerik etiketler atadı ( Örneğin, genel olarak "win32 Truva Atı") kalan 28.481 örneğe. Kaspersky etiketlerinin tutarsızlığı nedeniyle (jorik ailesi gibi bazı Kaspersky etiket gruplamaları çok yaygın bir kötü amaçlı yazılım yelpazesini temsil ederken, webprefix gibi diğerleri çok özel bir varyantlar kümesini temsil eder) ve Kaspersky'nin genellikle gözden kaçırdığı gerçeği nedeniyle veya kötü amaçlı yazılımları yanlış etiketlediğinde, Kaspersky'nin yüksek güvenle algıladığı yedi kötü amaçlı yazılım sınıfı seçtim. Bunlar özellikle dapato, pasta, skor, vbna, webprefix, xtoober ve zango ailelerini içerir.

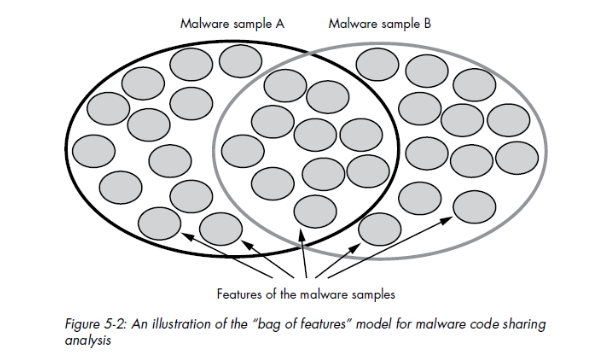
->Preparing Samples for Comparison by Extracting Features: Özellikleri Ayıklayarak Karşılaştırma için Örnekleri Hazırlama

Saldırganlar tarafından derlenmeden önce iki kötü amaçlı ikili dosyanın paylaşmış olabileceği kod miktarını tahmin etmeyi nasıl düşünebiliriz? Bu soruna yaklaşmanın birçok yolu vardır, ancak konuyla ilgili yayınlanan yüzlerce bilgisayar bilimi araştırma makalesinde ortak bir tema ortaya çıktı: ikili dosyalar arasında paylaşılan kod miktarını tahmin etmek için kötü amaçlı yazılım örneklerini " karşılaştırmadan önce bir sürü özellik ”. Özellikler derken, örnekler arasındaki kod benzerliğini tahmin ederken göz önünde bulundurmak isteyebileceğimiz herhangi bir kötü amaçlı yazılım özelliğini kastediyorum. Örneğin, kullandığımız özellikler, ikili dosyalardan çıkarabileceğimiz yazdırılabilir dizeler olabilir. Örnekleri birbirine bağlı işlevler sistemi, dinamik kitaplık içe aktarmaları vb. Olarak düşünmek yerine, kötü amaçlı yazılımı matematiksel kolaylık için bir dizi bağımsız özellik (örneğin, kötü amaçlı yazılımdan çıkarılan bir dizi dizi) olarak düşünürüz. .

->How Bag of Features Models Work:Çanta Özellikleri Modelleri Nasıl Çalışır?

Bir paket özelliğin nasıl çalıştığını anlamak için, Şekil 5-2'de gösterildiği gibi iki kötü amaçlı yazılım örneği arasındaki bir Venn diyagramını düşünün. Burada, örnek A ve örnek B, özellik paketleri olarak gösterilir (özellikler, Venn diyagramında elipsler olarak temsil edilir). İki örnek arasında hangi özelliklerin paylaşıldığını inceleyerek bunları karşılaştırabiliriz. İki özellik grubu arasındaki örtüşmeyi hesaplamak hızlıdır ve bulduğumuz gelişigüzel özelliklere dayanarak kötü amaçlı yazılım örneklerinin benzerliğini karşılaştırmak için kullanılabilir.

Örneğin, paketlenmiş kötü amaçlı yazılımlarla uğraşırken, kötü amaçlı yazılımın dinamik çalıştırma günlüklerine dayalı özellikleri kullanmak isteyebiliriz, çünkü bir sanal alanda kötü amaçlı yazılım çalıştırmak, kötü amaçlı yazılımın kendi paketini açmasını sağlamanın bir yoludur. Diğer durumlarda, karşılaştırmayı gerçekleştirmek için statik kötü amaçlı yazılım ikili dosyasından çıkarılan dizeleri kullanabiliriz.



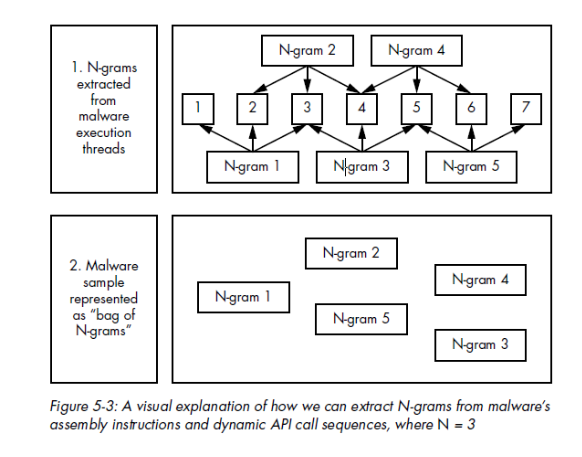
Şekil 5-2: Kötü amaçlı yazılım kod paylaşım analizi için "özellik çantası" modelinin bir resmi

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. Başka bir deyişle, Şekil 5-3'te gösterildiği gibi, bir dizi üzerinde yineleyerek ve her adımda, indeks i'deki olaydan indeks i + N - 1'deki olaya kadar olan alt sekansı kaydederek N-gram elde ederiz.

Şekil 5-3'te, 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). Elbette bunu herhangi bir sıralı veri ile yapabiliriz. Örneğin, 2 N gramlık bir kelime uzunluğu kullanıldığında, "how now brown cow" cümlesi şu alt dizileri verir: "how now”, “now brown”, and “brown cow.”Kötü amaçlı yazılım analizinde, bir kötü amaçlı yazılım örneğinin yaptığı N-gram sıralı API çağrılarını çıkarırdık. Daha sonra kötü amaçlı yazılımı bir dizi özellik olarak temsil eder ve kötü amaçlı yazılım örneğini başka bir kötü amaçlı yazılım örneğinin N-gramları ile karşılaştırmak için N-gram özelliklerini kullanır, böylece sıra bilgilerini özellik karşılaştırma modeline dahil ederiz.



Şekil 5-3: N-gramları kötü amaçlı yazılımın montaj talimatlarından ve dinamik API çağrısı dizilerinden nasıl çıkarabileceğimize dair görsel bir açıklama; burada N = 3

Kötü amaçlı yazılım örnekleri karşılaştırmamıza sıra bilgilerini dahil etmenin avantajları ve dezavantajları vardır. Avantaj, karşılaştırmada sipariş önemli olduğunda (örneğin, API çağrısının C API çağrısından önce gözlemlenen API çağrısı B'den önce gözlemlendiğini önemsediğimizde), siparişi yakalamamıza izin verir, ancak sipariş gereksiz olduğunda ( örneğin, her çalıştırmada A, B ve C o API çağrılarının sırasını rastgele hale getiren kötü amaçlı yazılım, paylaşılan kod tahminimizi çok daha kötü hale getirebilir. Kötü amaçlı yazılım paylaşılan kod tahmini çalışmamıza sipariş bilgilerinin dahil edilip edilmeyeceğine karar vermek, üzerinde çalıştığımız kötü amaçlı yazılımın türüne bağlıdır ve denememizi gerektirir.

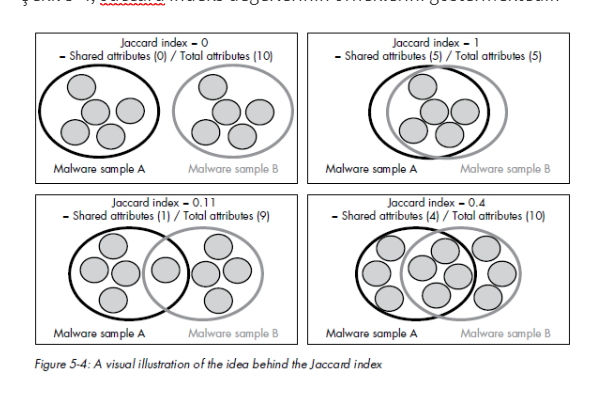
->Using the Jaccard Index to Quantify Similarity:Benzerliği Ölçmek için Jaccard Endeksini Kullanma

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. evlat edin - ve iyi bir nedenle. 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.

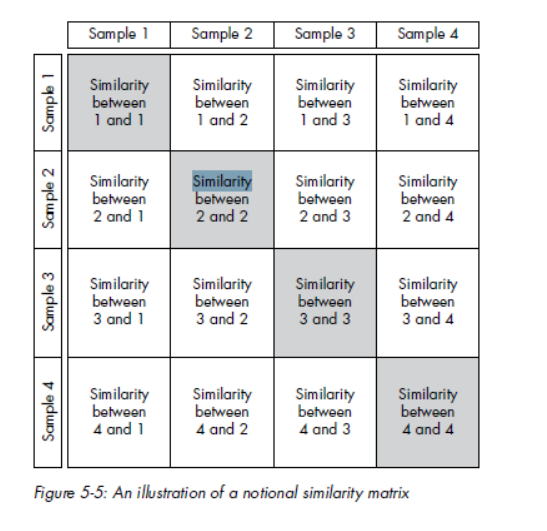
Şekil 5-4, Jaccard indeks değerlerinin örneklerini göstermektedir.



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.

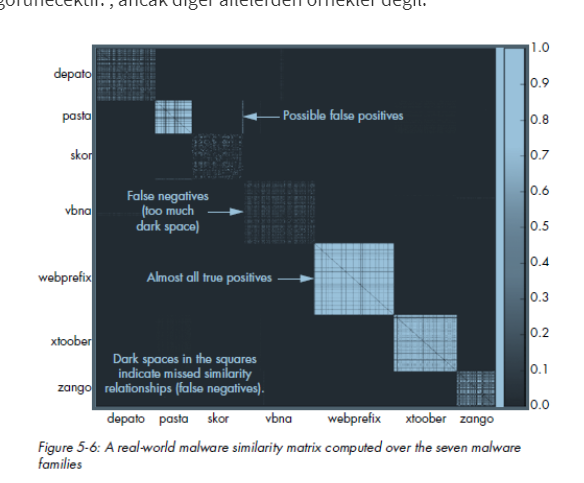
->Using Similarity Matrices to Evaluate Malware Shared Code Estimation Methods: Kötü Amaçlı Yazılım Paylaşılan Kod Tahmin Yöntemlerini Değerlendirmek için Benzerlik Matrislerini Kullanma

İki kötü amaçlı yazılım örneğinin aynı aileden gelip gelmediğini belirlemek için dört yöntemi tartışalım: 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. Şekil 5-5, bir benzerlik matrisi kullanarak hayali bir dizi kötü amaçlı yazılım örneğini karşılaştırır.



Bu matris, tüm örnekler arasındaki benzerlik ilişkisini görmenizi sağlar. Bu matriste bir miktar alanın boşa gittiğini görebilirsiniz. Örneğin, gölgeli kutularda gösterilen benzerlikler umurumuzda değil, çünkü bu girişler yalnızca belirli bir örnek ile kendisi arasındaki karşılaştırmaları içeriyor. Ayrıca gölgeli kutuların her iki tarafındaki bilgilerin tekrarlandığını da görebilirsiniz, bu nedenle yalnızca birine veya diğerine bakmanız gerekir.

Şekil 5-6, kötü amaçlı yazılım benzerlik matrisinin gerçek hayattan bir örneğini vermektedir. 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.



Şekil 5-6: Yedi kötü amaçlı yazılım ailesi üzerinden hesaplanan gerçek dünya kötü amaçlı yazılım benzerlik matrisi

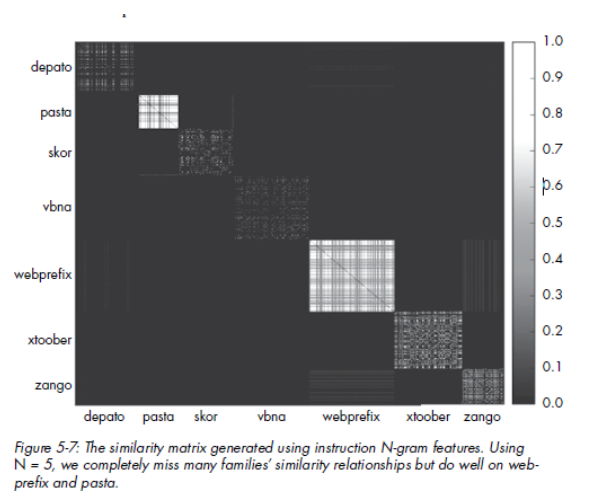
Şekil 5-6'da verilen sonuçlarda, bazı aile karelerinin tamamen beyaz olduğunu görebilirsiniz - bunlar iyi sonuçlardır, çünkü bir aile karesi içindeki beyaz pikseller, aynı aileden örnekler arasında çıkarsanan benzerlik ilişkisini gösterir. Bazıları çok daha koyu, yani güçlü benzerlik ilişkileri tespit etmedik. Son olarak, bazen aile karelerinin dışında, ilgili kötü amaçlı yazılım ailelerinin kanıtı veya yanlış pozitiflerin kanıtı olan piksel satırları olabilir; bu, doğası gereği farklı olmalarına rağmen aileler arasında kod paylaşımını tespit ettiğimiz anlamına gelir.

Daha sonra, talimat dizisi tabanlı benzerlik analizinin bir açıklamasından başlayarak, dört farklı kod paylaşma tahmin yönteminin sonuçlarını karşılaştırmak için Şekil 5-6 gibi benzerlik matrisi görselleştirmelerini kullanacağız.

->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. Bu, örneğin Bölüm 2'de tanıtılan doğrusal sökme tekniğini kullanarak kötü amaçlı yazılım örneklerini sökmeyi gerektirir. 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-gram ekstraksiyonu sırasında N için kullandığımız değer, analiz hedeflerimize bağlıdır. N ne kadar büyükse, çıkarılan talimat alt dizilerimiz o kadar büyük olur ve bu nedenle kötü amaçlı yazılım örneklerinin dizilerinin eşleşmesi o kadar zor olur. N'nin büyük bir sayıya ayarlanması, yalnızca kodu birbiriyle paylaşma olasılığı yüksek olan örneklerin belirlenmesine yardımcı olur. Öte yandan, numuneler arasında ince benzerlikler aramak için N'yi küçültebilirsiniz veya numunelerin benzerlik analizini belirsizleştirmek için yeniden sıralama talimatını kullandığından şüpheleniyorsanız.

Şekil 5-7'de N, numunelerin eşleşmesini zorlaştıran agresif bir ayar olan 5'e ayarlanmıştır.



Şekil 5-7: Komut N-gram özellikleri kullanılarak oluşturulan benzerlik matrisi. N = 5 kullanarak, birçok ailenin benzerlik ilişkilerini tamamen kaçırıyoruz, ancak web öneki ve "pasta "başarılı oluyoruz.

Şekil 5-7'deki sonuçlar çok zorlayıcı değildir. Öğretim temelli benzerlik analizi bazı aileler arasındaki benzerlikleri doğru bir şekilde tespit ederken, diğer aileler içinde tespit etmemektedir (örneğin, dapato, skor ve vbna'da birkaç benzerlik ilişkisi tespit etmektedir). Bununla birlikte, bu analizde çok az yanlış pozitif (farklı ailelerden alınan örnekler arasındaki benzerlik yanlış çıkarımlarına karşı aynı aileden örneklerdeki benzerliklerin gerçek çıkarımları) olduğuna dikkat etmek önemlidir. Gördüğünüz gibi, komut alt dizisi paylaşılan kod analizinin bir sınırlaması, örnekler arasındaki birçok kod paylaşım ilişkisini gözden kaçırabilmesidir.

Bunun nedeni, kötü amaçlı yazılım örneklerinin, talimatlarının çoğu yalnızca kötü amaçlı yazılım örneklerini çalıştırıp paketlerini açmalarına izin verdiğimizde görünür olacak şekilde paketlenmiş olabilmesidir. Kötü amaçlı yazılım örneklerimizi paketinden çıkarmadan, komut dizisi paylaşılan kod tahmin yöntemi muhtemelen çok iyi çalışmayacaktır. Kötü amaçlı yazılım örneklerimizi paketinden çıkardığımızda bile, kaynak kodu derleme işleminin getirdiği gürültü nedeniyle bu yaklaşım sorunlu olabilir. Aslında, derleyiciler aynı kaynak kodunu radikal olarak farklı montaj talimatları dizileri halinde derleyebilirler. Örneğin, C ile yazılmış aşağıdaki basit işlevi alın:

int f(void) {

int a = 1;

int b = 2;

(1) return (a\*b)+3;

}

Derleyiciden bağımsız olarak, işlevin aynı montaj talimatları sırasına indirgeneceğini düşünebilirsiniz. Ama aslında, derleme sadece hangi derleyiciyi kullandığınıza değil, aynı zamanda derleyici ayarlarına da bağlıdır. Örneğin, bu işlevi varsayılan ayarları altında clang derleyicisini kullanarak derlemek, kaynak kodda (1) satırına karşılık gelen aşağıdaki talimatları verir:

movl     $1, -4(%rbp)

movl     $2, -8(%rbp)

movl     -4(%rbp), %eax

imull     -8(%rbp), %eax

addl     $3, %eax

Buna karşılık, aynı işlevi derleyiciye kodu hız için optimize etmesini söyleyen –O3 bayrak kümesiyle derlemek, aynı kaynak kod satırı için aşağıdaki derlemeyi verir:

movl     $5, %eax

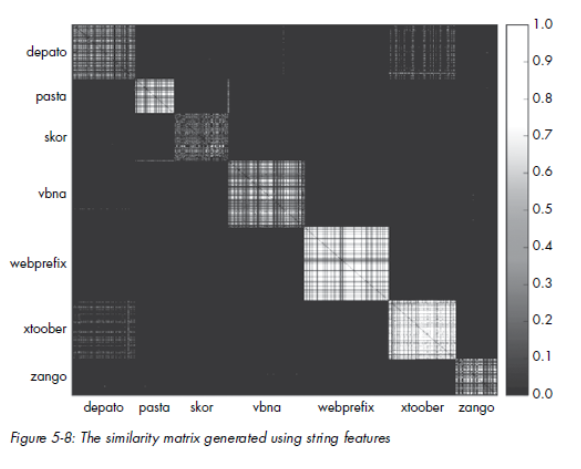
Fark, ikinci örnekte, derleyicinin, birinci derleme örneğinde olduğu gibi, açıkça hesaplamak yerine işlevin sonucunu önceden hesaplamış olmasından kaynaklanmaktadır. Bu, komut dizilerine göre bu işlevleri karşılaştırırsak, gerçekte tamamen aynı kaynak kodundan derlenmiş olsalar bile, hiç benzer görünmeyecekleri anlamına gelir. 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.

Bu yaklaşım derleyici probleminin üstesinden gelir çünkü bir ikili dosyadan çıkarılan dizeler, programcı tarafından tanımlanan biçim dizeleri olma eğilimindedir ve genel bir kural olarak derleyiciler, kötü amaçlı yazılım yazarlarının hangi derleyicileri kullandığına veya derleyicilere hangi parametreleri verdiklerine bakılmaksızın dönüştürmez. . Örneğin, bir kötü amaçlı yazılım ikili programından çıkarılan tipik bir dize, "Anahtar kaydedici% s tarihinde% s ve% s saatinde başlatıldı. Started key logger at %s on %s and time %s" şeklinde olabilir. Derleyici ayarlarından bağımsız olarak, bu dizge birden çok ikili arasında aynı görünme eğiliminde olacaktır ve aynı kaynak kodu tabanına dayalı olup olmadıklarıyla ilgilidir.

Şekil 5-8, dizi tabanlı kod paylaşım metriğinin, suç yazılımı veri setinde doğru kod paylaşım ilişkilerini ne kadar iyi tanımladığını göstermektedir.

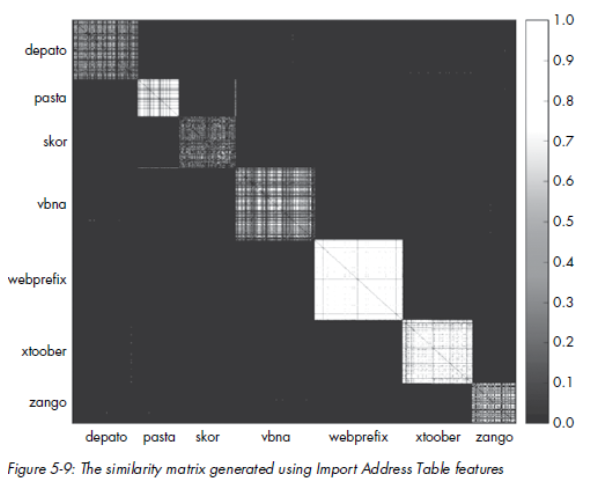


Şekil 5-8: Dize özellikleri kullanılarak oluşturulan benzerlik matrisi

İlk bakışta, bu yöntem kötü amaçlı yazılım ailelerini tanımlamada talimat tabanlı yönteme göre çok 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

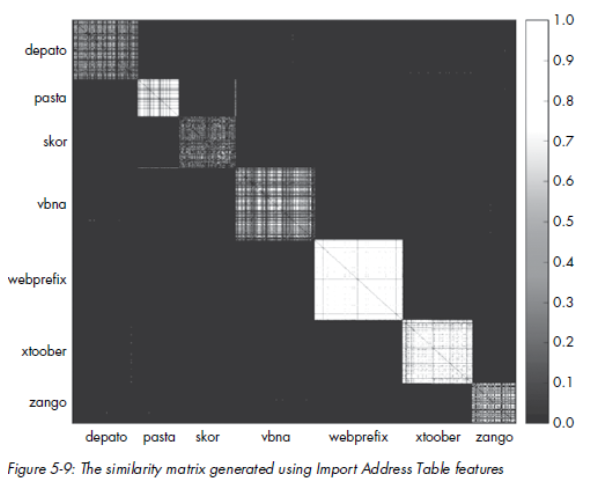
Kötü amaçlı yazılım ikili dosyaları tarafından yapılan DLL içe aktarmalarını karşılaştırarak, "Adres Tablosuna dayalı İçe Aktarma benzerliği" dediğim şeyi hesaplayabiliriz. Bu yaklaşımın arkasındaki fikir, kötü amaçlı yazılım yazarı talimatları yeniden sıralasa, kötü amaçlı yazılım ikili programının 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

Artık kötü amaçlı yazılım kod paylaşımını belirleme yöntemlerinin arkasındaki kavramları anladığınıza göre, bu analizi kötü amaçlı yazılım 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.

5-2 ila 5-6 arasındaki listeler örnek programımızı içerir. Liste uzun olduğu için parçalara ayırıyorum ve her parçayı ilerledikçe açıklıyorum. 5-2 listesi, kullanacağımız kitaplıkları içe aktarır ve iki örneğin özellik seti arasındaki Jaccard indeksini hesaplayan jaccard () işlevini bildirir.

#!/usr/bin/python

import argparse

import os

import networkx

from networkx.drawing.nx\_pydot import write\_dot

import itertools

def jaccard(set1, set2):

"""

Compute the Jaccard distance between two sets by taking

their intersection, union and then dividing the number

of elements in the intersection by the number of elements

in their union. İki küme arasındaki Jaccard mesafesini, kesişimlerini, birleşimlerini alarak ve ardından kesişimdeki eleman sayısını birleşimindeki elemanların sayısına bölerek hesaplayın.

"""

intersection = set1.intersection(set2)

intersection\_length = float(len(intersection))

union = set1.union(set2)

union\_length = float(len(union))

return intersection\_length / union\_length

Listing: 5-2:  Liste 5-2: İki örnek arasında Jaccard indeksini hesaplamak için içe aktarmalar ve yardımcı bir işlev

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.

def getstrings(fullpath):

"""

'Fullpath' parametresiyle gösterilen ikiliden dizeleri ayıklayın ve ardından ikili dosyadaki benzersiz dizeler kümesini döndürün.

"""

strings = os.popen("strings '{0}'".format(fullpath)).read()

strings = set(strings.split("\n"))

return strings

def pecheck(fullpath):

"""

Do a cursory sanity check to make sure 'fullpath' is

a Windows PE executable (PE executables start with the

two bytes 'MZ')  'Fullpath'in bir Windows PE yürütülebilir dosyası olduğundan emin olmak için üstünkörü bir mantık kontrolü yapın (PE yürütülebilir dosyaları iki bayt' MZ 'ile başlar)

"""

return open(fullpath).read(2) == "MZ"

Listing 5-3: Declaring the functions we'll use in feature extraction

Ardından, Liste 5-4'te, 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.

If \_\_name\_\_ == "\_\_main\_\_":

parser = argparse.ArgumentParser(

description="Identify similarities between malware samples and build similarity graph"

)

parser.add\_argument(

"target\_directory",

help="Directory containing malware"

)

parser.add\_argument(

"output\_dot\_file",

help="Where to save the output graph DOT file"

)

parser.add\_argument(

"--jaccard\_index\_threshold", "-j", dest="threshold", type=float,

default=0.8, help="Threshold above which to create an 'edge' between samples"

)

args = parser.parse\_args()

Liste 5-4: Kullanıcının komut satırı bağımsız değişkenlerini ayrıştırma

Ardından, Liste 5-5'te, 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.

malware\_paths = [] # where we'll store the malware file paths

malware\_features = dict() # where we'll store the malware strings

graph = networkx.Graph() # the similarity graph

for root, dirs, paths in os.walk(args.target\_directory):

# walk the target directory tree and store all of the file paths

for path in paths:

full\_path = os.path.join(root, path)

malware\_paths.append(full\_path)

# filter out any paths that aren't PE files

malware\_paths = filter(pecheck, malware\_paths)

# get and store the strings for all of the malware PE files

for path in malware\_paths:

features = getstrings(path)

print "Extracted {0} features from {1} ...".format(len(features), path)

malware\_features[path] = features

# add each malware file to the graph

graph.add\_node(path, label=os.path.split(path)[-1][:10])

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.

# iterate through all pairs of malware

for malware1, malware2 in itertools.combinations(malware\_paths, 2):

# compute the jaccard distance for the current pair

jaccard\_index = jaccard(malware\_features[malware1], malware\_features[malware2])

# if the jaccard distance is above the threshold, add an edge

if jaccard\_index > args.threshold:

print malware1, malware2, jaccard\_index

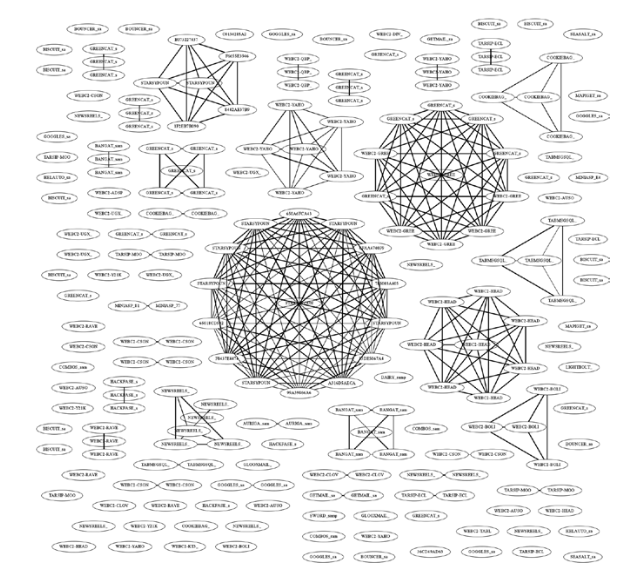
graph.add\_edge(malware1, malware2, penwidth=1+(jaccard\_index-args.threshold)\*10)

# write the graph to disk so we can visualize it

write\_dot(graph, args.output\_dot\_file)

Liste 5-6: Python'da bir kod paylaşım grafiği oluşturma

Liste 5-2 ila 5-6'daki kod, APT1 kötü amaçlı yazılım örneklerine uygulandığında Şekil 5 11'de gösterilen grafiği oluşturur. Grafiği görselleştirmek için, fdp -Tpng network.dot -o network.png komutunu girmek için fdp Graphviz aracını (Bölüm 4'te tartışılmıştır) kullanmanız gerekir.



Şekil 5-11: APT1 örnekleri için tam dizi tabanlı benzerlik grafiği The complete string-based similarity graph for the APT1 samples

Bu çıktıyla ilgili şaşırtıcı olan şey, APT1'in orijinal analistlerinin raporlarında ürettikleri manuel ve özenli çalışmanın çoğunu birkaç dakika içinde yeniden oluşturarak bu ulus devlet düzeyindeki saldırganlar tarafından kullanılan kötü amaçlı yazılım ailelerinin çoğunu tespit etmemizdir.

Yöntemimizin, bu analistlerin gerçekleştirdiği manüel tersine mühendislik çalışmasına göre doğru bir şekilde uygulandığını biliyoruz, çünkü düğümlerdeki isimler onlara Mandiant analistleri tarafından verilen adlardır. Bunu, merkez çemberdeki "STARSYPOUN" örnekleri gibi, Şekil 5-11'deki ağ görselleştirmesinde benzer adlara sahip örneklerin gruplanma biçiminde görebilirsiniz. Ağ görselleştirmemizdeki kötü amaçlı yazılım, bu aile adlarıyla aynı hizada olacak şekilde otomatik olarak gruplandığı için, yöntemimiz Mandiant kötü amaçlı yazılım analistleriyle "aynı fikirde" görünüyor. Kodları 5 2'den 5-6'ya kadar genişletebilir ve benzer zeka için kendi kötü amaçlı yazılımınıza uygulayabilirsiniz.

->Scaling Similarity Comparisons:Benzerlik Karşılaştırmalarını Ölçeklendirme

5-2 ila 5-6 Listelerindeki kod, 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. Spesifik olarak, aşağıdaki denklem, n büyüklüğünde bir veri kümesi üzerinden bir benzerlik matrisini hesaplamak için gerekli Jaccard indeksi hesaplamalarının sayısını verir:

n2-n / 2 formülü🙂

Örneğin, dört örneği hesaplamak için kaç Jaccard indeksine ihtiyacımız olacağını görmek için Şekil 5-5'teki benzerlik matrisine dönelim. İlk bakışta 16 (42) diyebilirsiniz, çünkü matriste bu kadar hücre vardır. Ancak, matrisin alt üçgeni, matrisin üst üçgeninin kopyalarını içerdiğinden, bunları iki kez hesaplamamız gerekmez. Bu, toplam hesaplama sayımızdan 6 çıkarabileceğimiz anlamına gelir. Dahası, kötü amaçlı yazılım örneklerini kendileriyle karşılaştırmamıza gerek yok, böylece matristeki köşegeni ortadan kaldırarak dört hesaplama daha çıkarmamıza olanak tanıyabiliriz. Gerekli hesaplama sayısı aşağıdaki gibidir:

4^2 -4 / 2 = 16 -4 /2 = 6

Veri kümemiz, örneğin 49.995.000 hesaplama gerektiren 10.000 kötü amaçlı yazılım örneğine ulaşana kadar bu, yönetilebilir görünüyor. 50.000 örneğe sahip bir veri kümesi, 1.249.975.000 Jaccard indeksi hesaplamaları gerektirir!

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 neden çalıştığını okumadan önce, bunun anlaşılması biraz zaman alabilecek karmaşık bir algoritma olduğuna dikkat edin. "Derinlikte Minhash" bölümünü atlamaya karar verirseniz, "Özetle Minhash" bölümünü okuyun ve sağlanan kodu kullanın ve kod paylaşım analizinizi ölçeklendirmede sorun yaşamazsınız.

->Minhash in Nutshell: Özetle Minhash

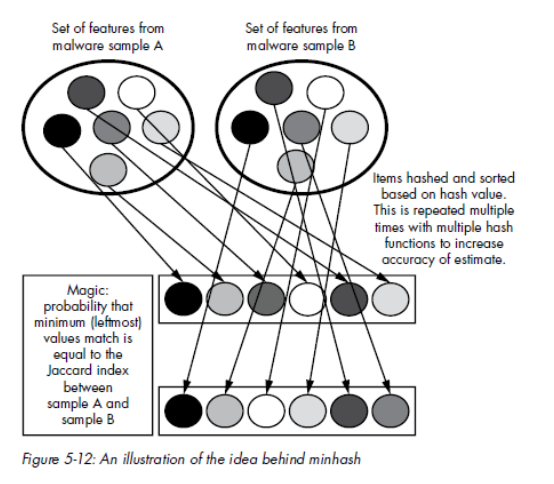
Minhash, bir kötü amaçlı yazılım örneğinin özelliklerini alır ve bunları k hash işlevleriyle hash hale getirir. Her bir karma işlevi için, tüm özellikler üzerinden hesaplanan karma değerlerinin yalnızca minimum değerini saklarız, böylece kötü amaçlı yazılım özellikleri kümesi, minhash olarak adlandırdığımız 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. .

->Minhash in Depth: Derinlikte Minhash:

Şimdi minhash'in arkasındaki matematiği derinlemesine tartışalım. Şekil 5-12, iki kötü amaçlı yazılım örneği için özellik kümelerini (gölgeli dairelerle temsil edilir), bunların nasıl karma hale getirildiğini ve ardından hash'lerine göre nasıl sıralandığını ve her listenin ilk öğesinin değerine göre nihayet nasıl karşılaştırıldıklarını gösterir. .



79.sayfa:

İlk elemanların eşleşme olasılığı, numuneler arasındaki Jaccard indeksine eşittir. Bunun nasıl çalıştığı bu kitabın kapsamı dışındadır, ancak bu tesadüfi gerçek, hash kullanarak Jaccard indeksine yaklaşmamızı sağlayan şeydir.

Elbette, bu hashing, sıralama ve ilk element kontrol işlemini sadece bir kez yaparsak bize pek bir şey söylemez - hash'ler eşleşir veya eşleşmez ve temeldeki Jaccard indeksini pek tahmin edemeyiz. bu tek eşleşmeye göre doğru şekilde. Bu temel değerin daha iyi bir tahminini elde etmek için, k hash fonksiyonlarını kullanmalı ve bu işlemi k kez tekrarlamalı ve ardından bu ilk elemanların kaç kez eşleştiğini k ile bölerek Jaccard indeksini tahmin etmeliyiz. Jaccard endeksini tahmin etmede beklenen hatamız şu şekilde tanımlanmaktadır:

1.0 / kök k

Bu nedenle, bu prosedürü ne kadar çok kez uygularsak, o kadar emin oluruz (k'yi 256'ya ayarlama eğilimindeyim, böylece tahmin ortalama yüzde 6 düşmüş olur).

Bir milyon örnek içeren bir kötü amaçlı yazılım veri kümesindeki her kötü amaçlı yazılım örneği için bir minhash dizisi hesapladığımızı varsayalım. Daha sonra, veri kümesindeki kötü amaçlı yazılım aileleri aramasını hızlandırmak için minhash'leri nasıl kullanacağız? Veri kümesindeki her kötü amaçlı yazılım örneğini yineleyebilir ve minhash dizilerini karşılaştırabiliriz, bu da 499.999.500.000 karşılaştırmaya yol açar. Minhash dizilerini karşılaştırmak Jaccard indeksini hesaplamaktan daha hızlı olsa da, bu hala modern donanımlarda yapılamayacak kadar çok karşılaştırma. Karşılaştırma sürecini daha da optimize etmek için mini hasılatlardan yararlanmanın bir yoluna ihtiyacımız var.

Bu soruna standart yaklaşım, eskiz oluşturmayı veritabanı indeksleme ile birlikte kullanmaktır; bu, yalnızca zaten benzer olduğunu bildiğimiz örnekleri karşılaştırdığımız bir sistem oluşturur. Birden çok minhash'ı birlikte hashing yaparak bir çizim yapıyoruz.

Yeni bir numune aldığımızda, veritabanının yeni numunenin çizimleriyle eşleşen herhangi bir eskiz içerip içermediğini kontrol ederiz. Eğer öyleyse, yeni örnek ile daha eski, benzer örnekler arasındaki Jaccard indeksine yaklaşmak için yeni örnek minhash dizileri kullanılarak eşleşen örneklerle karşılaştırılır. Bu, yeni numuneyi veri tabanındaki tüm numunelerle karşılaştırmak zorunda kalmaz ve bunun yerine sadece bu yeni numuneyle yüksek Jaccard indekslerine sahip olma olasılığı yüksek olan numunelerle karşılaştırır. numune = samples örnek

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

Artık kötü amaçlı yazılım örnekleri arasındaki paylaşılan kod ilişkilerini tahmin etmek için çeşitli kötü amaçlı yazılım özelliği türlerini kullanmanın artılarını ve eksilerini öğrendiğinize göre. Ayrıca, Jaccard indeksi, benzerlik matrisleri ve minhash'in çok büyük veri kümelerinde bile izlenebilir kötü amaçlı yazılım örnekleri arasında hesaplama benzerliklerini nasıl yapabildiğini öğrendiniz. Elinizdeki tüm bu bilgilerle, ölçeklenebilir bir kötü amaçlı yazılım paylaşımlı kod arama sistemi oluşturmak için gerekli tüm temel kavramları anlarsınız.

listeler: listing

5-7 ila 5-12 arasındaki listeler, kötü amaçlı yazılım örneklerini dize özelliklerine göre indekslediğim basit bir sistem örneğini gösterir. Kendi çalışmanızda, bu sistemi diğer kötü amaçlı yazılım özelliklerini kullanacak şekilde değiştirme veya daha fazla görselleştirme özelliğini destekleyecek şekilde genişletme konusunda kendinizi güvende hissetmelisiniz. Giriş uzun olduğu için, onu ayırdım ve sırayla her bir alt bölümü ele alacağız.

Başlangıç olarak, Liste 5-7 programımız için gerekli olan Python paketlerini içe aktarır.

#!/usr/bin/python

import argparse

import os

import murmur

import shelve

import numpy as np

from listings\_5\_2\_to\_5\_6 import \*

NUM\_MINHASHES = 256

SKETCH\_RATIO = 8

Liste 5-7: Python modüllerini içe aktarma ve minhash ile ilgili sabitleri bildirme

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. Listings\_5\_2\_to\_5\_6.py örnek benzerliği hesaplamak için işlevler elde etmek 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.

(1)def wipe\_database():

"""

This problem uses the python standard library 'shelve' database to persist

information, storing the database in the file 'samples.db' in the same

directory as the actual Python script. 'wipe\_database' deletes this file

effectively reseting the system. Bu sorun, bilgileri kalıcı hale getirmek için python standart kitaplığı 'raf' veritabanını kullanır ve veritabanını gerçek Python betiğiyle aynı dizindeki 'samples.db' dosyasında depolar. 'wipe\_database' bu dosyayı siler ve sistemi etkin bir şekilde sıfırlar.

"""

dbpath = "/".join(\_\_file\_\_.split('/')[:-1] + ['samples.db'])

os.system("rm -f {0}".format(dbpath))

(2) def get\_database():

"""

Helper function to retrieve the 'shelve' database, which is a simple

key value store. Basit bir anahtar değer deposu olan 'raf' veritabanını almak için yardımcı işlev.

"""

dbpath = "/".join(\_\_file\_\_.split('/')[:-1] + ['samples.db'])

return shelve.open(dbpath,protocol=2,writeback=True)

Listing 5-8: Database helper functions:

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.

def minhash(features):

"""

This is where the minhash magic happens, computing both the minhashes of

a sample's features and the sketches of those minhashes. The number of

minhashes and sketches computed is controlled by the NUM\_MINHASHES and

NUM\_SKETCHES global variables declared at the top of the script.

Minhash büyüsünün gerçekleştiği yer burasıdır, hem bir örneğin özelliklerinin minhash'lerini hem de bu minhash'ların çizimlerini hesaplar. Hesaplanan kısa çizgi ve çizimlerin sayısı, komut dosyasının en üstünde belirtilen NUM\_MINHASHES ve NUM\_SKETCHES genel değişkenler tarafından kontrol edilir.

"""

minhashes = []

sketches = []

(1) for i in range(NUM\_MINHASHES):

minhashes.append(

(2) min([murmur.string\_hash(`feature`,i) for feature in features])

)

(3)for i in xrange(0,NUM\_MINHASHES,SKETCH\_RATIO):

(4)sketch = murmur.string\_hash(`minhashes[i:i+SKETCH\_RATIO]`)

     sketches.append(sketch)

return np.array(minhashes),sketches

Liste 5-9: Bir örnek için mini kareler minhashes  ve eskizler sketches edinme

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.

def store\_sample(path):

"""

Function that stores a sample and its minhashes and sketches in the

'shelve' database

"""

(1) db = get\_database()

(2) features = getstrings(path)

(3) minhashes,sketches = minhash(features)

(4) for sketch in sketches:

sketch = str(sketch)

      (5)if not sketch in db:

db[sketch] = set([path])

else:

obj = db[sketch]

(6) obj.add(path)

db[sketch] = obj

db[path] = {'minhashes':minhashes,'comments':[]}

db.sync()

print "Extracted {0} features from {1} ...".format(len(features),path)

Liste 5-10: Bir örneğin mini karmalarını, çizimlerini sketches anahtar olarak kullanarak raf shelve veritabanında saklama

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 ().

(1) def comment\_sample(path):

"""

Function that allows a user to comment on a sample. The comment the

user provides shows up whenever this sample is seen in a list of similar

samples to some new samples, allowing the user to reuse their

knowledge about their malware database. Kullanıcının bir numune hakkında yorum yapmasına olanak tanıyan işlev. Kullanıcının sağladığı yorum, bu örnek bazı yeni örneklere benzer örnekler listesinde her görüldüğünde ortaya çıkar ve kullanıcının kötü amaçlı yazılım veritabanı hakkındaki bilgilerini yeniden kullanmasına olanak tanır.

"""

db = get\_database()

comment = raw\_input("Enter your comment:")

if not path in db:

store\_sample(path)

comments = db[path]['comments']

comments.append(comment)

db[path]['comments'] = comments

db.sync()

print "Stored comment:", comment

(2) def search\_sample(path):

"""

Function searches for samples similar to the sample provided by the

'path' argument, listing their comments, filenames, and similarity values

İşlev, 'yol' bağımsız değişkeni tarafından sağlanan örneğe benzer örnekleri arar, yorumlarını, dosya adlarını ve benzerlik değerlerini listeler.

"""

db = get\_database()

features = getstrings(path)

minhashes, sketches = minhash(features)

neighbors = []

(3) for sketch in sketches:

sketch = str(sketch)

if not sketch in db:

continue

(4) for neighbor\_path in db[sketch]:

neighbor\_minhashes = db[neighbor\_path]['minhashes']

similarity = (neighbor\_minhashes == minhashes).sum()

/ float(NUM\_MINHASHES)

neighbors.append((neighbor\_path, similarity))

neighbors = list(set(neighbors))

(5) neighbors.sort(key=lambda entry:entry[1], reverse=True)

print ""

print "Sample name".ljust(64), "Shared code estimate"

for neighbor, similarity in neighbors:

short\_neighbor = neighbor.split("/")[-1]

comments = db[neighbor]['comments']

print str("[\*] "+short\_neighbor).ljust(64), similarity

for comment in comments:

print "\t[comment]",comment

Liste 5-11: Kullanıcıların örnekler üzerinde yorum yapmasına ve bir sorgu örneğine benzer örnekleri aramasına olanak tanıyan bildirme işlevleri

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).

Liste 5-12, programımızın bağımsız değişken çözümleme bölümünü uygulayarak programımızın kodunu sonlandırır.

if \_\_name\_\_ == '\_\_main\_\_':

parser = argparse.ArgumentParser(

description="""

Simple code-sharing search system which allows you to build up

a database of malware samples (indexed by file paths) and

then search for similar samples given some new sample Kötü amaçlı yazılım örneklerinden oluşan bir veritabanı oluşturmanıza (dosya yollarına göre indekslenmiş) ve ardından yeni bir örnek verildiğinde benzer örnekleri aramanıza olanak tanıyan basit kod paylaşım arama sistemi

"""

)

parser.add\_argument(

"-l", "--load", dest="load", default=None,

help="Path to malware directory or file to store in database Veritabanında saklanacak kötü amaçlı yazılım dizini veya dosyanın yolu"

)

parser.add\_argument(

"-s", "--search", dest="search", default=None,

help="Individual malware file to perform similarity search on Benzerlik araması yapılacak bağımsız kötü amaçlı yazılım dosyası"

)

parser.add\_argument(

"-c", "--comment", dest="comment", default=None,

help="Comment on a malware sample path Kötü amaçlı yazılım örnek yolu hakkında yorum yapın"

)

parser.add\_argument(

"-w", "--wipe", action="store\_true", default=False,

help="Wipe sample database Örnek veritabanını silin"

)

args = parser.parse\_args()

(1) if args.load:

malware\_paths = [] #kötü amaçlı yazılım dosyası yollarını nerede saklayacağız

malware\_features = dict() # kötü amaçlı yazılım dizelerini nerede saklayacağız

for root, dirs, paths in os.walk(args.load):

# hedef dizin ağacında yürüyün ve tüm dosya yollarını saklayın

for path in paths:

full\_path = os.path.join(root,path)

malware\_paths.append(full\_path)

# PE dosyası olmayan tüm yolları filtreleyin

malware\_paths = filter(pecheck, malware\_paths)

# tüm kötü amaçlı yazılım PE dosyalarının dizelerini alın ve saklayın

for path in malware\_paths:

store\_sample(path)

(2) if args.search:

search\_sample(args.search)

(3)if args.comment:

comment\_sample(args.comment)

(4) if args.wipe:

wipe\_database()

Liste 5-12: Kullanıcı komut satırı argümanlarına göre benzerlik veritabanı güncellemeleri ve sorguları gerçekleştirme

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.

mds@mds:~/malware\_data\_science/ch5/code$ python listings\_5\_7\_to\_5\_12.py -l ../

data

Extracted 240 attributes from ../data/APT1\_MALWARE\_FAMILIES/WEBC2-YAHOO/WEBC2-

YAHOO\_sample/WEBC2-YAHOO\_sample\_A8F259BB36E00D124963CFA9B86F502E ...

Extracted 272 attributes from ../data/APT1\_MALWARE\_FAMILIES/WEBC2-YAHOO/WEBC2-

YAHOO\_sample/WEBC2-YAHOO\_sample\_0149B7BD7218AAB4E257D28469FDDB0D ...

Extracted 236 attributes from ../data/APT1\_MALWARE\_FAMILIES/WEBC2-YAHOO/WEBC2-

YAHOO\_sample/WEBC2-YAHOO\_sample\_CC3A9A7B026BFE0E55FF219FD6AA7D94 ...

Extracted 272 attributes from ../data/APT1\_MALWARE\_FAMILIES/WEBC2-YAHOO/WEBC2-

YAHOO\_sample/WEBC2-YAHOO\_sample\_1415EB8519D13328091CC5C76A624E3D ...

Extracted 236 attributes from ../data/APT1\_MALWARE\_FAMILIES/WEBC2-YAHOO/WEBC2-

YAHOO\_sample/WEBC2-YAHOO\_sample\_7A670D13D4D014169C4080328B8FEB86 ...

Extracted 243 attributes from ../data/APT1\_MALWARE\_FAMILIES/WEBC2-YAHOO/WEBC2-

YAHOO\_sample/WEBC2-YAHOO\_sample\_37DDD3D72EAD03C7518F5D47650C8572 ...

--snip--

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

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

Sample name Shared code

estimate

[\*] GREENCAT\_sample\_5AEAA53340A281074FCB539967438E3F    1.0

[\*] GREENCAT\_sample\_1F92FF8711716CA795FBD81C477E45F5    1.0

[\*] GREENCAT\_sample\_3E69945E5865CCC861F69B24BC1166B6    1.0

[\*] GREENCAT\_sample\_AB208F0B517BA9850F1551C9555B5313    1.0

[\*] GREENCAT\_sample\_3E6ED3EE47BCE9946E2541332CB34C69    0.99609375

[\*] GREENCAT\_sample\_C044715C2626AB515F6C85A21C47C7DD    0.6796875

[\*] GREENCAT\_sample\_871CC547FEB9DBEC0285321068E392B8    0.62109375

[\*] GREENCAT\_sample\_57E79F7DF13C0CB01910D0C688FCD296    0.62109375

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.

!!!!!!!

İLK OLARAK BU KİTABI UYGULA ÖZET ÇIKAR

PRACTİCAL MALWARE ANALYST KİTABINA BAŞLAMAM LAZIM