FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Deteção de nomes de domínios gerados aleatoriamente

António Jorge Aguiar do Vale

DISSERTAÇÃO

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Ricardo Morla

26 de Julho de 2019

Deteção de nomes de domínios gerados aleatoriamente

António Jorge Aguiar do Vale

Mestrado Integrado em Engenharia Informática e Computação

26 de Julho de 2019

Abstract

Nowadays, botnets depend on domain generation algorithms to construct command and control infrastructures resistant to detection. Thus, with the prevalence of this mechanism, there has been much discussion in the automated detection of domains coming from such algorithms.

The main objective of this work is the study of the use of machine learning for the detection of domains generated by algorithms and their classification by their name.

The development of this work began with the study of domain generation algorithms that previously had an impact on global security. These algorithms were brought to the general public through the use of reverse engineering in domains already used by different malware families. Second, with the algorithms already known, domains were generated in order to have relevant data to go to the next step of machine learning. This next stage of the project is defined by the implementation of the model to be used.

In a final part of this work some experiments were conducted to better understand what should be used for the final model, as well as to analyse characteristics in machine learning in the detection of domains coming from generation algorithms such as prediction on domains from the same malware family but using different seeds. Experiments were also performed using Alexa domains in order to understand the impact in training of different data sets.

Resumo

Hoje em dia, *botnets* dependem de algoritmos de geração de domínio para construir infraestruturas de comando e controle resistentes à sua deteção. Assim, com a prevalência desse mecanismo, tem havido muita discussão na deteção automatizada de domínios provenientes de tais algoritmos.

O objetivo principal deste trabalho é o estudo do uso de aprendizagem computacional para a deteção de domínios gerados por algoritmos de geração de domínios e sua classificação através do nome dos domínios.

O desenvolvimento deste trabalho iniciou-se pelo estudo de algoritmos de geração de domínio que previamente já tiveram impacto na segurança global. Esses algoritmos foram divulgados ao público geral por meio do uso de engenharia reversa em domínios já utilizados pelas diferentes famílias de *malware*. Em segundo lugar, com os algoritmos já conhecidos, foram gerados domínios de forma a ter dados relevantes para passar ao próximo passo de aprendizagem computacional. Esta próxima etapa do projeto é definida pela implementação do modelo a ser utilizado.

Numa parte final da realização deste trabalho, foram feitas algumas experiências relativamente aos domínios usados no treino do modelo e para analisar características no uso de aprendizagem computacional na deteção de domínios provenientes de algoritmos de geração, como previsão de domínios da mesmo família de *malware*, mas usando sementes pseudo-aleatórias diferentes. Foram também realizadas experiências utilizando domínios do Alexa de forma a entender o impacto no treino com diferentes domínios.

Agradecimentos

Aos meus pais e meu irmão, o meu sincero obrigado por todo o amor, carinho, apoio e confiança que depositaram em mim dia após dia. Pela disponibilidade e por colocarem a minha educação à frente de tudo e dar-me oportunidade de ter percorrido este percurso de que nada me arrependo.

À Valentina, minha namorada, que teve a paciência para me aturar nos dias mais difíceis e por me ter apoiado durante esta jornada, sempre a meu lado. Pelo carinho e preocupação que teve sempre comigo e o meu bem-estar. Obrigado pela dedicação e amor que dedicaste à nossa relação.

Ao meu orientador, professor Ricardo Morla, que desde o primeiro dia do desenvolvimento desta dissertação demonstrou um apoio e disponibilidade que nunca tinha experienciado a nível académico. Durante o decorrer deste projeto, houve alterações que tiveram que ser feitas e novas ideias surgiram e agradeço ao professor mais uma vez pela dedicação a esta dissertação, sempre à procura do melhor deste projeto, e pela atenção prestada durante este tempo todo.

Aos meus amigos, pelas noitadas, embora trabalhosas, bem passadas na FEUP, pela discussão de ideias que também contribuíram para a realização desta dissertação, pela paciência e pela ajuda. Momentos que estarão sempre presentes na minha vida e que fizeram esta jornada mais brilhante do que jamais imaginara.

Aos amigos de infância e de tempos mais antigo que, embora o tempo passado distante, a disponibilidade para falar nunca se desgastou. A boa disposição e carinho que mantiveram mesmo perante tempos de ausência devido à vida. Aos momentos de descontração e aos convites para tomar café apenas servindo de desculpa para retomar presença na vida deles.

Não esquecendo o meu grupo, os PvP, que, embora não esteja as vezes que gostaria com eles presencialmente, fizeram parte da minha vida quase todos os dias seja durante o trabalho à frente ao computador, seja pelas horas bem passados nas formas variadas de entretimento online.

A Todos,
O meu sincero agradecimento.

Jorge Vale.

"Life isn't about finding yourself.
Life is about creating yourself."

George Bernard Shaw

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

DGA	Domain Generation Algorithm
C&C	Command and Control
IP	Internet Protocol
LSTM	Long Short-Term Memory
NXDomains	Non-Existent Domains
DNS	Domain Name System
URL	Uniform Resource Locator
P2P	Peer to Peer
2LD	Domínio de segundo nível
API	Application Programming Interface
LMS	Longest Meaningful Substring

Capítulo 1

Introdução

Este primeiro capítulo contém uma descrição breve do projeto desenvolvido durante a realização desta dissertação, que atua na deteção automática de DGAs através do nome de domínios. Para além da descrição do projeto do qual este documento se baseia, este também analisa o enquadramento do projeto, que visa analisar projetos e trabalhos feito sobre a deteção de DGAs utilizando a aprendizagem computacional, assim como os problemas que o mesmo atua e os seus respetivos objetivos a realizar.

1.1 Contexto

Hoje em dia, e cada vez mais, observa-se um aumento contínuo de utilizadores na *Internet*, sejam eles pessoais, empresariais e/ou governamentais. Como consequência desta variedade de utilizadores, presencia-se na *Internet* também diferentes tipos de informação, podendo-se dividir estes tipos em três: pública, privada e privilegiada, sendo que cada um destes tipos apresenta diferentes restrições a nível de acesso.

Tabela 1.1: Previsão de Tráfego Global da Cisco [8]

Ano	**Tráfego Global Internacional**
1992	*100 GB por dia*
1997	*100 GB por hora*
2002	*100 GB por segundo*
2007	*2,000 GB por segundo*
2017	*46,600 GB por segundo*
2022	*150,700 GB por segundo*

Como podemos ver na Tabela1.1, o tráfego na *Internet* tem vindo a aumentar de uma forma exponencial nos passados vinte anos e irá continuar a aumentar conforme a previsão de Cisco[8] sendo que chegará aos cento e cinquenta mil e setecentos GB por segundo e trezentos e noventa e

seis *EB*(Figura1.1). A informação também aumenta, não só pública mas também informação privada e privilegiada que, por diferentes razões, requer um acesso restrito, seja informação pessoal, bancária, governamental, etc. Assim, com esta abundância de informação restrita pode levar a que pessoas com más intenções tentem aceder a essa informação na qual o acesso lhes é negado. Com este objetivo, estes utilizadores maliciosos usam *botnets*, que são um conjunto de máquinas(*bots*) que são remotamente controlados pelo atacante através de um canal de comando e controlo (C&C).

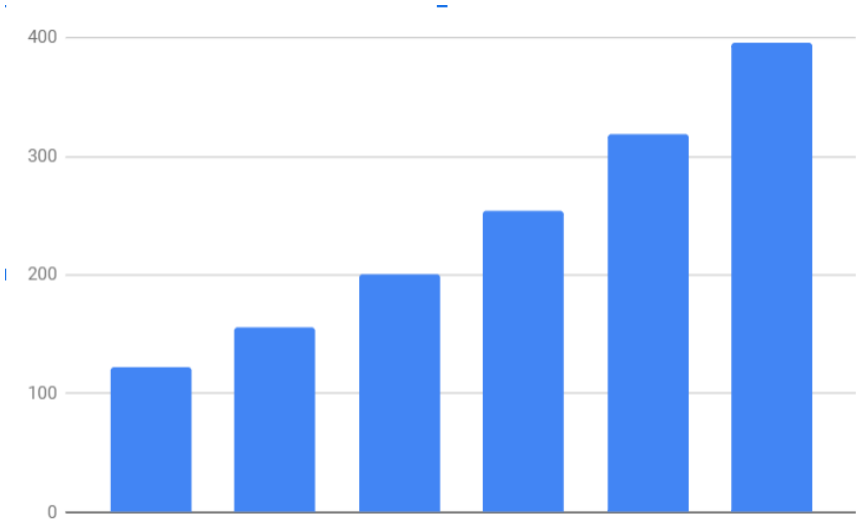

Figura 1.1: Previsão de tráfego IP por Cisco(2017-2022)
[8]

Com o decorrer do tempo, já se identificaram diferentes métodos de implementação destes canais usados para realizar C&C. Inicialmente, maior parte dos servidores C&C eram centralizados o que levava a um problema para os atacantes([9]). Os *bots* consultavam o domínio C&C predefinido que resolvia para o endereço IP do servidor C&C a partir do qual os comandos são recebidos. Isto leva a que se o domínio C&C fosse identificado e removido e o atacante perderia o controlo sobre toda a *botnet*. Para resolver este problema, começaram a aparecer *botnets* que usamos estruturas P2P para implementação de C&C, no entanto a implementação era bastante difícil e custosa o que inviabiliza este método.

Com o objetivo de desafiar os métodos de segurança e conseguir manter a anonimidade, foi desenvolvido um método em que o atacante insere em máquinas infetadas um algoritmo que gera domínios aleatórios através de uma *seed* conhecida apenas por ele mesmo. Conforme a geração dos domínios aleatórios, as máquinas infetadas vão fazendo pedidos a esses domínios gerados anteriormente([10]). Com este método, as máquinas infetadas fazem bastantes pedidos a domínios falsos e obtêm respostas de domínios não existentes (NXDomain) aquando o pedido feito. No entanto, um domínio entre os milhares gerados diariamente será um domínio existente e válido que, o atacante através do conhecimento prévio do algoritmo e da *seed* conseguiu registar. A máquina ao fazer o pedido a este domínio escolhido pelo atacante não terá como resposta NXDomains mas sim o IP desse domínio, que será o servidor C&C. Assim, através da complexidade dos algoritmos

e da existência de *seeds* complexas torna-se bastante difícil a analistas de segurança saber que domínios entre os milhares serão os que realmente o atacante irá usar para contactar a máquina infetada.

1.2 Motivação e Objetivos

Este trabalho tem como principal motivação a dificuldade na deteção manual de domínios provenientes de DGAs que, para analistas de segurança, conseguir detetar que domínio faz parte de um DGA manualmente é preciso fazer um trabalho complexo de *reverse engineering* o que é bastante demorado, o que leva a que, por vezes, consegue-se descobrir. No entanto, essa descoberta pode ser tardia o que poderá levar graves prejuízos à empresa e/ou utilizador. Assim, a motivação do desenvolvimento desta dissertação tem por base o aumento de segurança na tecnologia usada hoje em dia.

Esta dissertação estuda a deteção automática de domínios criados por DGAs com suporte a *keras*, uma *framework* de *python* que fornece funções auxiliares no desenvolvimento e treino de modelos de *machine learning*.

Embora já existam algumas ferramentas que tentam ajudar no referido problema, o objetivo final é conseguir criar uma ferramenta com uma elevada taxa de sucesso na deteção de DGA, tentando diminuir ao máximo falsos positivos e negativos. Assim, o foco desta dissertação é estudar o uso de *machine learning* na deteção automática de domínios vindos de DGAs e, posteriormente, ter uma taxa de sucesso elevada de deteção perante diferentes DGAs previamente conhecidos e, de seguida, criar uma ferramenta que utilize o resultado final do estudo e do treino do modelo final para facilitar o trabalho quer a analistas de segurança em empresas quer a utilizadores.

1.3 Estrutura da Dissertação

Esta dissertação tem 6 capítulos. O primeiro capítulo é esta que serve de um contacto inicial ao leitor sobre o que será tratado neste documento, contêm um enquadramento e o objetivo final da dissertação.

Os dois capítulos seguintes referem a revisão bibliográfica relevante para o tema desta dissertação. O segundo capítulo analisa DGAs existentes, o terceiro revê trabalho relacionado já existente sobre o tema.

O quarto capítulo apresenta a metodologia usada no desenvolvimento do projeto, que inclui os algoritmos e ferramentas usadas para a sua implementação.

O quinto sexto capítulo demonstra as experiências feitas durante o projeto e analisam os resultados obtidos quer no modelo multiclasse e binário.

O último capítulo apresenta as conclusões retiradas durante o desenvolvimento deste trabalho assim como uma perspetiva futura de trabalho futuro.

Capítulo 2

Visão geral de DGAs

Neste capítulo são descritos os diferentes algoritmos de geração de domínios utilizados neste trabalho. São algoritmos existentes e que já tiveram um impacto grande a certa altura. Estes algoritmos, como explicado no Capítulo1.1, utilizam diferentes sementes de algoritmos pseudo-aleatórios (*seeds*) para tentar aumentar a complexidade do algoritmo. Após uma análise dos diferentes algoritmos existentes, este trabalho divide as *seeds* em quatro grandes grupos: *string*, número hexadecimal, número inteiro e a data no qual o algoritmo é executado ou uma definida no algoritmo.

De modo a conseguir perceber melhor os algoritmos e as suas semelhanças/diferenças entre eles foram analisados diagramas que mostravam a abundância de caracteres nos domínios criados pelos diferentes algoritmos.

Em todos os algortimos são apresentados algumas partes da geração de domínios e uns domínios exemplos gerados pelos mesmos algoritmos sem o *top-level domain*.

2.1 Banjori

```
1 dl = [ord(x) for x in list(seed)]
2 dl[0] = map_to_lowercase_letter(dl[0] + dl[3])
3 dl[1] = map_to_lowercase_letter(dl[0] + 2*dl[1])
4 dl[2] = map_to_lowercase_letter(dl[0] + dl[2] - 1)
5 dl[3] = map_to_lowercase_letter(dl[1] + dl[2] + dl[3])
```

Banjori [11], também chamado de MultiBanker, Patcher e BankPatcher é um tipo de *malware Trojan* bancário que começou a aparecer desde o início de 2007. Banjori é um *malware* muito específico, quer nas suas vítimas, quer no seu uso, foi desenvolvido e usado apenas por um grupo e tem como alvos países e bancos predefinidos, sendo que os principais foram Áustria e Alemanha. Devido à especificada de Banjori, o servidor C&C responde apenas a endereços vindo da Europa

O algoritmo de geração usado em Banjori recebe uma *string* e vai alterando os quatro primeiro dígitos com o decorrer do algoritmo. Este mantêm o endereço IP do servidor C&C igual durante semanas no entanto o domínio respetivo altera-se diariamente.

Por último, este algoritmo tem como objetivo roubar dinheiro aos utilizadores de máquinas infetadas. Sempre que é realizada uma transação bancária online numa máquina infetada, o pedido antes de ser realizado o número da conta do beneficiário é alterado.

Domínios gerados: kwtoestnessbiophysicalohax, rvcxestnessbiophysicalohax, hjbtestnessbiophysicalohax, txmoestnessbiophysicalohax, agekestnessbiophysicalohax, dbzwestnessbiophysicalohax, sgjxestnessbiophysicalohax, igjyestnessbiophysicalohax, zxahestnessbiophysicalohax, zfrpestnessbiophysicalohax.

2.2 Corebot

```
len_l = 0xC
len_u = 0x18
r = (1664525*r + 1013904223) & 0xFFFFFFFF
domain_len = len_l + r % (len_u - len_l)
domain = ""
for i in range(domain_len, 0, -1):
    r = ((1664525 * r) + 1013904223) & 0xFFFFFFFF
    domain += charset[r % len(charset)]
domain += tld
```

Corebot [12] é conhecido pelo seu sistema de *plugins*, tornando-o fácil de alterar e acrescentar novas capacidades de roubo. O Corebot faz o *download* de *plugins* do seu servidor C&C e depois carrega os *plugins* usando a função de exportação *plugininit* na *Dynamic Link Library*, DLL, do *plugin*.

Domínios gerados: ahodu2kpmpw614itgv3, mt32uxgfmp7d5tgjotc, qdqxkr3t1fifu41, i0k6mnyhejwlsnw, s23t1dkjaxwd50iduhg, w2c6k2ehkpid1nux3hs, 3lgrst5rurm61jyl7xihw8k, qfgxy0mbepy8sxy, yro4kf5da2q6qlq8w0epelu ubc81v38y8ut3xy.

2.3 Cryptolocker

```
year = ((year ^ 8 * year) >> 11) ^ ((year & 0xFFFFFFF0) <<
    17)
month = ((month ^ 4 * month) >> 25) ^ 16 * (month & 0
    xFFFFFFF8)
```

```
day = ((day ^ (day << 13)) >> 19) ^ ((day & 0xFFFFFFFE) <<
    12)
domain += chr(((year ^ month ^ day) % 25) + 97)
```

Cryptolocker [13] foi a família de *malware* que deu ao mundo o conhecimento de DGAs em 2013, tornando-o famoso. Este infetou mais de 200,000 computadores a nível mundial, sendo metade localizados nos Estados Unidos da América e os autores deste *malware*, apenas nos primeiros dois meses, adquiriram mais de 27 milhões de dólares em pagamentos.

Os atacantes que usam Cryptolocker recebem o dinheiro a partir de pagamentos das vitimas. Estas vitimas pagam porque o Cryptolocker pega em todos os ficheiros no hardware do computador, incluindo em dispositivos externos ligados ao computador, e encripta os documentos com uma chave pública na máquina infetada. Estes documentos apenas podem ser desencriptados apenas com uma chave privada que o atacante conhece, então as pessoas pagam para recuperar os seus documentos.

Com a abrangência tão grande deste *malware*, foram criados vários *malwares* a tentar imitar o mesmo no entanto não conseguirem ter o sucesso a nível de encriptação que o mesmo teve.

Domínios gerados: ejfodfmf, vtlfccmf, nerrjcct, foxiiyct, wyeupyai, ojklovai, btbpurnk, sehccrly, konsboly, cytfiobn.

2.4 DirCrypt

```
ix = self.seed
ix = int(16807*(ix % 127773) - 2836*(ix / 127773)) & 0
    xFFFFFFFF
self.seed = ix
```

DirCrypt [14] também é um *malware* que se aproveita do acesso a máquinas infetadas para encriptar os ficheiros dessa mesma máquina e exigir dinheiro ao utilizador parar recuperar os mesmos.

DirCrypt espalha-se através de emails de *spam* ou por fazer download direto de uma fonte não segura. Este *malware* apenas afetou computadores *Windows* e uma máquina, aquando afetada, podia sofrer uma diminuição de segurança da mesma devido a alterações feitas pelo *malware*.

Domínios gerados: eoirycofnywgmw, dqtmarskjibchpdguvxz, rvxbooexbnhtbf, pefjimjo, xsoqioqku, uzkcpzosaonuoqu, wophsufgoybgwpwg, hpwpepvwfaazkpzobji, zyfrteedbnmcewq, fdfiydtoevdtuvxdv.

2.5 Fobber

```
ror32 = ((v >> n) | (v << (32-n))) & 0xFFFFFFFF
r = ror32((321167 * r + c) & 0xFFFFFFFF, 16)
domain += chr( (r & 0x17FF) % 26 + ord('a') )
```

Fobber [15] concentra-se na parte de roubo bancário assim como Banjori, no entanto este, ao invés de alterar diretamente o beneficiário de uma transação bancária, concentra-se mais em adquirir dados que foram preenchidos em formulários em máquinas infetadas.

Uma característica que sobressaiu deste *malware* foi que consegue-se atualizar sozinho, sendo que os atacantes que utilizam o mesmo conseguem fazer alterações à medida que este é analisado, tornando muito mais difícil a análise do mesmo.

Domínios gerados: pozqbwdduuabttngq, wbshytimicgmspssi, lrgeevxxtqtqnvypr, mnyvhpnkthbzcpmxh, mvgbqpdmxhlvapzhp, rvtsptxwthlmlrbhw, urfcibcffmuijggrq, kazbcgdqsbjhcmneo, zrxczrrvojhlrzyqa, ransbpjnorlwkpqjr.

2.6 Kraken

```
rands = 3*[0]
for i in range(3):
    r = rand(r)
    rands[i] = crop(r)
domain_length = (rands[0]*rands[1] + rands[2]) % 6 + 7
domain = ""
for i in range(domain_length):
    r = rand(r)
    ch = crop(r) % 26 + ord('a')
    domain += chr(ch)
```

Foi a primeira família de *malware* que usou DGA, em 2008, sendo que esta família de *malware* foi considerada a *botnet* maior do mundo com mais de 400,000 *bots* em, pelo menos, 50 das 500 empresas referidas na Fortune 500(lista anual compilada, publicada pela revista Fortune que contém as 500 maiores empresas dos Estados Unidos por receita total nos respetivos anos fiscais)[16].

O DGA que Kraken usa, inicialmente, era estático no entanto, mais tarde, começaram a usar uma *seed* para a geração de domínios consoante o tempo atual(ano, mês, dia, hora e minuto) no momento de geração do domínio [17].

Domínios exemplo: syxyymsruy, lvctmusxcyz, egbmbdey, iuhqhbmq, toogdpdiekwh, egbmbdey, qqvgpzewpaj, vplyype, vsdvzwt, protqvuwt.

2.7 Locky

```
seed_shifted = rol32(c['seed'], 17)
dnr_shifted = rol32(domain_nr, 21)
k = 0
year = date.year
for _ in range(7):
    t_0 = ror32(0xB11924E1 * (year + k + 0x1BF5), c['shift'
        ]) & 0xFFFFFFFF
    t_1 = ((t_0 + 0x27100001) ^ k) & 0xFFFFFFFF
    t_2 = (ror32(0xB11924E1 * (t_1 + c['seed']), c['shift'
        ])) & 0xFFFFFFFF
    t_3 = ((t_2 + 0x27100001) ^ t_1) & 0xFFFFFFFF
    t_4 = (ror32(0xB11924E1 * (date.day//2 + t_3), c['shift
        ])) & 0xFFFFFFFF
    t_5 = (0xD8EFFFFF - t_4 + t_3) & 0xFFFFFFFF
    t_6 = (ror32(0xB11924E1 * (date.month + t_5 - 0x65CAD),
         c['shift'])) & 0xFFFFFFFF
    t_7 = (t_5 + t_6 + 0x27100001) & 0xFFFFFFFF
    t_8 = (ror32(0xB11924E1 * (t_7 + seed_shifted +
        dnr_shifted), c['shift'])) & 0xFFFFFFFF
    k = ((t_8 + 0x27100001) ^ t_7) & 0xFFFFFFFF
    year += 1
length = (k % 11) + 7
domain = ""
for i in range(length):
    k = (ror32(0xB11924E1*rol32(k, i), c['shift']) + 0
        x27100001) & 0xFFFFFFFF
    domain += chr(k % 25 + ord('a'))
```

Domínios gerados: nhxaxng, alctdak, qpcnfkvsmqrjve, ihksmugfuqhg, dxntbtvxmtu, rikuaprdyaat, hryqwrexvnp, nuobathplwgc, rvgnjabclr, qrxpljgtsjfssepf.

2.8 Murofet

```
seed = 7*[0]
seed[0] = ((date.year & 0xFF) + 0x30) & 0xFF
```

```
seed[1] = date.month
seed[2] = (date.day//7)*7
seed_str = seed
m = hashlib.md5()
m.update(seed_str.encode('utf-8'))
md5 = m.digest()
domain = ""
for m in md5:
    d = (m & 0x1F) + ord('a')
    c = (m >> 3) + ord('a')
    if d != c:
        if d <= ord('z'):
            domain += chr(d)
        if c <= ord('z'):
            domain += chr(c)
```

Domínios gerados: lfvbmdehfakhubqpgxcajramq, vwtksoucsglsgpnmrnrhsoovbu, ofekbyylxcrwheuwlfdnjsnb, xccmydemjfeyrowgpkrfujzgyhmr, oncmqsorjrphahmkfytxuscmnvx, dgmugsogqrseytirwkxpbxp, dqhrshqqohkrxgpxgyx, ydtorlfhwqktkmvtkgmdgaxgbafm, kbpfknozqoqkzteqojumzt, vcifswxinwltosogezdeidaogi.

2.9 Necurs

```
for index in range(random):
    value += ((value*7) ^ (value << 15)) + 8*index - (value
        >> 5)
    value &= ((1 << 64) - 1)
n = pseudo_random(date.year)
n = pseudo_random(n + date.month + 43690)
n = pseudo_random(n + (date.day>>2))
n = pseudo_random(n + sequence_nr)
n = pseudo_random(n + magic_nr)
domain_length = mod64(n, 15) + 7
```

Domínios gerados: jygtnmxrtewk, afmtyyocxbxcuqlsfgb, dclweqieukl, ijioqeynojkfky, unpfougrcnao, deoiasnykpvnhvbgjoi, axmsxayrhjunjdoxrlrsd, jqxviaylld, farqfurxjyp, bovokkbjllwhjqvqce.

2.10 Pykspa

```
modulo = 541 * length + 4
a = length * length
for i in range(length):
    index = (a + (seed*((seed % 5) + (seed % 123456) +
            i*((seed & 1) + (seed % 4567))) & 0xFFFFFFFF))
                % 26
    a += length
    a &= 0xFFFFFFFF
    sld += chr(ord('a') + index)
    seed += (((7837632 * seed * length) & 0xFFFFFFFF) +
       82344) % modulo
```

Domínios gerados: awomwyiuecaq, jgfmccp, baitgxbvbpn, xqdeip, ouacegiw nrdrxhf, xduwompbrxrc, bobingbnnx, tkjixivutt, wfxxotmpok.

2.11 Qakbot

```
class MT19937:
    def __init__(self, seed):
        # Initialize the index to 0
        self.index = 624
        self.mt = [0] * 624
        self.mt[0] = seed  # Initialize the initial state
           to the seed
        for i in range(1, 624):
            self.mt[i] = _int32(
                1812433253 * (self.mt[i - 1] ^ self.mt[i -
                   1] >> 30) + i)
    def extract_number(self):
        if self.index >= 624:
            self.twist()
        y = self.mt[self.index]
        y = y ^ y >> 11
        y = y ^ y << 7 & 2636928640
        y = y ^ y << 15 & 4022730752
```

```
17          y = y ^ y >> 18
18          self.index = self.index + 1
19          return _int32(y)
20      def twist(self):
21          for i in range(0, 624):
22              y = _int32((self.mt[i] & 0x80000000) +
23                         (self.mt[(i + 1) % 624] & 0x7fffffff
                               ))
24              self.mt[i] = self.mt[(i + 397) % 624] ^ y >> 1
25              if y % 2 != 0:
26                  self.mt[i] = self.mt[i] ^ 0x9908b0df
27          self.index = 0
28      def rand_int(self, lower, upper):
29          r = self.extract_number()
30          r &= 0xFFFFFFF
31          t = lower + float(r) / (2**28)*(upper - lower + 1)
32          t = int(t)
33          return t
```

Domínios gerados: tlstukjiihwcfjysh, grzdxsurypvg, ycgirscsh, senccrgchupgxj, ladrkfpu, luvxiuhiomqjrbznujhhpvtj, whouvlzdbkjqey, bdkfsydbqpromazvfknweffke, aqobzbxhmrsnmx erbmilrirzismcwnamg

2.12 Ramdo

```
1  while domain_length < length:
2      # xor1_divide = xor1 / 0x1a
3      xor1_remainder = xor1 % 0x1a
4      xo1_rem_20 = xor1_remainder + 0x20
5      xo1_step2 = xo1_rem_20 ^ 0xa1
6      dom_byte = 0x41 + (0xa1 ^ xo1_step2)
7      dom += chr(dom_byte)
8      imul_iter = domain_length * step1
9      imul_result = domain_length * imul_iter
10     imul_1a = 0x1a * imul_result
11     xor2 = xor1 ^ imul_1a
12     xor1 = xor1 + xor2
13     domain_length += 1
```

Domínios gerados: ocomiakmywmiwsig, qgykmagmiaoweime, ceicisyuckmcusqu, myecmyaagmaoaasw, mycsgqqswkswokam, ockmweuegsakciyc, ywqqokcuayomkmgq, wsuikgqceyauqcko, eioqmsesmkykieme, ceuusyykeacasamw.

2.13 Ramnit

```
ix = self.seed
ix = 16807*(ix % 127773) - 2836*(ix / 127773) & 0xFFFFFFFF
self.seed = ix
```

Domínios gerados: ydctqhqvjbkldnfwd vbjgudvjxmbwqr situhqgt elxquoejqkqkfh pleobkhvinjssgiv hggcvcqdxgx tohgaucteyvgepmhls ixtissuwfqaxkachax ybgcrtxdgkjbfgrua ujtcxnaotbqfipaocyr

2.14 Simda

```
for i in range(length):
    index = int(base/(3+2*i))
    if i % 2 == 0:
        char = consonants[index % 20]
    else:
        char = vowels[index % 6]
    domain += char
```

Domínios gerados: gatyfusyfi, lyvyxoryco, vojyqemute, qetyfuviku, puvyxilomo, gahyqahofa, lyryfydacy, vocyziteti, qegyqaqeko, purydyvyme.

Capítulo 3

Trabalho relacionado

Aqui serão descritos os trabalhos relacionados com o tema desta dissertação, tendo como objetivo de explorar os trabalhos que, de certea forma, se enquadram no mesmo tema que este documento. Aqui estão destacadas os trabalhos que achei mais importantes rever, seja pelos seus pontos fortes como fracos.

3.1 Notos

Notos é um sistema de reputação dinâmica para DNS. Notos usa dados de consulta DNS passivos, analisa a rede e características de domínios. Para alcançar este objetivo, de conseguir classificar e dar uma pontuação que reflete se certo URL possa ser malicioso ou não, Notos cria dois tipos de modelos, um que contêm domínios legítimos e outro que possui domínios maliciosos previamente conhecidos. A partir destes dois modelos é calculada a legitimidade de um novo URL (uma pontuação indicativa se o domínio é malicioso ou não).

De modo a avaliar os domínios novos Notos analisa várias esta características destes, sendo que podemos dividir estas características em três grandes grupos: história relacionada do IP, história relacionada de cada domínio e, por último, características tendo por base evidências. O primeiro grupo descreve como é que os operadores têm certo domínio e os IPs para os quais esse domínio aponta os seus recursos de rede. O segundo grupo analisa o nome do domínio e o histórico das *substrings* que compoêm o domínio. Depois, o último grupo tenta avaliar até que ponto é que este domínio está ligado a outros domínios/IPs conhecidos que sejam maliciosos[1].

Para a avaliação deste sistema, foram usados 1,4 milhões de utilizadores, sendo que obtiveram uma precisão de 96,8% e falsos positivos de 0,38%.

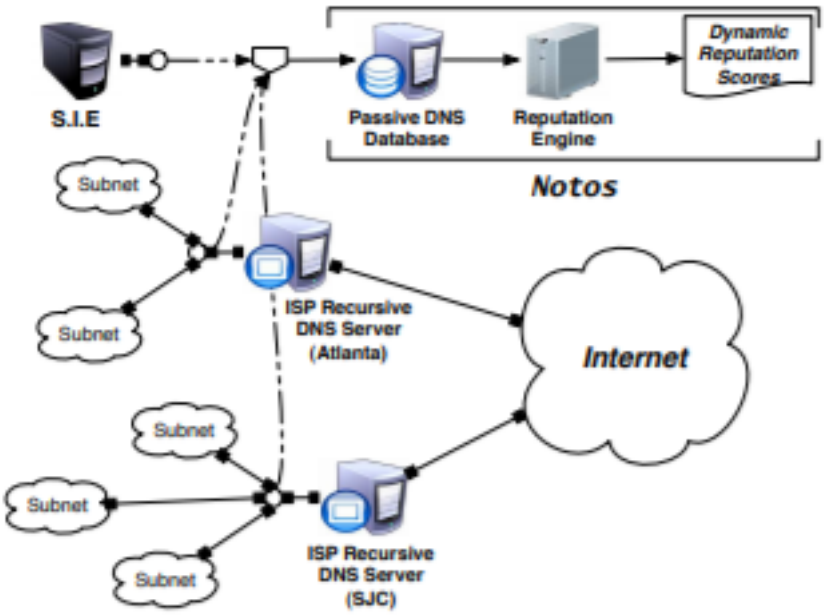

Figura 3.1: Funcionamento de Notos [1]

3.2 Kopis

Kopis, o segundo sistema, emprega dados DNS coletados de uma hierarquia de DNS superior, conseguindo assim analisar padrões globais de consultas de DNS. Este diferencia-se de Notos, referido na seção anterior, pois não obriga a conhecimento anterior do IP para classificar um domínio como maligno ou não.

O funcionamento de Kopis(Figura 3.2) divide-se em duas partes. A primeira foca-se no treino do conhecimento do sistema, sendo que se baseia numa base de dados constituída por domínios conhecidos que foram usados como *malware* e outros domínios que são considerados confiáveis[2].

A parte restante nomeia-se como modo de operação em que este sistema monitoriza tráfegos de DNS e separa os domínios que não sejam conhecidos ao sistema dos outros. Quando este tem os domínios desconhecidos à parte, estes passam por um classificador estático que atribui um rótulo a cada domínio e a sua confiança.

De modo a avaliar os domínios Kopis baseia-se nas seguintes características: o perfil da máquina solicitadora e a sua diversidade e a reputação dado pelo classificador que compara o domínio com domínios existentes anteriormente. A avaliação deste sistema foi feita durante 8 meses com dados reais e resultou numa avaliação de 98.4% de deteção e falsos positivos de 0.3% ou 0.5%.

3.3 Pleiades

Pleiades, um sistema mais recente, sendo que é o que se foca mais na deteção de DGA, analisando consultas DNS que resultem em NXDomains.

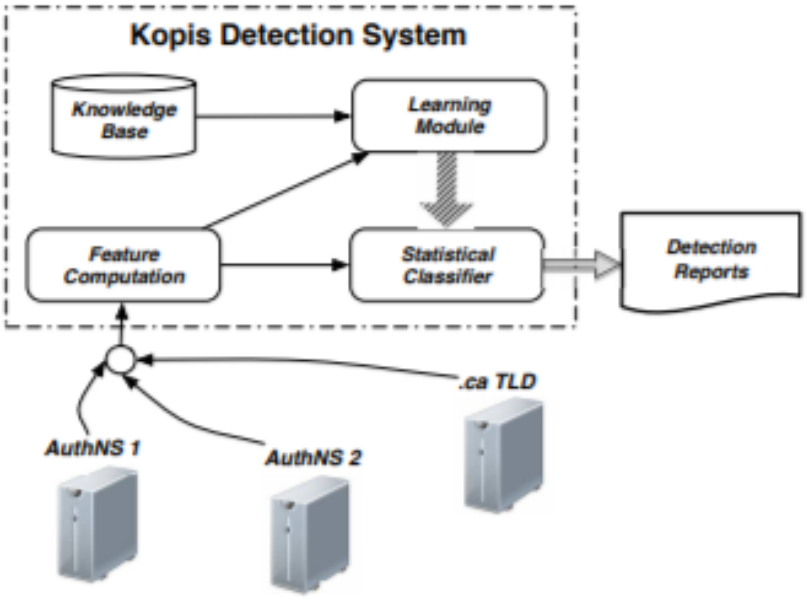

Figura 3.2: Funcionamento de Kopis [2]

Pleiades baseia-se no pressuposto de que em redes grandes possam haver máquinas contaminadas pelo mesmo DGA e assume que, ao obter um conjunto de domínios que tenha como resposta NXDomain, consegue construir um modelo que, posteriormente, sirva de base para detetar domínios que sejam provenientes do mesmo DGA.

Assim, para conseguir automaticamente detetar domínios provenientes de DGA, o sistema Pleiades procura por *clusters* grandes de NXDomains que têm recursos sintáticos e são consultados por várias máquinas potencialmente comprometidas durante uma determinada época. Este sistema, é capaz de identificar e filtrar automaticamente domínios NXDomais que sejam criados acidentalmente devido a erros ou configurações incorretas pois, embora adicione erros no treino do modelo, este assume que o número de ocorrências destes erros são demasiado pequenas para terem um impacto no modelo final. Quando Pleiades encontra um conjunto de NXDomains, aplica técnicas de aprendizagem estatística para construir um modelo, que depois é usado para detetar domínios parecidos em máquinas que, futuramente, possam estar "infetadas" [3].

Pleiades tem algumas limitações como por exemplo, uma vez que um novo DGA é descoberto, Pleiades consegue desenvolver modelos estatísticos bastante precisos através da parecença dos domínios que obtiveram a resposta NXDomains, no entanto é incapaz de aprender ou reconstruir o algoritmo exato de geração de domínio. Tendo por base esta incapacidade, Pleiades irá gerar um certo número de falsos positivos e falsos negativos

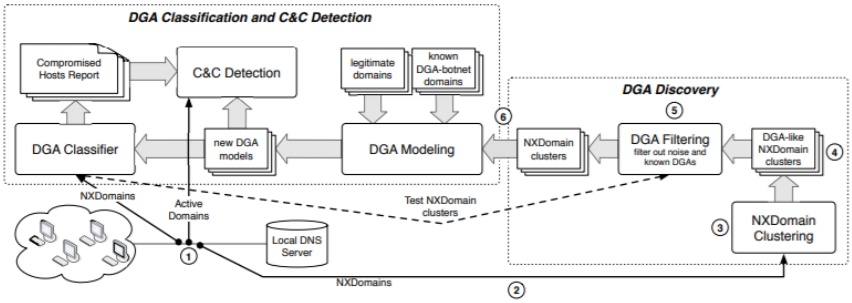

Figura 3.3: Funcionamento de Pleíades [3]

3.4 Exposure

Exposure é um sistema no qual o objetivo é a deteção de domínios que possam estar envolvidos em atividades maliciosas. Para isto, retira 15 características(Tabela 3.1) diferentes dos pedidos de DNS e analisa-os, usando essas mesmas características para criar um modelo de treino para conseguir diferenciar domínios maliciosos de domínios benignos.

Como podemos ver na Figura 3.4, Exposure usa primeiro o *Data Collector*, este recebe a informação da rede que o Exposure está a analisar a uma certa altura, depois é realizado um *feature selection* para obter consultas de DNS que contenham as características presentes na Tabela 3.1, de forma a posteriormente conseguirem analisar o domínio. Depois temos o *Learning Module* que por sua vez pega na informação coletada no *Malicious/Benign Data Collector* e treina um modelo para conseguir futuramente diferenciar domínios maliciosos de benignos. O último componente, *Classifier*, utiliza o modelos previamente criado e é aqui que dados ainda não classificados recebem uma nomeação de maliciosos ou benignos [4].

De forma a avaliar o resultado final, o modelo foi classificado através de uma *10-fold cross validation* em que 66% da informação foi usada como treino e o resto como teste. Os falsos positivos foram domínios benignos classificados como malignos.

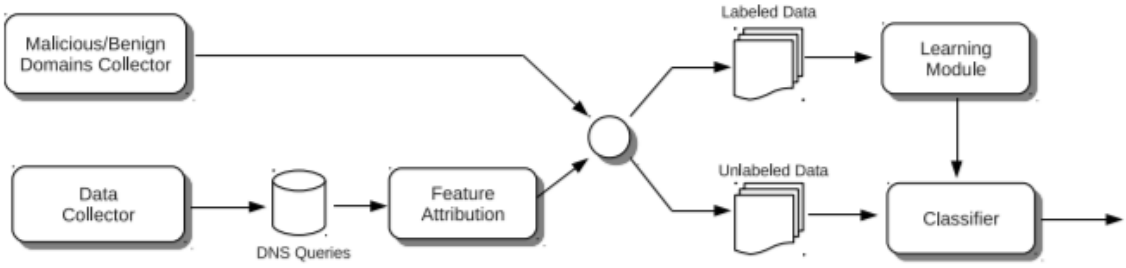

Figura 3.4: Funcionamento de Exposure[4]

Tabela 3.1: Características de Exposure [4]

Conjunto	**#**	**Nome da característica**
Tempo	*1*	Curta Longevidade
	2	Similaridade diária
	3	Padrões repetidos
	4	Rácio de acesso
Respostas DNS	*5*	Número de IPs diferentes
	6	Número de países diferentes
	7	Número de domínios com mesmo IP
	8	Resultados inversos de DNS
TTL	*9*	TTL médio
	10	Desvio padrão do TTL
	11	Diferentes TTL
	12	Mudança de TTL
	13	Percentagem do uso de TTL específicos
Nome do domínio	*14*	Rácio de caractéres numéricos
	15	Rácio do comprimento de LMS

3.5 BotDigger

BotDigger [5] foca-se no mesmo que esta dissertação, a deteção de DGAs numa rede. Com o foco neste objetivo, BotDigger utiliza várias características para alcançar com sucesso a deteção de DGAs: quantidade, temporais e linguísticas, sendo que o de quantidade baseia-se no facto de que o número de 2LDs consultados por *bots* é muito mais do que os *hosts* legítimos, o temporal divide-se em dois fatores: número de 2LDs consultados por um *bot* aumenta quando ele começa a tentar conectar-se com o domínio C&C registado e que quando ele consegue chegar ao domínio C&C o número de 2LDS consultados diminui muito, a última característica, linguística baseia-se na similaridade de atributos linguísticos.

Na Figura 3.5, podemos verificar o funcionamento do BotDigger, primeiro, filtros são aplicados para remover NXDomains não suspeitos sendo que os restantes são agrupados dependendo do *host*. Após esta filtragem, tendo um grupo de NXDomains agrupados por *hosts*, são verificadas as evidências anteriormente referidas: quantidade, temporal e linguística.

BotDigger foi avaliado usando domínios de Kraken e Conflicker sendo que é referido que este sistema deteta 100% de *bots* que são baseados no Kraken, mencionando apenas 0.05% de falsos positivos, e 99.8% dos *bots* de Conflicker e 0.39% de falsos positivos. Este é um dos problemas que quero resolver na minha dissertação, que é a diferença da eficácia da deteção em diferentes DGAs.

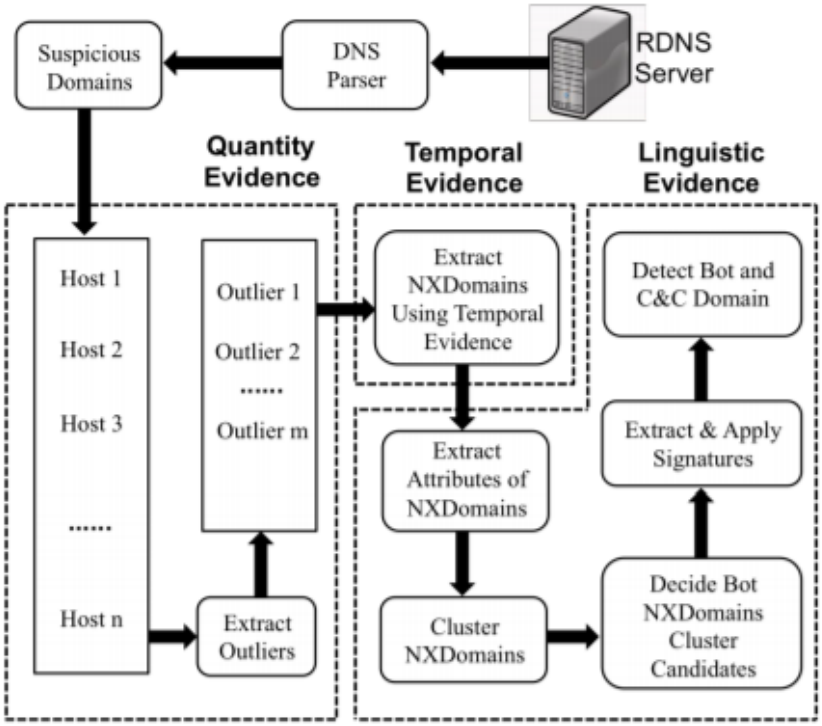

Figura 3.5: Funcionamento de BotDigger [5]

3.6 Phoenix

Phoenix é um sistema que separa domínios provenientes de DGAs de domínios legítimos. Este usa uma combinação de string e características baseados no endereço IP para conseguir caracterizar os DGAs por trás deles e encontram grupos de domínios gerados por DGAs que são representativos das respetivas *botnets*.

Como resultado, Phoenix pode associar domínios anteriormente desconhecidos gerados por DGA para grupos e produzir novos conhecimentos sobre o comportamento em evolução de cada *botnet* rastreado.

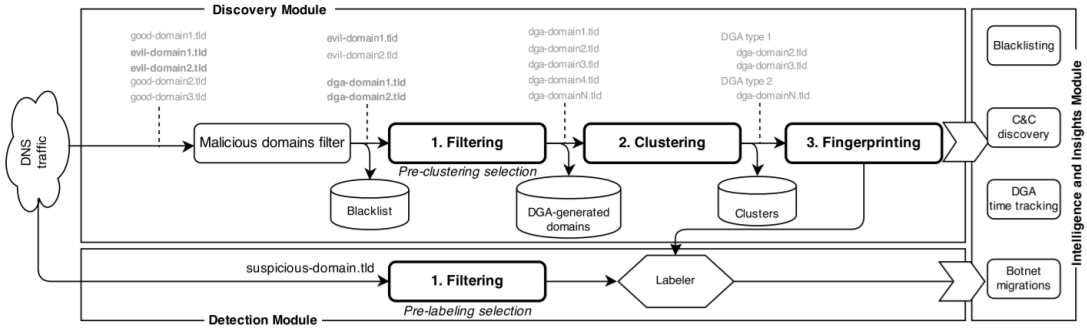

Figura 3.6: Funcionamento de Phoenix [6]

Phoenix foi avaliado em 1.153.516 domínios, incluindo domínios gerados por *botnets* modernos e bem conhecidos: sem supervisão, distinguia corretamente domínios gerados por DGAs e não DGAs com uma taxa de sucesso de 94.8% e, supostamente, ajudou investigadores a obter informação sobre domínios suspeitos [6].

3.7 Análise do artigo LSTM — Predicting Domain Generation Algorithms with Long Short-Term Memory Networks

Este artigo [18], em semelhança com este, analisa o funcionamento do uso de LSTM para conseguir detetar domínios de DGAs. Tem uma estrutura muito semelhante ao modelo desta dissertação:

- *Embedding*
- *LSTM*
- Classificador

Esta pesquisa foca-se maioritariamente no modelo binário, em que o resultado é DGA ou não-DGA. Para este modelo, foram utilizadas 30 diferentes famílias de DGAs emparelhadas e domínios do Alexa.

Para além da criação do modelo referido anteriormente, também realizaram uma experiência interessante que se baseia na tentativa de classificação de domínios de DGAs não usados no treino como DGAs. Foram obtidos resultados de 0.9942.

Também mencionam uma tentativa de treino de um modelo multi classe no entanto não mostraram resultados pois mencionaram que não tinham sido bons o suficiente para tal.

De uma forma a analisar os resultados também criaram vários esquemas em que mostram a distribuição de cada DGAs em relação aos seus caracteres.

3.8 Análise do artigo - A LSTM based framework for handling multiclass imbalance in DGA botnet detection

Este artigo ([19]) surgiu depois do artigo referido na secção anterior. Este afirma que as LSTM, embora fornecerem uma forma significativa de combater *botnets*, são propensas ao desequilíbrio na avaliação multi classe. Refere também que este desequilíbrio é mais significativo na deteção de domínios provenientes de DGAs, justificando este desequilíbrio na falta de suporte no conjunto de dados.

Assim, este artigo apresenta um novo algoritmo, LSTM.MI, que combina modelos de classificação binários e multi classe para a classificação de domínios vindos de DGAs. Este algoritmo utiliza LSTM original sensível ao custo. Os custos introduzidos no LSTM são introduzidos em *backpropagation learning procedure* ([20])de forma a conseguir ter em conta a importância dos custos na identificação das diferentes classes.

Para avaliar este novo algoritmo foram realizadas várias experiências em que referem que o novo algortimo LSTM fornece, pelo menos, uma melhoria de 7% em termos de *macro-averaging recall* e de precisão comparando ao artigo referido na secção 3.7.

3.9 Conclusão

A deteção automática de domínios de DGAs tem sido um tema abundante nos dias de hoje e pesquisa e trabalhos sobre este temas estão a aumentar cada vez mais.

Como mencionado neste capítulo, já houve várias abordagens a este tema, desde métodos de análise de características para além do nome do domínio a modelos treinados por domínios obtidos em tempo real([21]). A abordagem que foi feita nesta dissertação também foi mencionada na Secção 3.7, o uso de LSTM para a deteção de domínios. No entanto, ainda não tinha sido destacado a importância, desempenho de *seeds* diferentes do mesmo DGA e também uma análise de que domínios seriam mais corretos a usar no treino de modelos para este caso.

Vários outros documentos também analisam outros aspetos importantes como estratégias para deteção mais rápida([22] e [23]) a deteção de DGAs em redes *flast-flux*([24], [21]) e outras estudos que analisam a deteção de *botnets* que sejam independentes a nível de protocolo e estrutura([25]).

Capítulo 4

Metodologia

4.1 Ferramentas usadas

Para conseguir verificar a utilidade da aprendizagem computacional da deteção de domínios foi implementado código em Python3[26] com ajuda da biblioteca de *software tensorflow*[27] que possui funções da API Keras. Keras[28] contêm funções que permitem criar e treinar modelos de aprendizagem computacional e pela versatilidade que fornece em criar modelos diferentes necessários para o desenvolvimento do projeto. Também foi usado Docker[29] que permitiu correr o projeto em máquinas e ambientes diferentes para correr as experiências necessárias para avaliação do projeto sem preocupações de compatibilidades de *software*.

```
1 import tensorflow as tf
2 from tensorflow import keras
```

4.2 Conjunto de dados

4.2.1 DGAs

De modo a obter uma base para treinar o modelo ao longo das iterações e conseguir realizar as diferentes experiências explicadas no capítulo seguinte (Capítulo 5), foi obtido código para conseguir gerar diferentes DGAs que já foram avistados em diferentes alturas passadas e que, através de *reverse engineering* respetivo de cada DGA, permite a ter uns dados mais realistas.

Os DGAs escolhidos foram obtidos maioritariamente em https://github.com/baderj/domain_generation_algorithms que tem uma enorme variedade de diferentes DGAs, no entanto, como havia limitações a nível de poder computacional, inicialmente foram escolhidos cinco diferentes algoritmos como uma experiência inicial até que foram aumentados até quinze algoritmos finais de modo a ter um modelo variado e com mais conteúdo possível, sendo os mesmos

Tabela 4.1: *Seeds* usadas nos DGAs

DGA	***seed***
Banjori	*earnestnessbiophysicalohax.com*
Corebot	*1DBA8930*
Cryptolocker	*579*
Dircrypt	*1DBA8930*
Dnschanger	*579*
Fobber	*1DBA8930*
Kraken	*579*
Lockyv2	*579*
Murofet	*Data*
Necurs	*579*
Pykspa	*579*
Qakbot	*579*
Ramdo	*579*
Ramnit	*579*
Simda	*1DBA8930*

apresentados na Tabela 4.1 com as respetivas *seeds* usadas na implementação e desenvolvimento do modelo.

Quando esta base de dados foi obtida, permitiu ter um bom conjunto para passar ao treino do modelo, em que foram usados 10 mil de domínios de cada tipo de dados, ou seja, 10 mil domínios de cada DGA e outros tantos do Alexa.

De forma a conseguir usar os nomes de domínios foi usado um mapeamento de forma a ter os nomes como *arrays* com índices, cada índice refere-se aos caracteres alfanuméricos possíveis, sendo que estes são todos de letra minúscula.

4.2.2 Alexa

Para além dos domínios gerados através dos DGAs referidos 4.1, de modo a ter uma base de domínios benignos, foram utilizados um conjunto de domínios que foram nomeados pelo Alexa como o top 1 milhão de domínios utilizados.

Estes domínios nomeados pelo Alexa foram nomeados em 2007[30].

4.3 Modelo

Nesta secção é apresentado as características modelo usado no projeto, as diferentes camadas que o mesmo usa e como é que se realizou o treino para posteriormente conseguir prever e classificar domínios desconhecidos.

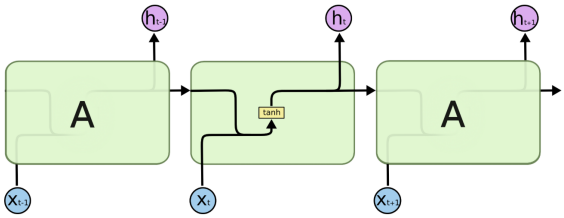

Figura 4.1: RNN[7]

4.3.1 Definição do Modelo

Com a ajuda das ferramentas e dados referidos das secções anteriores (Secção 4.1 e Secção 4.2) foram realizadas várias experiências de modo a ter uma análise completa da viabilidade de aprendizagem computacional na deteção de DGAs, na classificação de DGAs e diferenciação entre domínio maligno ou benigno.

Ao longo do projeto foram desenvolvidos dois modelos:

- Modelo binário(4.3.1.4)
- Modelo multi-classe(4.3.1.5)

Os modelos, em quase o seu todo, são semelhantes, ambos consistem em modelos *Sequential* que consiste numa pilha linear de diferentes camadas[31].

4.3.1.1 Embedding

A primeira camada, também comum a ambos, é a *Embedding*, esta tem como funcionalidade transformar os índices em vetores densos de tamanhos fixos que, para o desenvolvimento do projeto, foram usados maior parte vetores de tamanho 128 [32] .

4.3.1.2 LSTM

Para este trabalho foi implementada uma rede LSTM[7], que é a segunda camada em ambos modelos apresentados. Esta camada diferencia-se de outras, como RNN [33] (Figura 4.1, pois são capazes de aprender dependências de longo termo, sendo que não lhes é difícil armazenar informação durante longos períodos de tempo. LSTM também contêm um módulo repetitivo como as RNN, como se vê na Figura 4.1.

Como indica a Figura 4.2, o primeiro passo da LSTM é decidir que informação irá manter-se na iteração atual. Esta decisão é tomada por uma camada sigmoide chamada de camada de esquecimento que olha para a saída do módulo anterior de LSTM($h_t - 1$) e na entrada atual(Xt) e produz um número entre 0 e 1 para cada número no estado da célula($C_t - 1$), que é a memória do módulo anterior. Se o resultado for 1 quer dizer que a informação é totalmente mantida e se for 0 quer dizer que é totalmente esquecida.

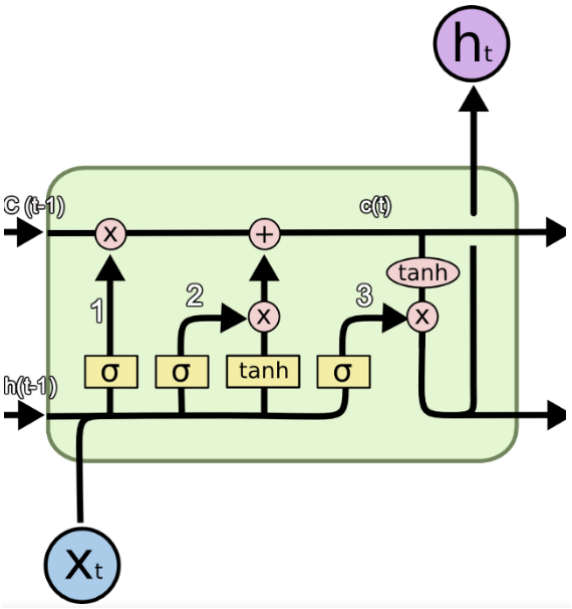

Figura 4.2: LSTM[7]

O próximo passo de LSTM é decidir que novas informações são armazenadas no estado da célula. Este passo divide-se em dois passos, primeiro, uma camada sigmoide chamada camada de entrada decide que valores serão atualizados. Em seguida, uma camada de tanh cria um vetor de novos valores candidatos que podem ser adicionados ao estado.

Na etapa seguinte, são combinados os dois dados resultados do passo anterior para criar uma atualização para o estado. Depois de ser atualizado o valor entre os dados resultantes é atualizado a célula de memória do LSTM tendo em conta os valores anteriores e novos gerados.

Por último, é decidido que resultado terá este módulo. Primeiro, uma camada sigmoide decide quais partes do estado da célula são produzidas. Depois o estado da célula é colocado em tanh de forma a meter os valores entre 1 e -1 e multiplica-se pelo resultado tirado anteriormente da célula de memória.

O resultado será:

$$h_t = tanh(C_t).(\sigma.(W_o[h_t - 1, x] + b_o))$$

4.3.1.3 Dropout

Por seguinte, temos a camada *Dropout*(Figura 4.3) que consiste em, aleatoriamente, remover um conjunto de arestas entre camadas durante cada iteração de treino, no entanto contribuem à mesma para os testes. *Dropout* consegues tornar o processo de treino "barulhento", forçando os nós dentro de uma camada a assumirem,probabilisticamente, mais ou menos responsabilidades pelas entradas [34]. Esta camada também oferece um método de regularização barato e eficaz para reduzir o *overfitting* e melhorar o erro de generalização.

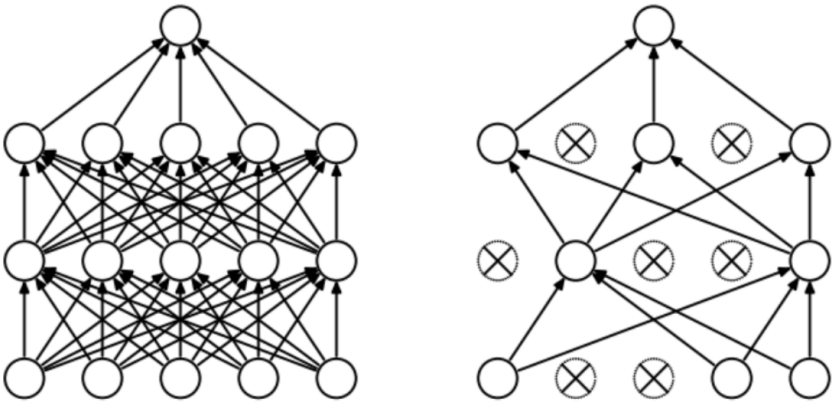

Figura 4.3: Efeito de *Dropout*

4.3.1.4 Modelo binário

O modelo binário foi implementado numa fase inicial do projeto. Este modelo inicial permite apenas diferenciar domínios malignos de não malignos.

Apesar deste modelo conseguir apenas rotular como maligno ou não, permite a utilização de mais que um DGA no treino do modelo sendo que se, por exemplo, usar 10 DGAs apenas usaria o número total de domínios/10 de cada DGA para o seu treino sendo que esses são rotulados como maligno e não como cada DGA diferenciado.

Para conseguir realizar este treino binário de maligno ou não maligno, foram utilizados *binary_crossentropy* e *activation sigmoid*.

```
model = Sequential()
model.add(Embedding(caracteres_alfanumericos, 128))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(activation='sigmoid', units=1,
                                kernel_initializer="uniform"
    ))
model.compile(loss='binary_crossentropy',
                        optimizer='adam', metrics=['accuracy'])
```

4.3.1.5 Modelo multi classes

O modelo multi classes foi o mais usado durante o desenvolvimento do meu projeto. Este permite a análise de vários domínios ao mesmo tempo com maior abundância e, o mais importante, para além de verificar se um domínio é maligno ou não, consegue também diferenciar a que família de DGAs, pertencentes no treino do modelo, faz parte.

Assim com esta habilidade de conseguir diferenciar de que família de DGAs o domínio vem, será mais fácil para depois um utilizador ao analisar os resultados consiga ter uma melhor perceção do que tipo de domínio é.

As principais alterações deste modelo com o modelo binário(4.3.1.4) foram o *activation softmax*, o *categorial_crossentropy* e o número de unidades que serão o número de rótulos diferentes existentes no modelo, que seria o número de diferentes DGAs analisados mais um, que seriam os domínios vindo do Alexa.

```
model = Sequential()
model.add(Embedding(caracteres_alfanumericos, 128))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(activation='softmax', units=
    dominios_diferentes,            kernel_initializer="
    uniform"))
model.compile(loss='categorical_crossentropy',
                    optimizer='adam', metrics=['accuracy'])
```

4.3.2 Treino do modelo

Para treinar o modelo final, os domínios referidos na Secção 4.2 foram divididos em 80% e 20%, respetivamente para treino e para teste. Para complementar o treino no modelo foram usados o *EarlyStopping* e o *ModelCheckpoint* que são duas *callbacks*[35] que interagem com o modelo de treino à medida que ele é executado.

```
dominios_treino, dominios_teste, rotulos_treino,
    rotulos_teste =             train_test_split(dominios,
    rotulos, test_size=0.2)
monitor = EarlyStopping(monitor='val_loss', patience=5,
    mode='min')
checkpointer = ModelCheckpoint(filepath='melhor.hdf5',
    monitor='val_acc', mode='max', save_best_only=True,
    verbose=1)
model.fit(dominios_treino, rotulos_treino,
                                    validation_data=(
    dominios_teste, rotulos_teste), callbacks=[monitor,
    checkpointer], verbose=2, epochs=25)
```

O *EarlyStopping* [36] permite especificar a medida de desempenho a ser monitorizada referindo uma condição que, quando ocorrer, o modelo será parado. No caso desta dissertação foi utilizado a medida *val_loss*. No modelo usado, o *EarlyStopping* atua no caso em que 5 *epochs* seguidos não diminuem mais o valor referido por *val_loss*. Quando este acontecimento surgir, o modelo irá ser parado a meio garantindo assim que não está a ser executado um número de vezes desnecessária e assim diminuindo o tempo de execução e aumentado a eficiência.

O *ModelCheckpoint* [37] também atua sobre uma medida, neste caso, *val_acc* em que o objetivo é maximizar o valor da mesma. Esta *callback*, ao contrário da anterior, não interrompe o treino do modelo a meio mas sim atualiza um ficheiro de extensão *hdf5* de modo a guardar apenas o peso do melhor modelo treinado ao longo do treino.

Capítulo 5

Avaliação

O desenvolvimento deste projeto pretendo explorar uma das lacunas, a nível de segurança informática, a automatização de deteção de domínios provenientes de DGAs, sendo que estas poderiam vir a ser beneficiadas através do uso da tecnologia. De forma a abordar este problema de deteção manual, este capítulo pretende avaliar vários aspetos que afetam o desempenho da solução.

5.1 Multiclasse

Nestas experiências foi usado o modelo multiclasse e LSTM(128), menos na secção 5.1.3 que analisa alterações do modelo e consequentemente alterações nos atributos de LSTM. Nesta secção o documento analisa os domínios do Alexa que devem ser utilizados para o treino do modelo, verifica que domínios serão os mais corretos de forma a melhorar a precisão do modelo, mantendo a diversidade de DGAs usados e observa o impacto de diferentes *seeds* na deteção da presença de mesmos DGAs usados no treino do modelo.

5.1.1 Análise do domínio Alexa

De maneira a perceber qual os melhores domínios, considerados benignos, a utilizar no treino do modelo, foram realizadas experiências em que em vez de serem usados apenas os 10 mil primeiros domínios fornecidos pelo Alexa, foram utilizados os 100 mil primeiros domínios do Alexa e dividiram-se em conjuntos de 10 mil de forma a averiguar se existe diferenças em treinar com domínios aleatórios do Alexa ou os primeiros 10 mil, por exemplo.

Como podemos verificar na Figura 5.1, obtiveram-se resultados bons para o desenvolvimento deste projeto pois nota-se que não existe uma diferença notável entre os domínios do Alexa. Nota-se apenas uma diferença maior na primeira coluna da tabela, que se refere aos primeiros 10 mil domínios, em que mostra que a maior parte dos domínios confundem-se como fazendo parte dos primeiros 10 mil, assim, decidiu-se utilizar os primeiros 10 mil domínios para realizar o treino do modelo.

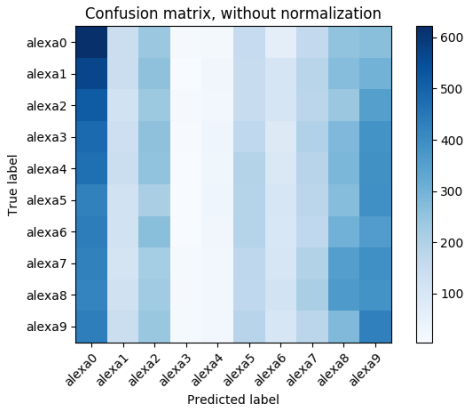

Figura 5.1: Resultados com diferentes domínios do Alexa.

5.1.2 Análise diferentes domínios

Em primeiro lugar, o modelo foi treinado com todas as famílias de DGAs separadas, sendo que cada família tinha 10 mil de domínios com uma *seed* específica, cada *seed* igual entre os domínios. Esta experiência foi feita para analisar que domínios se confundem entre si através do nosso modelo definido e de que modo se podia alterar os modelos de forma a ter melhores resultados.

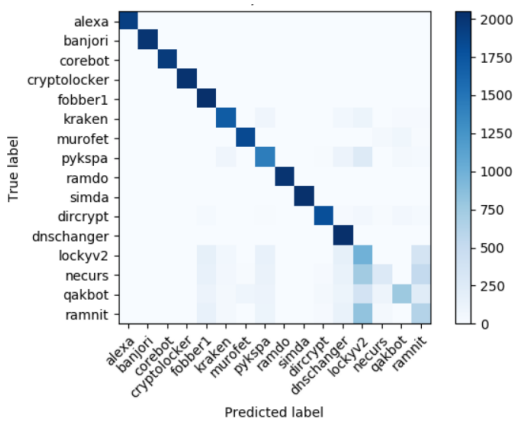

Figura 5.2: Resultados com vários DGAs.

Ao analisar a Imagem 5.2 dá para ver que os domínios considerados benignos do *alexa* conseguem-se distinguir dos outros DGAs todos. No entanto, algumas das famílias de DGAs não se conseguem distinguir uma das outras, sendo as piores a este nível as que estão indicadas em último na figura: *lockyv2*, *necurs*, *qakbot* e *ramnit*. Este modelo teve um valor de precisão de 0.4688, o que podemos ver que é muito baixo.

Devido a esta confusão entre famílias de DGAs foram retirados os domínios que tinham maus resultados e o modelo foi testado sendo que os resultados foram os mesmos, no entanto sem a

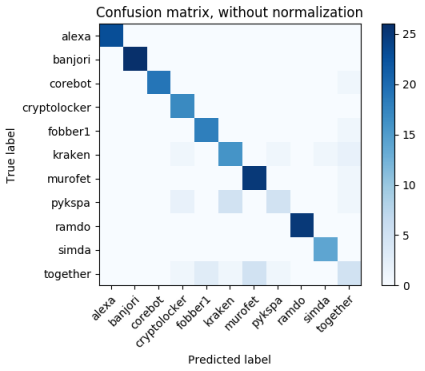

Figura 5.3: Resultados com DGAs juntos.

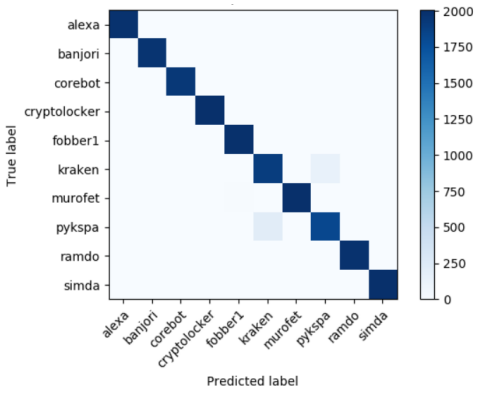

Figura 5.4: Resultados com alguns DGAs.

confusão dos domínios retirados(Figura 5.4).

Ao verificar-se esta impossibilidade de distinguir diferentes famílias de DGAs, primeiramente, o modelo foi treinado sem estas famílias que se confundiam entre si, no entanto, o modelo ia ser menos completo porque iria conter menos variedade de famílias de DGAs. Assim, foram juntados os vários domínios de cada família, numa só classe no modelo, em que em vez de ter 10 mil de domínios de cada família, tem 1/4 domínios de 10 mil de cada família.

5.1.3 Alteração do modelo

Para decidir qual o melhor modelo a usar e ter atenção também ao custo de computação, com os algoritmos que não se confundiam entre si, foram realizados diferentes testes nos que se mudava os parâmetros no modelo LSTM. Previamente, já se conhecia os resultados do modelo com LSTM 128, depois como se verificava valores de *val_acc* bons, verificou-se se seria necessário ter um valor tão alto, então foi testado com valores de 64 e 32. Os resultados de *LSTM(128)*, 0.97580, o que não mudou muito com o *LSTM(64)*, 0.97080, no entanto, com *LSTM(32)* o resultado final já foi mais baixo, 0.9375. Também houve tentativas de modificar o número de domínios por cada classe a ver se haveria muita diferença.

5.1.4 Seeds diferentes

Esta experiência foi realizada de modo a conseguir analisar o impacto das *seeds* no treino do modelo e posterior previsão dos domínios. Surgiu de uma questão que questionava a possibilidade da deteção de domínios vindos do mesmo DGA, no entanto com *seeds* diferentes, sendo que uma *seed* seria usada para o treino do modelo enquanto que outra para previsão.

Ao abordar esta questão, durante a realização deste trabalho, usei dois métodos:

- Criação de um modelo e treino do mesmo com *seeds* diferentes do mesmo DGA (5.1.4.1).
- Análise na previsão de domínios de um DGA com uma *seed* diferente que a usada no treino do respetivo modelo.(5.1.4.2).

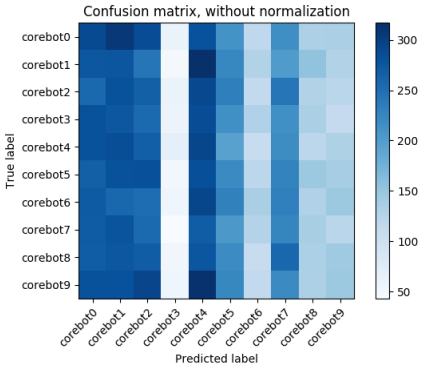

Figura 5.5: Modelo com várias *seeds* de Corebot.

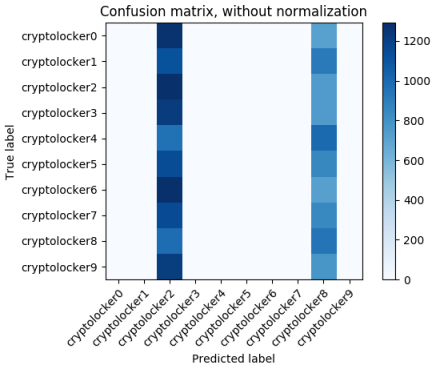

Figura 5.6: Modelo com várias *seeds* de Cryptolocker.

Estes métodos foram utilizados no modelo referido na Figura 5.4 e foram escolhidos um DGA para cada tipo de *seed*: inteira e hexadecimal, respetivamente Cryptolocker e Corebot.

5.1.4.1 Modelos criados através de *seeds* diferentes do mesmo DGA

Para a análise de *seeds* inteiras, experimentou-se usar *seeds* de 1 a 999 em intervalos de 100, ou seja, 100,200,300,400,500,600,700,800,900 e também foi usada a *seed* usada previamente no modelo:579.

Os resultados desta experiência, podem ser vistos na Figura 5.5, para os domínios gerados por *seeds* diferentes no algoritmo Corebot.Consegue-se ver que o modelo não consegue distinguir uns domínios de outros independentemente das *seeds*. Estes resultados foram uma amostra positiva para o trabalho, pois mostra que, embora os DGAs usados são usados constantemente com *seeds* diferentes, a *seed* não é um fator determinante por si.

Com estes resultados, é previsto que na experiência seguinte(5.1.4.2) seja possível detetar domínios de Corebot com *seed* diferente à usada durante o treino do modelo.

Para as *seeds* foi escolhido o algoritmo Cryptolocker. Para esta experiência, como não é fácil escolher de forma aleatória números hexadecimais, estes mesmo foram obtidos através de um site de geração aleatória.

Como podemos ver pela Figura 5.6 os resultados dividem-se bastante entre duas colunas, ou seja, existem duas *seeds* base que se destacam no meio das outras. Estes resultados não foram negativos também pois também mostra que a mudança de *seed* na geração dos domínios não é um fator determinante na geração. Independentemente de serem duas *seeds* apenas visíveis, pela Figura também é possível observar que estas mesmas *seeds* entre si "confundem-se"ou seja ao usar apenas uma das *seeds* no treino do modelo poderemos conseguir a vir detetar domínios gerados por Cryptolocker com outras *seeds*.

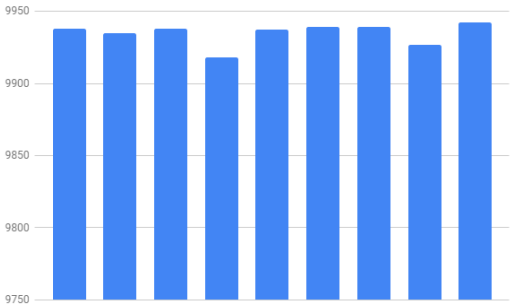

Figura 5.7: Previsão de *seeds* Corebot.

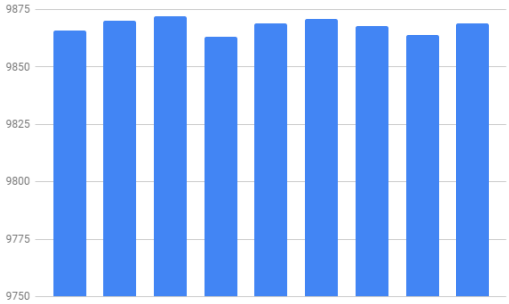

Figura 5.8: Previsão de *seeds* Cryptolocker.

5.1.4.2 Previsão de domínios do mesmo DGA com *seed* diferente

Nestas experiências foram usados os domínios descritos na secção anterior, domínios com *seeds* diferentes.

Com estes domínios já conhecidos, com o modelo definido para multiclasse e com o peso do modelo respetivo já obtido, foram executados *scripts* de forma a prever que DGA os domínios fazem parte conforme o modelo treinado anteriormente.

Os resultados obtidos dos domínios obtidos por Corebot tiveram uma taxa de sucesso de 99.35%. Na Figura 5.7 podemos ver quantos domínios foram previstos corretamente em 10000.

Para domínios gerados por Cryptolocker a Figura representa os resultados. Também tiveram uma taxa de sucesso muito elevada de 98.68%.A Figura 5.8, tal como a anterior, representa quantos domínios foram previstos corretamente como fazendo parte da família Cryptolocker pelo modelo.

5.2 Binário

Nas experiências do modelo binário foi usado como mencionado anteriormente *binary_crossentropy* e LSTM(128).

Foram feitas experiências deste o treino do modelo com os domínios do Alexa e cada um dos DGAs usados no modelo multiclasse e viu-se que, em todos deles, consegue-se verificar uma diferenciação entre os domínios do Alexa e o respetivo DGA.

Como os resultados eram perfeitos, juntou-se os vários DGAs num só grupo de dados de modo a analisar se com menos domínios de DGAs consegue-se, na mesma, ter bons resultados. Os resultados obtidos também foram perfeitos sendo que o modelo treinado consegue distinguir dados do Alexa dos DGAs usados. Esta experiência forneceu um aumento de variedade de DGAs no modelo binário.

Embora os resultados tenham sido elevados, o modelo binário não permite diferenciar os DGAs entre si ao contrário do modelo multiclasse e no modelo multiclasse também se verificaram resultados altos para a diferenciação dos domínios Alexa de outros DGAs.

Capítulo 6

Conclusões e Trabalho Futuro

6.1 Resumo do Trabalho Desenvolvido

De forma a atingir os objetivos deste trabalho na deteção automatizada de domínios provenientes de DGAs foi necessário:

- Analisar o historial de DGAs e estudar de que forma são implementados os algoritmos respetivos.
- Rever documentos e softwares que analisavam, de certa forma, temas parecidos com o assunto do meu trabalho.
- Procurar e estudar ferramentas a usar para ajudar na implementação do projeto e na utilização futura do mesmo.
- Preparar dados gerados por DGAs de forma a poder utiliza-los de forma organizada e estudada.
- Implementar e, posteriormente, treinar um modelo de forma a estudar a usabilidade e viabilidade da aprendizagem computacional na deteção automatizada de domínios vindos de DGAs.
- Realizar diferentes experiências de modo a avaliar o trabalho desenvolvido e desenvolver métodos de visualização dos resultados das mesmas.
- Analisar os resultados obtidos com as experiências feitas.

6.2 Principais Resultados e Conclusões

As experiências realizadas durante o processo de desenvolvimento do trabalho demonstraram que realmente é possível usar LSTM para conseguir detetar e classificar domínios que façam parte de famílias de DGAs.

Com a análise de diferentes domínios, foi verificado, no modelo multiclasse, que existem domínios de DGAs que confundem-se entre si, o que cria uma barreira na classificação de DGAs, no entanto não afeta a distinção de domínios de DGAs e domínios do Alexa. A separação de domínios Alexa com outros DGAs, tanto no modelo multiclasse tanto no binário, é feita com uma precisão bastante alta o que é uma mais valia para alertar analistas de segurança sobre a origem dos mesmos.

Ao ser verificada a confusão de domínios de DGAs entre si, foi feita uma experiência em que os resultados foram satisfatórios pois manteve-se uma valor de *accuracy* alto durante o treino do modelo e ao mesmo tempo, manteve-se a variedade dos DGAs. Assim, permite a que, analistas de segurança, saibam a que grupo pertence o DGA reduzindo assim as hipóteses.

Por último, no modelo de multiclasse, foi possível concluir que, mesmo com *seeds* diferentes, é possível detetar DGAs diferentes. Este foi um aspeto muito importante pois a previsão está ser feita tendo por base o nome dos domínios, no entanto, cada DGA tem uma *seed* para tentar fornecer uma aleatoriedade ao algoritmo o que poderia dificultar na previsão do DGA correto. No entanto, após as duas experiências feitas foi concluído que mesmo com *seeds* diferentes é possível fazer uma previsão acertada sobre o DGA que o domínio pertence e distinguir dos domínios benignos do Alexa.

6.3 Trabalho Futuro

Ao longo da realização deste trabalho, foram surgindo alguns aspetos que podem ser contribuições futuras para resultados melhores na deteção automática de domínios vindos de DGAs:

- Aumentar a diversidade e quantidade de domínios de DGAs.
- Adicionar no modelo a possibilidade de analisar e treinar vários níveis de nomes DNS.
- Melhorar o *script* de previsão de modo a mostrar de forma mais clara e intuitiva ao adicionar também a probabilidade resultante do modelo.
- Estudar de forma mais detalhada os domínios a usar que são considerados benignos.
- Otimizar o modelo de forma a aceitar mais domínios e a executar num tempo inferior.
- Analisar mais detalhadamente com resultados reais.
- Adicionar atributos de treino ao modelo de forma a ser mais completo e específico.

Para obter resultados mais reais é necessário ter acesso a uma rede que tivesse tráfego suficiente e variado para analisar os domínios e, posteriormente, prever se seriam derivados de um DGA mas não foi possível analisar uma rede de tal maneira vasta para ter resultados com DGAs mais recentes e esta experiência seria uma mais valia nesta análise.

6.4 Limitações

No final do projeto ao analisar os resultados foram levantadas algumas questões que podiam causar dúvida sobre realmente os resultados obtidos, como por exemplo, a impossibilidade de analisar todas as *seeds* possíveis, pois seria um número infinito de testes para cada DGA. Também existem mais DGAs conhecidos, no entanto, devido a uma questão de processamento não foi possível analisar muitos mais pois tal requereria que houvesse uma diminuição de números de domínios de cada DGAs o que poderia diminuir significa mente os resultados obtidos.

Este documento analisa a deteção de nomes de domínios gerados aleatoriamente, no entanto, podem ser atribuídos mais características para além do nome a domínios que poderiam a vir melhorar o modelo a nível de ser mais realista.

Referências

[1] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, e Nick Feamster. Building a dynamic reputation system for dns. Em *USENIX security symposium*, páginas 273–290, 2010.

[2] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou, e David Dagon. Detecting Malware Domains at the Upper DNS Hierarchy. Em *USENIX security symposium*, volume 11, páginas 1–16, 2011.

[3] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, e David Dagon. From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware. página 16.

[4] Leyla Bilge, Engin Kirda, Christopher Kruegel, e Marco Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. página 17.

[5] et al Zhang. BotDigger: Detecting DGA Bots in a Single Network. página 20.

[6] Stefano Schiavoni, Federico Maggi, Lorenzo Cavallaro, e Stefano Zanero. Phoenix: DGA-Based Botnet Tracking and Intelligence. Em Sven Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, páginas 192–211. Springer International Publishing, 2014.

[7] Understanding LSTM Networks. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[8] Cisco. Cisco Visual Networking Index: Forecast and Trends, 2017–2022, 2019.

[9] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, e Giovanni Vigna. Your botnet is my botnet: analysis of a botnet takeover. Em *Proceedings of the 16th ACM conference on Computer and communications security*, páginas 635–647. ACM, 2009.

[10] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, Geoff Hulten, e Ivan Osipkov. Spamming botnets: signatures and characteristics. *ACM SIGCOMM Computer Communication Review*, 38(4):171–182, 2008.

[11] Johannes Bader. The DGA of Banjori. URL: https://johannesbader.ch//2015/02/the-dga-of-banjori/.

[12] Norah Abokhodair, Daisy Yoo, e David W McDonald. Dissecting a social botnet: Growth, content and influence in twitter. Em *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, páginas 839–851. ACM, 2015.

[13] Ashley Hansberry, Allan Lasser, e Andrew Tarrh. Cryptolocker: 2013's Most Malicious Malware. página 5.

[14] Ben Herzog e Yaniv Balmas. Great crypto failures. 2016.

[15] Analysing a new eBanking Trojan called Fobber. URL: https://www.govcert.admin.ch/blog/12/analysing-a-new-ebanking-trojan-called-fobber.

[16] Paul Royal. On the Kraken and Bobax botnets. *Whitepaper, Damball, Apr*, 2008.

[17] Sandeep Yadav, Ashwath Kumar Krishna Reddy, A.L. Narasimha Reddy, e Supranamaya Ranjan. Detecting algorithmically generated malicious domain names. Em *Proceedings of the 10th annual conference on Internet measurement - IMC '10*, página 48, Melbourne, Australia, 2010. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1879141.1879148, doi:10.1145/1879141.1879148.

[18] Jonathan Woodbridge, Hyrum S. Anderson, Anjum Ahuja, e Daniel Grant. Predicting Domain Generation Algorithms with Long Short-Term Memory Networks. *CoRR*, abs/1611.00791, 2016. URL: http://arxiv.org/abs/1611.00791.

[19] Duc Tran, Hieu Mac, Van Tong, Hai Anh Tran, e Linh Giang Nguyen. A LSTM based framework for handling multiclass imbalance in DGA botnet detection. *Neurocomputing*, 275:2401–2413, Janeiro 2018. URL: https://linkinghub.elsevier.com/retrieve/pii/S0925231217317320, doi:10.1016/j.neucom.2017.11.018.

[20] Martin A. Riedmiller. Advanced supervised learning in multi-layer perceptrons — From backpropagation to adaptive learning algorithms. 1994. doi:10.1016/0920-5489(94)90017-5.

[21] Roberto Perdisci, Igino Corona, e Giorgio Giacinto. Early detection of malicious flux networks via large-scale passive DNS traffic analysis. *IEEE Transactions on Dependable and Secure Computing*, 9(5):714–726, 2012.

[22] Sandeep Yadav, Ashwath Kumar Krishna Reddy, AL Narasimha Reddy, e Supranamaya Ranjan. Detecting algorithmically generated domain-flux attacks with DNS traffic analysis. *IEEE/Acm Transactions on Networking*, 20(5):1663–1677, 2012.

[23] Sandeep Yadav e AL Narasimha Reddy. Winning with DNS failures: Strategies for faster botnet detection. Em *International Conference on Security and Privacy in Communication Systems*, páginas 446–459. Springer, 2011.

[24] Emanuele Passerini, Roberto Paleari, Lorenzo Martignoni, e Danilo Bruschi. Fluxor: Detecting and monitoring fast-flux service networks. Em *International conference on detection of intrusions and malware, and vulnerability assessment*, páginas 186–206. Springer, 2008.

[25] Guofei Gu, Roberto Perdisci, Junjie Zhang, e Wenke Lee. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. 2008.

[26] Welcome to Python.org. URL: https://www.python.org/.

[27] TensorFlow. URL: https://www.tensorflow.org/.

[28] Home - Keras Documentation. URL: https://keras.io/.

[29] Enterprise Container Platform. URL: https://www.docker.com/.

[30] Alexa - Top sites. URL: https://www.alexa.com/topsites.

[31] Recurrent Layers - Keras Documentation. URL: https://keras.io/layers/recurrent/.

[32] Ofir Press e Lior Wolf. Using the Output Embedding to Improve Language Models. *CoRR*, abs/1608.05859, 2016. URL: http://arxiv.org/abs/1608.05859.

[33] Antonio Gulli e Sujit Pal. *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[34] Yarin Gal e Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. Em D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, e R. Garnett, editores, *Advances in Neural Information Processing Systems 29*, páginas 1019–1027. Curran Associates, Inc., 2016. URL: http://papers.nips.cc/paper/6241-a-theoretically-grounded-application-of-dropout-in-recurrent-neural-ne
pdf.

[35] Callbacks - Keras Documentation. URL: https://keras.io/callbacks/.

[36] Jason Brownlee. How to Stop Training Deep Neural Networks At the Right Time Using Early Stopping, Dezembro 2018. URL: https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-sto

[37] Jason Brownlee. How to Check-Point Deep Learning Models in Keras, Junho 2016. URL: https://machinelearningmastery.com/check-point-deep-learning-models-keras/.

REFERÊNCIAS

Anexo A

Código

A.1 Modelo binário

```
1
2       label = 1
3       for malign in maligns:
4           with open(os.path.join(filepath), 'r') as csvfile:
5               readCSV = csv.reader(csvfile)
6               count = 0
7               for row in readCSV:
8                   if(count < int(num)/len(maligns)):
9                       malign_domains.append(row[0])
10                      label_malign.append(1)
11                      count=count+1
12                  else:
13                      label = label + 1
14                      break
15
16      with open(os.path.join(filepath), 'r') as csvfile:
17          readCSV = csv.reader(csvfile)
18          count = 0
19          multiple = 1
20          for row in readCSV:
21              if(count < int(num)):
22                  benign_domains.append(row[0])
23                  label_benign.append(0)
24                  count=count+1
```

```
25
26      for domain in malign_domains:
27          domains.append(tldextract.extract(domain).domain)
28      for domain in benign_domains:
29          domains.append(tldextract.extract(domain).domain)
30
31      labels = label_malign + label_benign
32
33      max_features = len(valid_chars) + 1
34      X = [[valid_chars[y] for y in x] for x in domains]
35      X = sequence.pad_sequences(X, maxlen=75)
36      label_np = np.array(labels)
37
38      model = Sequential()
39      model.add(Embedding(max_features, 128, input_length=75)
            )
40      model.add(LSTM(128))
41      model.add(Dropout(0.5))
42      model.add(Dense(activation='sigmoid', units=2,
            kernel_initializer="uniform"))
43      model.compile(loss='binary_crossentropy',
44                    optimizer='adam', metrics=['accuracy'])
45
46      model_json = model.to_json()
47      with open(os.path.join('binary.json'),'w') as json_file
            :
48          json_file.write(model_json)
49      X_train, X_test, y_train, y_test = train_test_split(X,
            label_np, test_size=0.2)
50
51      monitor = EarlyStopping(monitor='val_loss', patience=5,
             mode='min')
52      checkpointer = ModelCheckpoint(filepath='binary.hdf5',
            monitor='val_acc', mode='max', save_best_only=True,
            verbose=1)
53      model_labels = keras.utils.to_categorical(y_train,
            num_classes=2)
54      model_labels_test = keras.utils.to_categorical(y_test,
            num_classes=2)
```

```
55    model.fit(X_train, model_labels, validation_data=(
          X_test, model_labels_test), callbacks=[monitor,
          checkpointer], verbose=2, epochs=25)
```

A.2 Modelo multiclasse

```
1
2     label = 1
3     for malign in maligns:
4         with open(os.path.join(filepath), 'r') as csvfile:
5             readCSV = csv.reader(csvfile)
6             count = 0
7             for row in readCSV:
8                 if(count < int(num)):
9                     malign_domains.append(row[0])
10                    label_malign.append(label)
11                    count=count+1
12                else:
13                    label = label + 1
14                    break
15
16    with open(os.path.join(filepath), 'r') as csvfile:
17        readCSV = csv.reader(csvfile)
18        count = 0
19        multiple = 1
20        for row in readCSV:
21            if(count < int(num)):
22                benign_domains.append(row[0])
23                label_benign.append(0)
24                count=count+1
25            else: break
26
27    for domain in malign_domains:
28        domains.append(tldextract.extract(domain).domain)
29    for domain in benign_domains:
30        domains.append(tldextract.extract(domain).domain)
31
```

```
32      labels = label_malign + label_benign
33
34      max_features = len(valid_chars) + 1
35      X = [[valid_chars[y] for y in x] for x in domains]
36      X = sequence.pad_sequences(X, maxlen=75)
37      label_np = np.array(labels)
38
39      model = Sequential()
40      model.add(Embedding(max_features, 128, input_length=75)
            )
41      model.add(LSTM(128))
42      model.add(Dropout(0.5))
43      model.add(Dense(activation='softmax', units=len(maligns
            )+1, kernel_initializer="uniform"))
44      model.compile(loss='categorical_crossentropy',
45                    optimizer='adam', metrics=['accuracy'])
46
47      model_json = model.to_json()
48      with open(os.path.join(filepath),'w') as json_file:
49          json_file.write(model_json)
50      X_train, X_test, y_train, y_test = train_test_split(X,
            label_np, test_size=0.2)
51
52      monitor = EarlyStopping(monitor='val_loss', patience=5,
             mode='min')
53      checkpointer = ModelCheckpoint(filepath, monitor='
            val_acc', mode='max', save_best_only=True, verbose
            =1)
54      model_labels = keras.utils.to_categorical(y_train,
            num_classes=len(maligns)+1)
55      model_labels_test = keras.utils.to_categorical(y_test,
            num_classes=len(maligns)+1)
56      model.fit(X_train, model_labels, validation_data=(
            X_test, model_labels_test), callbacks=[monitor,
            checkpointer], verbose=2, epochs=25)
```

A.3 Previsão de domínios usando modelos guardados em ficheiros

```
    for domain in dgas:
        with open(os.path.join(filepath), 'r') as csv_file:
            readCSV = csv.reader(csv_file)
            for row in readCSV:
                temp_domains.append(row[0])
    count = 0
    for domain in temp_domains:
        domains.append(tldextract.extract(domain).domain)

    json_file = open('weights/best_few.json', 'r')
    model = json_file.read()
    json_file.close()
    loaded_model = model_from_json(model)
    loaded_model.load_weights(filepath)

    predictions = loaded_model.predict(domains)
    predictions = np.argmax(predictions, axis=1)

    with open('predictions.csv', 'w') as csv_file:
        for prediction in predictions:
            csv_file.write(str(count) + '-' + classes[
               prediction] + '\n')
            count = count + 1
```

A.4 Caractéres aceites pelos modelos

```
valid_chars = {
        'a' : 1,
        'b' : 2,
        'c' : 3,
        'd' : 4,
        'e' : 5,
        'f' : 6,
        'g' : 7,
        'h' : 8,
        'i' : 9,
```

```
11          'j' : 10,
12          'k' : 11,
13          'l' : 12,
14          'm' : 13,
15          'n' : 14,
16          'o' : 15,
17          'p' : 16,
18          'q' : 17,
19          'r' : 18,
20          's' : 19,
21          't' : 20,
22          'u' : 21,
23          'v' : 22,
24          'w' : 23,
25          'x' : 24,
26          'y' : 25,
27          'z' : 26,
28          '0' : 27,
29          '1' : 28,
30          '2' : 29,
31          '3' : 30,
32          '4' : 31,
33          '5' : 32,
34          '6' : 33,
35          '7' : 34,
36          '8' : 35,
37          '9' : 36
38      }
```

A.5 Criação de matriz de confusão

```
1   def plot_confusion_matrix(y_true, y_pred, classes,
        maligns,
2                             normalize=False,
3                             title=None,
4                             cmap=plt.cm.Blues):
5
```

```
6       if not title:
7           if normalize:
8               title = 'Normalized confusion matrix'
9           else:
10              title = 'Confusion matrix, without
                    normalization'
11
12      cm = confusion_matrix(y_true, y_pred)
13
14      classes = classes[unique_labels(y_true, y_pred)]
15
16      classes_str = [None] * len(maligns)
17      count = 0
18
19      for _ in maligns:
20          classes_str[count] = maligns[count]
21          count = count + 1
22
23      if normalize:
24          cm = cm.astype('float') / cm.sum(axis=1)[:, np.
                newaxis]
25          print("Normalized confusion matrix")
26      else:
27          print('Confusion matrix, without normalization')
28
29      print(cm)
30
31      fig, ax = plt.subplots()
32      im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
33      ax.figure.colorbar(im, ax=ax)
34      ax.set(xticks=np.arange(cm.shape[1]),
35             yticks=np.arange(cm.shape[0]),
36             xticklabels=classes_str, yticklabels=classes_str
                   ,
37             title=title,
38             ylabel='True label',
39             xlabel='Predicted label')
40
41      plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
```

```
42              rotation_mode="anchor")
43
44      fmt = '.2f' if normalize else 'd'
45      thresh = cm.max() / 2.
46      for i in range(cm.shape[0]):
47          for j in range(cm.shape[1]):
48              ax.text(j, i, '',
49                      ha="center", va="center",
50                      color="white" if cm[i, j] > thresh else
                            "black")
51      fig.tight_layout()
52      return ax
```