UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TÉCNICAS PARA IDENTIFICAÇÃO DE FUNÇÕES DE BIBLIOTECAS EM BINÁRIOS VINCULADOS ESTATICAMENTE

William Akihiro Alves Aisawa

Orientador: Prof. Dr. Paulo Matias

São Carlos – SP
Agosto/2020

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TÉCNICAS PARA IDENTIFICAÇÃO DE FUNÇÕES DE BIBLIOTECAS EM BINÁRIOS VINCULADOS ESTATICAMENTE

WILLIAM AKIHIRO ALVES AISAWA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Redes e Sistemas Distribuídos
Orientador: Prof. Dr. Paulo Matias

São Carlos – SP

Agosto/2020

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato William Akihiro Alves Aisawa, realizada em 21/08/2020.

Comissão Julgadora:

Prof. Dr. Paulo Matias (UFSCar)

Prof. Dr. Cesar Henrique Comin (UFSCar)

Prof. Dr. Paulo Licio de Geus (UNICAMP)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.
O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Agradecimentos

A Deus.

Aos meus pais, pelo apoio incondicional na busca de meus sonhos.

Ao meu orientador Paulo Matias, pelo empenho dedicado ao meu projeto de pesquisa.

Ao meu ex-orientador Professor Dr. Cesar Marcondes.

Aos meus amigos Ariane Gomes, André Landi, Laís Vilioni, Diego Pedroso e Eduardo Henrique (Mato) e a todos do laboratório de pesquisa Asgard, por toda a ajuda e apoio durante este período tão importante da minha formação acadêmica.

À VirusTotal por fornecer os dados utilizados neste projeto de pesquisa.

Ao CNPq pela bolsa concedida.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior Brasil (CAPES).

"Computer viruses are urban myths"

– Peter Norton, 1988

Resumo

A vinculação de bibliotecas estáticas pode tornar o trabalho de um analista de engenharia reversa desproporcional ao trabalho do desenvolvedor que criou o programa sob estudo, pois torna-se necessário examinar uma quantidade muito maior de código. Muitas vezes essa situação ocorre não como tática de ofuscação, mas sim como uma medida para distribuir o programa de forma mais simples. Como exemplo, uma grande proporção de códigos maliciosos (*malware*) para a plataforma Linux utiliza vinculação estática para evitar problemas de compatibilidade ao se propagar. Diversas ferramentas comumente utilizadas nos trabalhos de engenharia reversa, como IDA Pro, Ghidra, Radare2 e Binary Ninja possuem mecanismos para tentar reconhecer as funções dessas bibliotecas, com técnicas que variam desde o casamento de sequências de *bytes* até a avaliação de métricas em grafos de controle de fluxo. Trabalhos da literatura propõe alternativas que raramente são adotadas na prática, em parte devido à falta de uma metodologia compreensiva de avaliação. Além disso, as técnicas geralmente assumem que o binário será analisado com uma assinatura gerada a partir da mesma versão da biblioteca utilizada para compilá-lo, mas o problema de reconhecer essa versão é negligenciado. Também não existem estudos sobre o impacto da aplicação de assinaturas com versão divergente da versão utilizada na compilação original de um programa. Este trabalho estuda esses aspectos do reconhecimento de funções de bibliotecas vinculadas estaticamente por meio da aplicação de assinaturas geradas a partir de uma base com diversas versões da biblioteca padrão da linguagem C, além de propor uma técnica que permite reconhecer de forma rápida, em até 72% dos casos, a biblioteca padrão da linguagem C vinculada em binários para a plataforma Linux. Desta forma, o trabalho pretende contribuir para uma maior acurácia no reconhecimento de funções de bibliotecas vinculadas estaticamente.

Palavras-chave: Engenharia Reversa, Grafo de Fluxo de Controle, Análise Estática, Vinculação Estática

Abstract

Statically-linked libraries can cause the work of a reverse engineering analyst to get disproportionately hard compared to the work of the programmer who developed the software under study. This situation often arises not as an obfuscation tactic but as a measure to ease software distribution. For example, many malware programs designed for the Linux platform employ static linking to avoid compatibility problems when propagating to other systems. Many tools often used in reverse engineering practice, such as IDA Pro, Ghidra, Radare2, and Binary Ninja, have mechanisms that aim to recognize functions from these libraries, employing techniques that vary from byte sequence matching to the evaluation of control flow graph metrics. Works from the literature propose alternatives rarely adopted in practice, in part due to the lack of a comprehensive evaluation methodology. Besides, the techniques usually assume that the same version of the library used to compile a binary will be used to analyze it but neglect the issue of recognizing that version. There are also no studies about the impact of applying signatures with a different version than the one used to build the program. The present work studies these aspects on recognizing statically linked libraries by applying signatures generated from several distinct versions of the standard C language library and proposes a technique that allows fast recognition, up to 72% cases, of the version of the standard C library linked to Linux binary. This way, the work hopes to contribute to achieving better accuracy when recognizing statically linked library function.

Keywords: Reverse Engineering, Control-Flow Graph, Static Analysis, Static Linking

Lista de Figuras

2.1 Exemplos de declarações de comandos em CFG. Fonte: Adaptado de (ALLEN, 1970). 16

3.1 Visão geral do BinShape. Fonte: (SHIRANI; WANG; DEBBABI, 2017). 21

4.1 Exemplo de arquivo .exc gerado pelo sigmake. 29

4.2 Geração de assinaturas com o F.L.I.R.T 30

4.3 CFG da função dlclose. 35

4.4 Versões da libc entre lancamentos do GCC. 37

5.1 Tempo depreendido pelas ferramentas Radare2 e IDA Pro na análise inicial dos binários. Cada ponto corresponde a um dos binários. As retas correspondem a um ajuste linear desses pontos. 44

5.2 Tempo médio necessário para as ferramentas Radare2 e IDA Pro aplicarem uma das assinaturas. Cada ponto corresponde a um dos binários. As retas correspondem a um ajuste linear desses pontos. 45

5.3 Melhor F1 score obtido pelo IDA Pro para o reconhecimento de funções de cada um dos 459 binários que originalmente continham nomes de símbolos. 47

5.4 Melhor F1 score obtido pelo Radare2 para o reconhecimento de funções de cada um dos 459 binários que originalmente continham nomes de símbolos. 48

5.5 Melhor F1 score obtido pelo IDA Pro quando são considerados apenas binários para os quais a biblioteca padrão do C estava disponível na versão exata estimada via *changelogs*. 49

5.6 Melhor F1 score obtido pelo Radare2 quando são considerados apenas binários para os quais a biblioteca padrão do C estava disponível na versão exata estimada via *changelogs*. 50

5.7 F1 score obtido pelo IDA Pro com assinaturas geradas a partir de diferentes versões da biblioteca libc, em ordem decrescente, e ajustado a uma curva de decaimento exponencial. 52

5.8 F1 score obtido pelo Radare2 com assinaturas geradas a partir de diferentes versões da biblioteca libc, em ordem decrescente, e ajustado a uma curva de decaimento exponencial. 53

5.9 F1 score obtido pelo Radare2, isolando-se o mecanismo baseado em grafos, com assinaturas geradas a partir de diferentes versões da biblioteca libc, em ordem decrescente, e ajustado a uma curva de decaimento exponencial. 54

5.10 Histograma do desvio no F1 score obtido pelo IDA Pro com a escolha da assinatura que gerou o maior número de casamentos. 55

5.11 Melhor F1 score obtido com o IDA Pro usando assinaturas geradas a partir de bibliotecas de outra distribuição Linux. 56

5.12 Histograma da quantidade de dias transcorridos entre o lançamento da versão de libc estimada pela análise de *changelogs* e o lançamento da versão que gerou o máximo de casamentos no IDA Pro. 57

5.13 Quantidade de amostras da base analisada por ano em que se estima que elas foram compiladas. 59

Sumário

CAPÍTULO 1 – INTRODUÇÃO 12

1.1 Contexto e Motivação . . . 12

1.2 Justificativa . . . 13

1.3 Objetivos . . . 13

1.4 Hipóteses . . . 13

1.5 Organização do Trabalho . . . 14

CAPÍTULO 2 – REFERENCIAL TEÓRICO 15

2.1 *Malware* . . . 15

2.2 CFG - *Control Flow Graph* . . . 16

2.3 Aprendizado de Máquina . . . 17

2.3.1 Definição do problema . . . 18

CAPÍTULO 3 – TRABALHOS CORRELATOS 19

3.1 *Automatic library version identification, an exploration of techniques* . . . 19

3.2 *Function recognition in stripped binary of embedded devices* . . . 20

3.3 *Binshape: Scalable and robust binary libraryfunction identification using function shape* . . . 21

3.4 *Building a libc offset database* . . . 23

CAPÍTULO 4 – METODOLOGIA 24

4.1 Coleta . . . 24

4.2 Ferramentas 25
4.2.1 IDA Pro 25
4.2.2 Ghidra 25
4.2.3 Radare2 26
4.2.4 Binary Ninja 26
4.3 Seleção dos Binários 26
4.4 Estudo das Assinaturas 27
4.4.1 IDA F.L.I.R.T. 27
4.4.2 Ghidra Function ID 30
4.4.3 Rasign2 32
4.5 Escolha das ferramentas 36
4.6 Previsão de versão da biblioteca padrão 36
4.7 Execução do Experimento 38
4.7.1 Especificações 38
4.7.2 Análise com IDA Pro 38
4.7.3 Análise com Radare2 40
4.7.4 Análise dos resultados 40

CAPÍTULO 5 – RESULTADOS 43
5.1 Tempo de execução 43
5.2 Acurácia da identificação 46
5.3 Decaimento da acurácia com versões incorretas 51
5.4 Impacto de extrapolar para o conjunto completo 54
5.5 Acurácia com bibliotecas de outras distribuições 55
5.6 Avaliação da estimativa de versão 57
5.7 Linha do tempo 58

CAPÍTULO 6 – CONSIDERAÇÕES FINAIS 60

ANEXO A – CÓDIGO PARA CÁLCULO DO *F1-SCORE* 62

REFERÊNCIAS 63

Capítulo 1

Introdução

Este capítulo apresenta o contexto geral no qual o presente trabalho está inserido, bem como a motivação, contextualização, objetivo principal e a hipótese que deu origem ao projeto de pesquisa proposto.

1.1 Contexto e Motivação

A engenharia reversa é uma prática importante em diversas atividades profissionais. Algumas das aplicações são: análise de códigos maliciosos (*malware*) para entender como funcionam, como barrá-los e qual a extensão do comprometimento de uma máquina infectada com *malware*; busca de vulnerabilidades em *software* proprietário, com o objetivo de reportá-las, remediá-las ou explorá-las, incluindo vulnerabilidades em *firmware* de dispositivos embarcados; compreensão do funcionamento de formatos de arquivo ou protocolos para construção de *software* compatível; investigação de casos de violação de direitos autorais ou patentes; dentre muitas outras.

Uma das maiores dificuldades na desmontagem de programas escritos em linguagens de alto nível é analisar e isolar funções de bibliotecas padrão. Por ser uma tarefa totalmente manual, demanda um tempo que não agregará conhecimento profundo sobre tal programa, sendo apenas uma etapa obrigatória do processo que deve ser repetida a cada nova desmontagem (HEX-RAYS, 2020). Em função da necessidade de toda nova desmontagem ter de analisar as funções estáticas, essa acaba se tornando uma tarefa exaustiva e passível de erros, devido às mudanças que ocorrem no código durante o processo de compilação e variações de arquitetura de processadores (SHIRANI; WANG; DEBBABI, 2017).

1.2 Justificativa

As análises feitas pela Hex-Rays (2020) constataram que a média de funções oriundas de biblioteca padrão é de 50% em programas reais e, em alguns casos, como de um programa de "*hello world*", podem existir 58 funções de bibliotecas e apenas uma função *main()*. Estima-se, assim, que um analista utilizaria 50% do seu tempo para fazer o reconhecimento de funções de bibliotecas padrão.

Informações como nome das funções de bibliotecas são perdidas quando os símbolos de depuração são removidos do executável. Restaurar este tipo de informação ajuda otimizar a análise e interpretar o código do binário a ser analisado (JACOBSON; ROSENBLUM; MILLER, 2011).

No mercado existem ferramentas para suprir, de certa forma, a necessidade de reconhecer essas bibliotecas, como as que serão apresentadas na Seção 4.2, seja por meio de assinaturas contendo sequências de *bytes*, por análise de grafos, dentre outras técnicas. Entretanto, nenhum estudo aborda um método para o reconhecimento de versões exatas das bibliotecas, tampouco o impacto da aplicação de assinaturas com versões distintas da qual o programa tenha sido compilado originalmente.

1.3 Objetivos

O objetivo deste trabalho é propor métodos que possam contribuir para a acurácia da identificação de funções de bibliotecas em binários vinculados estaticamente.

1.4 Hipóteses

1. Utilizar a versão exata da biblioteca padrão do C (libc) gera melhor acurácia que simplesmente usar a última versão de cada *release* da distribuição Linux?

2. É possível prever a versão da biblioteca padrão do C sem ter de aplicar assinaturas geradas a partir de todas as versões possíveis?

3. Utilizar abordagens alternativas às baseadas em sequências de *bytes*, por exemplo métricas de grafos, tem potencial de gerar resultados melhores?

1.5 Organização do Trabalho

A presente dissertação está organizada da seguinte maneira: no Capítulo 2 são apresentados os conceitos teóricos básicos relacionados ao tema proposto. No Capítulo 3 é detalhado o atual estado da arte, com a seleção dos trabalhos correlatos encontrados. No Capítulo 4 é descrita toda a metodologia para que o desenvolvimento do projeto seja possível. No Capítulo 5 são apresentados os resultados obtidos. No Capítulo 6 são apresentadas as conclusões e trabalhos futuros. Por fim, são apresentadas as referências utilizadas para a escrita desta dissertação.

Capítulo 2

Referencial Teórico

Neste capítulo são apresentados os Fundamentos Teóricos para o desenvolvimento deste trabalho. A Seção 2.1 apresenta o conceito de malware e diferentes técnicas de análise. A Seção 2.2 apresenta o conceito de Control Flow Graph (CFG). A Seção 2.3 apresenta o conceito de Machine Learning.

2.1 *Malware*

Códigos maliciosos (*malwares*) são programas desenvolvidos especificamente para infiltrar-se em um sistema de forma ilícita, com o objetivo de causar dano, roubo ou alterações no sistema. Uma vez instalados, adquirem o mesmo nível de acesso a informações e ações sobre o sistema que o usuário que foi comprometido (CERT.BR, 2012).

Apesar de ser um ato recorrente, não é correto nomear todo e qualquer código malicioso como vírus. De acordo com McGraw e Morrisett (2000), os códigos maliciosos podem ser classificados dependendo de sua propagação. Abaixo, seguem exemplos de classificação.

- Vírus: são trechos de códigos maliciosos que se anexam em programas e ou documentos e se propagam quando o programa infectado é executado.
- *Worms*: são específicos para computadores em rede. Em vez de se acoplarem em programas, eles se propagam através da rede.
- *Trojan Horses*: mascaram-se como programas úteis, mas contém código malicioso para atacar o sistema ou vazar dados.

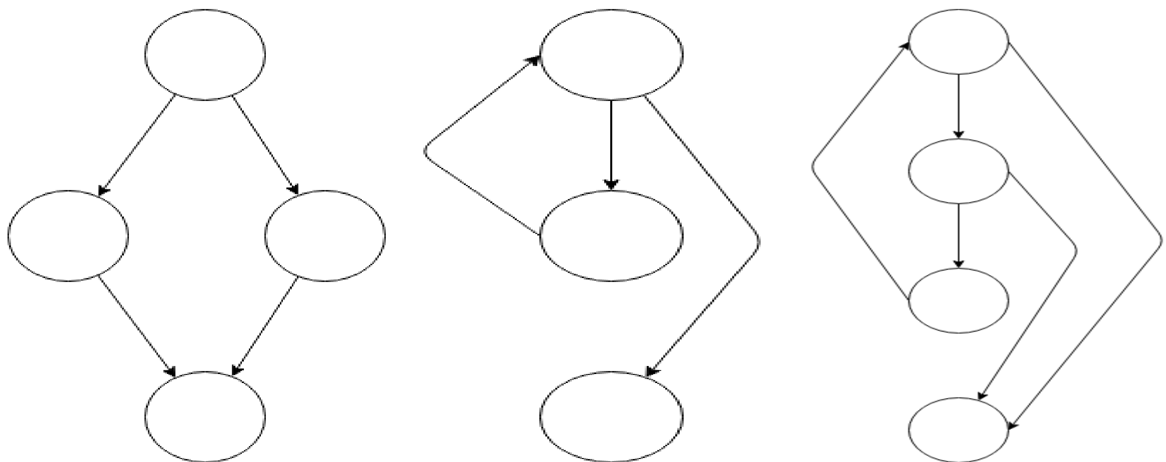
- ***Back Doors***: abrem o sistema para entidades externas, subvertendo as políticas de segurança locais para permitir o acesso remoto através de uma rede.

- ***Spyware***: é um pacote de software útil que também transmite dados privados do usuário para uma entidade externa através da rede.

A combinação de duas ou mais dessas categorias de códigos maliciosos pode levar a ferramentas de ataque poderosas. Por exemplo, um *worm* pode conter uma carga útil que instala um *backdoor* para permitir acesso remoto (CHRISTODORESCU; JHA, 2003).

2.2 CFG - *Control Flow Graph*

Control Flow Graphs (CFGs) são uma representação que utiliza notação de grafo para representar todos os caminhos que podem ser percorridos por um programa durante sua execução. CFG é um grafo direcional em que os nós representam os blocos básicos e as arestas representam o caminho do fluxo de controle (ALLEN, 1970).



(a) Declaração de um comando *IF*

(b) Declaração de um *while*

(c) Declaração de um *loop* com duas saídas

Figura 2.1: Exemplos de declarações de comandos em CFG. Fonte: Adaptado de (ALLEN, 1970).

Um bloco básico é uma sequência linear de instruções de um programa, tendo um ponto de entrada (primeira instrução executada) e um ponto de saída (última instrução executada). As Figuras 2.1(a), 2.1(b) e 2.1(c) representam grafos de comandos habitualmente utilizados na programação. Os blocos podem ter muitos predecessores e muitos sucessores, sendo que um bloco pode até mesmo ser seu próprio sucessor.

No entanto, os blocos de entrada do programa podem não ter predecessores internos ao programa, e os blocos de terminação de programa nunca têm sucessores no programa.

2.3 Aprendizado de Máquina

Aprendizado de Máquina refere-se a um conjunto de técnicas computacionais relacionadas ao aprendizado de funções matemáticas ou reconhecimento de padrões a partir de exemplos. Em seu livro, Simon (2013) define aprendizado de máquina como o campo de estudo que dá aos computadores a habilidade de aprender sem serem explicitamente programados.

Os diferentes sistemas de aprendizado de máquina possuem características que possibilitam sua classificação quanto à linguagem de descrição, modo, paradigma e forma de aprendizado utilizado (MONARD; BARANAUSKAS, 2003).

As tarefas de aprendizado de máquina são tipicamente classificadas em três categorias amplas, de acordo com a natureza do retorno de aprendizado disponível para um sistema de aprendizado. Em seu livro, Russell e Norvig (2016) definem essas três áreas da seguinte forma:

- **Aprendizado Supervisionado:** Exemplos de entradas e saídas desejadas são apresentadas ao computador, fornecidas por um "professor". O objetivo é aprender uma regra geral que mapeie as entradas para as saídas. Nesse tipo de aprendizado, existem duas tarefas — a de classificação e a de regressão.

- **Aprendizado Não Supervisionado:** Não é dado nenhum tipo de etiqueta ao algoritmo de aprendizado, deixando-o sozinho para encontrar estrutura nas entradas fornecidas. O aprendizado não supervisionado pode descobrir novos padrões nos dados ou um meio para atingir um fim. Esse tipo de aprendizado tem a tarefa de agrupamento.

- **Aprendizado Semi-Supervisionado:** Um programa de computador interage com um ambiente dinâmico, em que o programa deve desempenhar determinado objetivo (por exemplo, dirigir um veículo). É fornecida realimentação (*feedback*) ao programa quanto a premiações e punições, na medida em que é navegado o espaço do problema.

2.3.1 Definição do problema

O problema de reconhecimento de funções de bibliotecas vinculadas estaticamente geralmente é tratado como um problema de classificação supervisionada.

A classificação supervisionada faz uso de um conjunto de dados de treinamento com o intuito de aprender a relação de entrada e saída de uma amostra. Este conjunto consiste em exemplos de diferentes classes, onde a entrada da análise se dá por medidas ou parâmetros que diferenciam exemplos dessas classes e a saída se dá pelas classes pertencentes, ou seja, as que correspondem na classificação. A tarefa de aprendizado na classificação supervisionada é construir classificadores que consigam classificar previamente exemplos não conhecidos, ou seja, ela deve ser capaz de aprender com o conjunto de dados de treinamento e, assim, ser capaz de tomar decisões a partir do conhecimento adquirido (HUANG; KECMAN; KOPRIVA, 2006).

No entanto, as ferramentas tradicionalmente utilizadas não empregam algoritmos de classificação sofisticados. Como será apresentado na Seção 4.4, muitas ferramentas utilizam algoritmos de casamento de *strings* ou de comparação por meio de *hashes*. Uma exceção é a ferramenta Radare2, que calcula métricas em grafos e as compara com um algoritmo similar ao k-NN.

As técnicas propostas na literatura raramente são adotadas na prática. Um dos possíveis motivos desse problema é pela falta de uma metodologia compreensiva de avaliação, além disso as técnicas geralmente assumem que o binário será analisado com uma assinatura gerada a partir da mesma versão da biblioteca utilizada para compilação, não existindo estudos sobre o impacto da aplicação de assinaturas com versão de bibliotecas que difere na utilizada na compilação do binário.

Capítulo 3

Trabalhos Correlatos

Este capítulo apresenta diversos trabalhos relacionados que auxiliaram na idealização e desenvolvimento da ideia proposta.

3.1 *Automatic library version identification, an exploration of techniques*

Rinsma (2017) apresenta um trabalho que delineia e aplica comparações binárias nos problemas de identificação de versões de bibliotecas compartilhadas a partir de 6 diferentes técnicas de comparação implementadas:

- **`compare_bb_hash_bloomfilter (bloom)`**: Compara a amostra e a referência contrapondo as assinaturas *Bloom filter* de cada uma, usando o índice de Jaccard.
- **`compare_cc_list_levenshtein (cc1)`**: Compara os valores de complexidade ciclomática de todas as funções de amostra com as de referência utilizando a distância Levenshtein entre elas.
- **`compare_cc_list_set_union (cc2)`**: Realiza uma comparação de permutação independente entre as listas dos valores de complexidade ciclomática da amostra e da referência, tratando ambas como *sets* e calculando a sobreposição entre estes *sets* usando o índice de Jaccard.
- **`compare_cc_spp (cc3)`**: Compara os valores de complexidade ciclomática das funções de amostra e de referência fatorando os menores produtos primos de cada uma e determinando o valor de coincidência dos fatores.

- **`compare_strings_concat_levenshtein (str1)`**: Realiza uma comparação de *string* difusa e comparação de listas de *strings* utilizando a distância Levenshtein entre a concatenação da listas de *strings* na amostra e na referência.
- **`compare_strings_set_union (str2)`**: Executa uma comparação exata da permutação independente entre as listas de *strings* na amostra e na referência, tratando ambas como *sets* e calculando a sobreposição entre estes *sets* usando o índice de Jaccard.

A partir da análise destas técnicas, o trabalho evidenciou que a técnica denominada `str2` possui a melhor performance ao reconhecer a versão das bibliotecas utilizadas, visto que ao comparar um executável vinculado estaticamente com 188 referências, ela teve a classificação de similaridade mais alta, apesar de saber que o tipo de assinatura que ela usa para reconhecer uma biblioteca pode ser facilmente ofuscado (RINSMA, 2017). Apesar dos resultados positivos, é necessário testes com um *dataset* de binários reais, pois o experimento foi realizado apenas tentando reconhecer funções em arquivos de bibliotecas.

3.2 *Function recognition in stripped binary of embedded devices*

Já Yin et al. (2018) propõe o FRwithMBA, um algoritmo que faz o reconhecimento de funções baseado na informação de estrutura com *Maximum Branch Address* (MBA). Este algoritmo é capaz de determinar a posição final de uma função baseado em instruções de *branches* que aparecem no binário e no fim das assinaturas. Com um escaneamento linear, o processo de reconhecimento escaneia somente uma vez o programa, partindo da primeira instrução até o fim do segmento de código executável. Como primeiro passo, o FRwithMBA inicializa os parâmetros das variáveis que incluem: o conjunto das funções, MBA e *termination flag*. Começado o processo de reconhecimento de funções, as instruções são buscadas uma a uma a partir da sequência de instruções de entrada. Após isso, é determinado se o endereço da instrução atual está dentro do intervalo do *branch jump*, baseando-se no mapeamento virtual dos endereços. Se o endereço atual é maior que o MBA e a instrução anterior é uma função de instrução de término, o endereço é adicionado no conjunto de endereços das funções de início. Em seguida, a instrução é analisada para verificar se ela se encaixa em uma

das seguintes categorias: função *call*, *branch*, função de término e outras instruções. Para as funções *call*, o endereço a ser chamado é uma função e é adicionado diretamente à tabela de endereço de partida e ao *branch* com a flag de *function call*. Já para funções *branch*, o endereço de *branch* a ser chamado é comparado com o MBA, para obter o MBA atual. Por fim, quando surge uma instrução de função de término, a *termination flag* é marcada (YIN et al., 2018). Todo esse processo é repetido até que se termine a sequência de instruções de entrada. Os resultados foram positivos no reconhecimento de funções, porém o estudo limitou-se a trabalhar com as arquiteturas MIPS e PowerPC, não tendo sido analisadas outras arquiteturas.

3.3 *Binshape: Scalable and robust binary libraryfunction identification using function shape*

A ferramenta BinShape, proposta por (SHIRANI; WANG; DEBBABI, 2017), é um sistema escalável robusto para identificar funções de biblioteca padrão que gera assinaturas baseadas em análises sintáticas, semânticas, estrutural e estatística. A proposta é dividida em duas fases: Offline - onde é feita a extração e a seleção de características, que inclui o ranqueamento das características e então é gerada a assinatura - e a Online - onde é feita a extração de característica dos binários, a filtragem e a detecção, como mostrado na Figura 3.1.

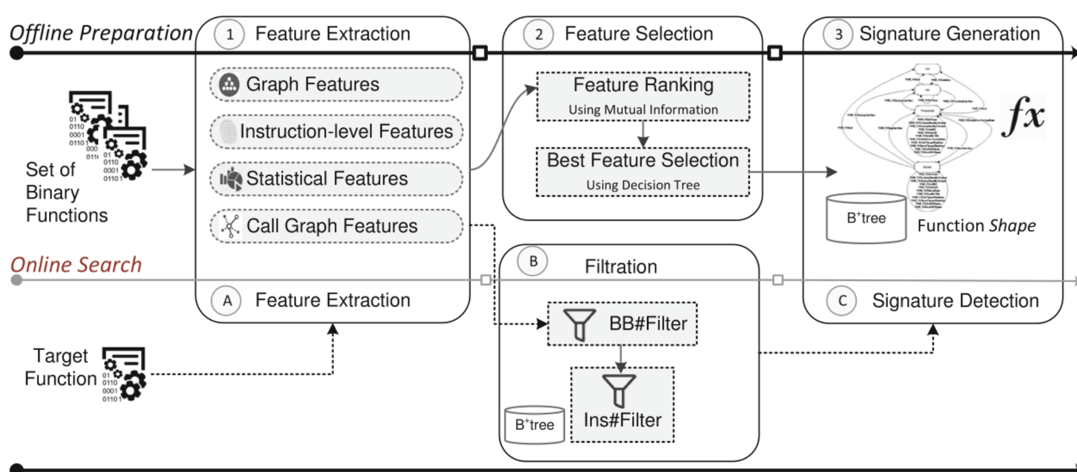


Figura 3.1: Visão geral do BinShape. Fonte: (SHIRANI; WANG; DEBBABI, 2017).

A extração de características da fase Offline ocorre em três etapas:

1. Extração das métricas de características do CFG de cada função da biblioteca, onde são extraídas métricas como quantidade de nós, quantidade de arestas, complexidade ciclomática e entre outras. Pode ocorrer de mais de dois grafos terem métricas semelhantes, para resolver a situação, as métricas que diferem são utilizadas para distinguir as funções.

2. Extração de características de nível de instrução, que é responsável pela extração de informações de sintaxe e semântica das funções desmontadas, como número de constantes, número de chamadas e informações.

   Possibilita-se agrupar essas funções por categorias que contém informações sobre as funcionalidades do programa. Essas categorias são utilizadas para enriquecer os CFGs.

3. Extração de características estáticas, utilizada para coletar as informações semânticas da função, como frequência de *opcodes*, que são utilizadas para detecção de binários metamorfos.

Já a seleção de características é realizada em duas etapas devido ao grande número de características extraídas no processo anterior, fazendo com que seja necessário selecionar as melhores.

1. Na seleção das melhores características foi utilizado um classificador de árvore de decisão nas características de melhor classificação, obtidas a partir do processo de classificação de características.

2. As assinaturas são armazenadas em uma estrutura denominada de Árvore B++, que basicamente indexa os melhores valores de recursos de todas as funções de biblioteca no repositório em árvores B+ separadas e vincula essas árvores B+ aos recursos e funções correspondentes.

A fase Online ocorre em três etapas: extração de características, filtragem e detecção. A extração ocorre de mesmo modo que na fase Offline, obedecendo as três etapas citadas anteriormente.

A filtragem ocorre para aumentar a escalabilidade, excluindo funções que provavelmente não terão correspondência. Para isso, são utilizados dois filtros:

1. Filtro de blocos básicos, em que são excluídas funções muito pequenas ou muito grandes, pois é pouco provável que uma função com 4 blocos básicos dê *match* com uma função de 100 blocos básicos.

2. Filtro de número de funções, calculado através da diferença entre a quantidade de instruções da função de entrada e a função armazenada no repositório. Se a diferença no número de instruções for menor do que um valor limite definido pelo usuário, então a função é considerada uma função candidata.

A detecção é realizada por meio da busca de possíveis *matches* na árvore B++ que contém a base com as assinaturas geradas. Os resultados mostram ser uma proposta eficiente, porém apresentou limitações quanto a ofuscação de código ou ofuscação no grafo, diminuindo a acurácia, além de realizar a análise somente de arquivos com arquitetura x86-64.

3.4 *Building a libc offset database*

Como alternativa para a identificação das funções de biblioteca padrão libc, Baumstark (2020) apresenta um *script* responsável por criar um banco de dados e fornecer diferentes métodos para a identificação e análise das funções, como buscas a partir do nome de uma função e o endereço que ela está armazenada, resposta do endereço de retorno da função `__libc_start_main` e obtenção dos endereços das funções (*offsets*), resultando em um ID da libc. Ademais, é possível descobrir se uma função já está no banco de dados ou a identificar a partir de uma *hash*, além do *script* também possibilitar o *download* de diferente versões da biblioteca, de acordo com o ID da libc correspondente à versão. No entanto, essa base é destinada à identificação de bibliotecas dinâmicas para facilitar o trabalho de exploração de vulnerabilidades do tipo *return-to-libc*, não sendo adequado à tarefa de engenharia reversa de binários vinculados estaticamente.

A partir das análises dos trabalhos mencionados acima, foi possível concluir que mesmo se tratando de problemas similares, nenhum deles propõe uma previsão de versão de biblioteca utilizada na compilação original do binário ou uma proposta que possa ser aplicada em diversas arquiteturas de processadores.

Capítulo 4

Metodologia

Neste capítulo são apresentadas as Metodologias utilizadas no trabalho. A Seção 4.1 apresenta o método utilizado para coleta de dados. A Seção 4.2 apresenta as ferramentas utilizadas na análise dos dados. A Seção 4.3 apresenta a metodologia empregada na seleção dos binários. A Seção 4.4 apresenta a forma com que foi realizada a geração de assinaturas. A Seção 4.5 apresenta as ferramentas escolhidas para o experimento. A Seção 4.6 apresenta o método de previsão de biblioteca padrão. A Seção 4.7 apresenta a execução dos experimentos.

4.1 Coleta

Na fase de coleta de exemplares de *malware*, foram procurados repositórios abertos como a VirusShare[1], porém foram encontradas apenas bases obsoletas e com poucos exemplares. Por isso, procurou-se contato com a VirusTotal[2], que forneceu uma base de dados com cerca de 400 gigabytes e aproximadamente 700 mil exemplares, além de disponibilizar livre acesso à sua API (*Application Programming Interface*) de consultas.

Além dos exemplares de *malware*, foram adicionados à base códigos não maliciosos, buscados em resoluções de problemas (*write-ups*) de competição no estilo CTF (*Capture The Flag*) como: DEF CON CTF[3], Google CTF[4], Pwn2Win CTF[5], dentre outras.

[1] <https://virusshare.com/>
[2] <https://www.virustotal.com>
[3] <https://www.defcon.org/html/links/dc-ctf.html>
[4] <https://capturetheflag.withgoogle.com>
[5] <https://pwn2.win>

Com o intuito de obter uma base robusta de versões das bibliotecas padrão da linguagem de programação C (*GNU C Library*) conhecidas como Glibc ou libc, foi realizado um levantamento das seguintes distribuições do sistema operacional Linux: Alpine[6], Debian[7], Ubuntu[8] e *Red Hat Enterprise Linux* (RHEL)[9]. Para obter os binários originais da Red Hat, foi realizada uma assinatura *trial* (30 dias) e construído um *script* robô em Python para automatizar o *download* dos pacotes.

Para este trabalho foram utilizadas diversas versões da libc por esta ser a biblioteca mais comumente vinculada a binários para o sistema operacional Linux (LINUX MANPAGES, 2020), no entanto este método pode ser estendido a outras bibliotecas.

4.2 Ferramentas

Para a análise dos binários, considerou-se utilizar as ferramentas a seguir, que possibilitam realizar descompilação, desmontagem (*disassembly*) e análise das assinaturas geradas.

4.2.1 IDA Pro

O IDA Pro[10] é um desmontador interativo que gera código *assembly* a partir da linguagem de máquina, tendo suporte a uma variedade de processadores e sistemas operacionais. Trata-se de uma das ferramentas de engenharia reversa mais conhecidas no mercado (AISAWA et al., 2019). Destaca-se também por ser uma das ferramentas mais antigas ainda em desenvolvimento no mercado, com registro de versões lançadas pelo menos a partir do ano de 1996 (ALEXA CRAWLS, 2020).

4.2.2 Ghidra

O Ghidra[11] é um software de código aberto para engenharia reversa desenvolvido na linguagem de programação Java, com suporte a uma variedade de processadores e disponível para os sistemas operacionais Windows 7 ou 10 (*64-bits*), Linux (*64-bits*)

[6] <https://alpinelinux.org>
[7] <https://www.debian.org/index.pt.html>
[8] <https://ubuntu.com>
[9] <https://www.redhat.com/pt-br>
[10] <https://www.hex-rays.com/products/idahome/>
[11] <https://www.nsa.gov/resources/everyone/ghidra>

e MacOS X. Lançado em março de 2019 pelo departamento de pesquisas da *National Security Agency* – NSA (AISAWA et al., 2019).

4.2.3 Radare2

O Radare2[12] é um projeto de código aberto composto por um conjunto de pequenos utilitários de linha de comando que podem ser usados juntos ou independentemente. O radare2, principal ferramenta do *framework*, é um desmontador que suporta uma variedade de máquinas virtuais e processadores.

4.2.4 Binary Ninja

O Binary Ninja[13] é uma plataforma de engenharia reversa que foca em uma *interface* de fácil uso, além de uma análise *multithread* construída em uma linguagem intermediária personalizada para se adaptar a diferentes tipos de arquiteturas, plataformas e compiladores (VECTOR 35, 2020).

4.3 Seleção dos Binários

O processo de extração inicia-se com a seleção dos binários para encontrar os arquivos com o formato ELF (*Executable and Linking Format* também chamado de *Extensible Linking Format*), utilizados em sistemas operacionais como: Linux, Solaris, entre outros e que possuam as bibliotecas vinculadas estaticamente, ou seja, códigos que tenham sido compilados estaticamente.

A busca foi realizada tanto nos diretórios de *malwares*, quanto nos de competições de CTF (*Capture-The-Flag*), utilizando um *grep* simples. No exemplo a seguir, foi feita uma busca no diretório `virusTotal_Database`, no qual estão todos os binários disponibilizados pela VirusTotal, para localizar *malwares* que sejam ELF e que tenham sido compilados estaticamente.

```
$ find virusTotal_Database | file * | grep -e "ELF" -e
  "statically linked"
```

[12] <https://rada.re/n/>
[13] <https://binary.ninja/>

4.4 Estudo das Assinaturas

4.4.1 IDA F.L.I.R.T.

O IDA F.L.I.R.T. (*Fast Library Identification and Recognition Technology*) é um algoritmo criado para o reconhecer funções de bibliotecas padrão em um programa (HEX-RAYS, 2020). Para isso, faz-se o uso de um banco de dados que armazena todas as funções de diversas bibliotecas. Ao analisar os *bytes* de um programa, o algoritmo é capaz de reconhecer se aquele é o início de uma das funções do banco de dados.

Cada função é representada por um padrão que considera os primeiros 32 *bytes* desta, em que todos os *bytes* que podem variar no processo de compilação são marcados. Entretanto, algumas funções compartilham a mesma sequência inicial de *bytes*. O número de 32 *bytes* foi selecionado na época em que o FLIRT foi projetado, pois forneceu os melhores resultados mantendo o tamanho da assinatura pequeno[14]. Para fins de otimização, o algoritmo faz uso de uma estrutura de dados chamada *trie*. Uma árvore *trie* é uma estrutura de árvore onde cada caminho, do nó-raiz até o nó-folha, representa uma *string* e cada nó desse caminho possui a identificação de um caractere desta *string* (FENG; WANG; LI, 2012). O uso desta estrutura de dados permite que os requisitos de memória sejam diminuídos, devido à armazenação não repetida de *bytes* em comum, além de agilizar o reconhecimento, pois o número de comparações a serem feitas torna-se menor em uma busca pela estrutura.

Não obstante, grande parte das funções podem ter os *bytes* iniciais semelhantes, o que acarreta no armazenamento de ambas em um único nó-folha. Para resolver essa situação, é calculado o CRC (*Cyclic Redundancy Check*) dos *bytes* começando da posição 33 até o primeiro *byte* variante. Em casos de *bytes* semelhantes e também de um mesmo CRC, são verificados os nomes de referência da instrução (HEX-RAYS, 2020).

Todas as informações requeridas pelo algoritmo de reconhecimento estão em um arquivo de assinatura específico para cada compilador ou versão de biblioteca. Um arquivo de inicialização automática de assinaturas também pode ser configurado para os casos em que o IDA Pro consiga detectar automaticamente o compilador. Assim que este é identificado, tem-se conhecimento de qual arquivo de assinatura deve ser utilizado em toda desmontagem. No entanto, esse processo só é implementado nativamente pelo IDA Pro para alguns compiladores comuns de ambientes MS-DOS e

[14]Comunicação pessoal de Ilfak Guilfanov, após contato com suporte técnico da Hex-Rays, em 19 de Outubro de 2020, recebida por correio eletrônico

Windows, por exemplo o Borland C. Não existe nenhum mecanismo automatizado para aplicação de assinaturas em ambientes Linux, problema que é muito mais complicado devido à grande variedade de versões.

Os padrões das funções não são armazenados em um arquivo de assinaturas no modo como estão originalmente contidos no binário. No lugar destes padrões, são armazenados os vetores de *bits* que determinam os *bytes* variantes e também os valores de cada *byte* individual. Assim sendo, o arquivo de assinatura não contém nenhum *byte* advindo da biblioteca original, exceto pelo nome das funções. O objetivo desse processo é evitar problemas de direitos autorais quando são geradas assinaturas de bibliotecas proprietárias.

De acordo com Hex-Rays (2020), a criação do arquivo de assinaturas é realizada em 2 passos: o pré-processamento das bibliotecas e a criação propriamente dita do arquivo. No caso de binários ELF, utiliza-se primeiramente a ferramenta `pelf` para pré-processar os arquivos `*.o` ou `*.a`, a fim de criar um arquivo contendo os padrões e o nome das funções, o CRC e todos os outros metadados necessários. Em um segundo passo, o programa `sigmake` constrói o arquivo de assinaturas a partir do arquivo de padrões. Essa divisão faz com que o `sigmake` possa ser independente do formato do arquivo original, fazendo com que outros pré-processadores possam ser escritos no futuro.

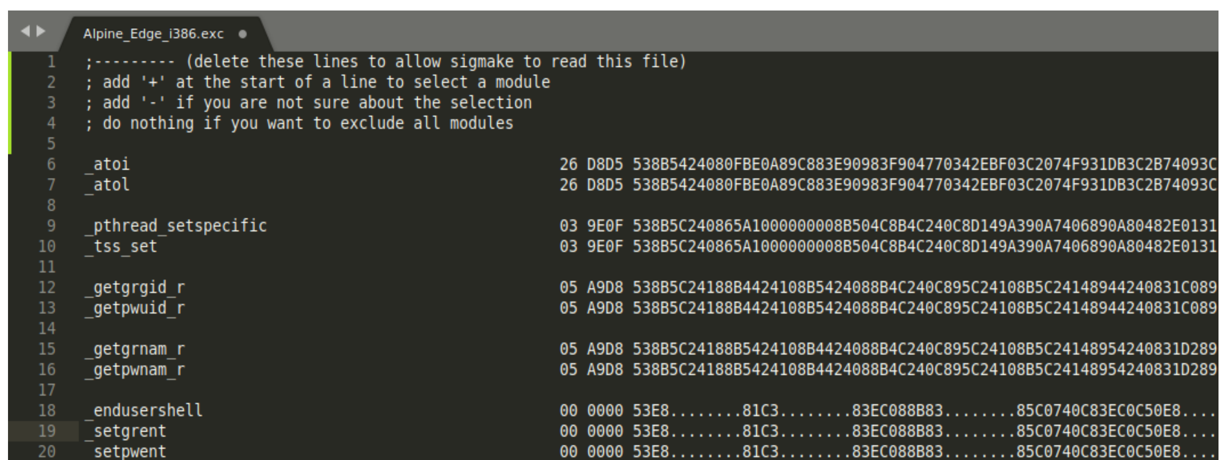
Para gerar a assinatura, os seguintes comandos foram utilizados:

```
$ ./pelf Alpine_Edge_i386.a
Diretorio_Libc/Alpine_Edge_i386.a: skipped 863, total 2639
$ ./sigmake -n Alpine_Edge_i386.pat Alpine_Edge_i386.sig
Alpine_Edge_i386.sig: modules/leaves: 1540/1769, COLLISIONS: 43
See the documentation to learn how to resolve collisions.
$ ./sigmake -s Alpine_Edge_i386.pat Alpine_Edge_i386.sig
```

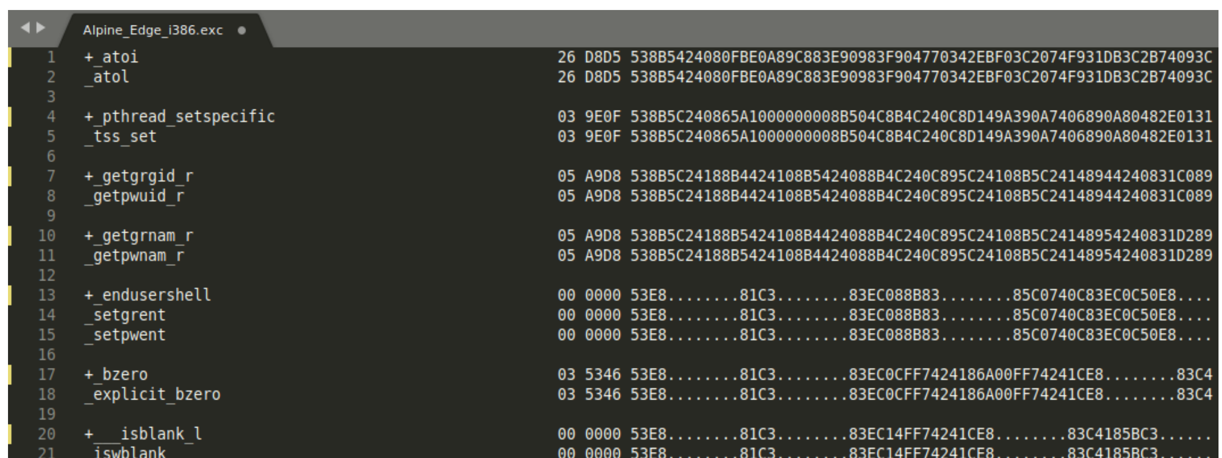
Ao executar o comando (`./pelf`) é gerado um arquivo .pat, que é utilizado para gerar a assinatura. O comando (`./sigmake`) com o parâmetro `-n` é utilizado para determinar o nome que irá aparecer no IDA quando uma assinatura é carregada. Sempre que o comando `sigmake` é usado para gerar uma nova assinatura, este gera um novo arquivo `.exc`, contendo as colisões encontradas durante o processo de geração da assinatura, como mostrado na Figura 4.1(a).

Para automatizar a geração de assinaturas foi desenvolvido um programa que busca o arquivo `libc.a`, permitindo que vários tipos de pacote de distribuições Linux podem ser processados, como `.deb`, `.rpm` ou `.apk`.

Para resolver as colisões encontradas durante o processo de geração de assinatura, um método importado do código `flair.py`, pertencente ao projeto ALLirt[15], foi utilizado para resolver as colisões, escolhendo a primeira opção como padrão, editando o arquivo `.exc` e executando o comando para gerar a assinatura novamente, como ilustra a Figura 4.1(b). O fluxo da geração de assinaturas utilizando o F.L.I.R.T é mostrado na Figura 4.2.



(a) Antes da edição



(b) Depois da edição

Figura 4.1: Exemplo de arquivo .exc gerado pelo sigmake.

15<https://github.com/push0ebp/ALLirt>

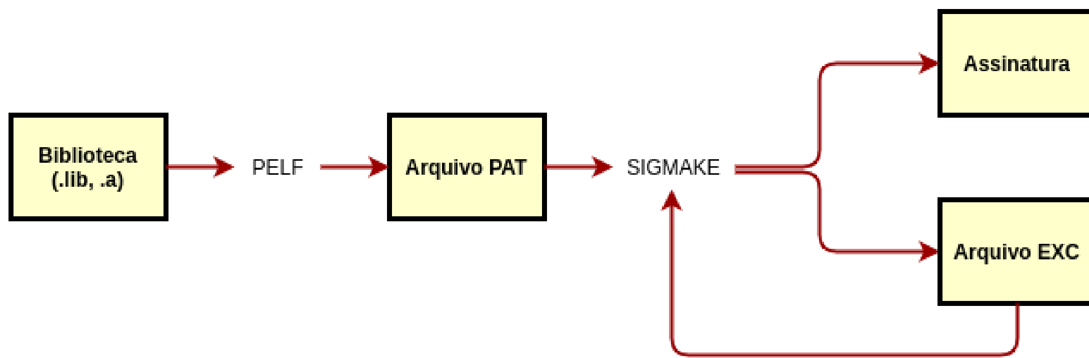


Figura 4.2: Geração de assinaturas com o F.L.I.R.T

4.4.2 Ghidra Function ID

O *Function ID* (FID) é um analisador responsável por gerar a análise e identificação da função em um programa. As Functions IDs são obtidas através de um processo de *hashing*.

Hashing

O *Function ID* é obtido através do processo do cálculo cumulativo de *hashs* do corpo de todas as funções. Para cada uma das funções são gerados dois tipos de *hashs* 64-bits, sendo que ambos são utilizadas para reconhecimento de futuros casamentos (*matches*):

- *Hash* Completa: Esta *hash* inclui o mnemônico e algumas das informações do modo de endereçamento de uma instrução. Operandos de registro específicos também são incluídos como parte da *hash*.

- *Hash* Especifica: Esta *hash* inclui tudo o que é usado na *hash* completa, mas também pode incluir os valores específicos de quaisquer operandos constantes. É empregada uma heurística que tenta determinar se a constante não faz parte de um endereço, caso em que o valor é acumulado na *hash*, o que acaba ajudando a distinguir variantes da função.

A *Hash* Completa torna-se robusta ao ser combinada com a específica, auxiliando o processo de desambiguação e tornando-o mais preciso (National Security Agency, 2020).

As *hashs* geradas são indexadas em um arquivo de armazenamento juntamente com os nomes das funções e outros metadados contidos na biblioteca em um arquivo próprio denominado de *Function ID Database*, com a extensão `.fidb`.

Single Matches e *Multiple Matches*

National Security Agency (2020) define como *Single Matches* os casamentos que obedecem às seguintes condições:

1. O FID pode reduzir as potenciais correspondências a um único nome de função.
2. A função ainda não possui um nome importado ou definido pelo usuário.
3. O número de instruções na função excede o limite de contagem de instruções.

Quando é identificado um *Single Match*, o FID irá aplicar o nome da função, inserir a descrição da biblioteca e adicionar um marcador para o FID. Tanto no comentário, quanto no marcador, será incluído a seguinte frase: "*Single Matches*".

Multiple Matches ocorrem quando o FID não é capaz de reduzir a apenas um único nome de função, mesmo aplicando toda a lógica de desambiguação, e então seu relatório irá depender das pontuações das incorrespondências restantes. Caso as pontuações sejam muito baixas, as correspondências serão consideradas aleatórias e nada será feito. Caso contrário, é encontrado um *Multiple Match*. Neste caso serão inseridos comentários em todas correspondências restantes e todos os comentários irão conter a seguinte frase: "*Multiple Match*".

Ao analisar uma função com o FID, funções pais e filhas são utilizadas para resolver ambiguidade em *Multiple Matches*. Suponha que duas funções são idênticas, exceto pelo fato de cada uma realizar a chamada para uma subfunção diferente. Nesse caso, os *hashes* da função serão idênticos. Então o sistema irá tentar realizar o *match* com uma das duas subfunções, permitindo assim a distinção entre as duas funções.

Pontuação e Desambiguação

O FID analisa e atribui pontuações a potenciais combinações. Esta pontuação é utilizada tanto para filtrar combinações pouco significantes, quanto para resolver ambiguidade entre potenciais combinações. De acordo com National Security Agency (2020): Uma instrução *Single Match* sem operadores constantes recebe pontuação 1,0. Instruções como `call` e *no operation* (`nop`) recebem pontuação 0. Já operandos constantes que casarem com um *hash* específico adicionam a pontuação de 0,67

por operando. Encontrado um casamento em potencial, atribui-se uma pontuação baseada nos seguintes critérios:

1. Instruções no corpo da função.
2. Operadores constantes que tenham *match* no corpo da função.
3. Instruções no corpo de qualquer função filha e também tenham um *match*.
4. Instruções no corpo de qualquer função pai e também tenham um *match*.

As funções com maior pontuação serão utilizadas, prevenindo assim casamentos aleatórios e melhorando a acurácia.

4.4.3 Rasign2

A ferramenta `rasign2` do projeto Radare2 permite a criação de arquivos do tipo SDB (*String-DataBase* ou *Simple-DataBase*)[16], assim, possibilitando a criação de um arquivo `libc.sdb` a partir de uma biblioteca `libc.so`, como no exemplo a seguir:

```
$ rasign2 -a -o Alpine_Edge_i386.sdb Alpine_Edge_i386.so
[x] Analyze all flags starting with sym. and entry0 (aa)
generated zignatures: 1611
```

A princípio, a prática de utilizar um arquivo de biblioteca dinâmica (`.so`) para gerar assinaturas destinadas a reconhecer funções de bibliotecas estáticas (`.a`) não é recomendada. No entanto, a documentação do Radare2 atualmente sugere esse método como uma forma simples de gerar as assinaturas, uma vez que o Radare2 não é capaz de interpretar arquivos (`.a`) diretamente: os diversos arquivos `.o` teriam de ser extraídos e analisados um a um. A prática de contornar esse processo utilizando o arquivo de biblioteca dinâmica talvez seja recomendada pelo fato de que o Radare2 atualmente é incapaz de utilizar as informações de relocação contidas nos arquivos objeto (`.o`) para gerar as assinaturas. Espera-se que, quando essa funcionalidade for implementada, essa prática deixe de ser recomendada.

A seguir está exemplificada a assinatura de uma função de forma detalhada:

[16]Comunicação pessoal de Pancake <https://github.com/radare>, fundador do Radare2, em 12 de Agosto de 2020, recebida pelo mensageiro instantâneo Telegram

```
$ rasign2 -r Alpine_Edge_i386.so | grep sym.dlclose
[x] Analyze all flags starting with sym. and entry0 (aa)
generated zignatures: 1611
za sym.dlclose b e9c21004005331c0658b0d00000000e88b59ffff81c35e7d07008a
512ef6c20274148b416883e2fd88512e83f8ff75068d83a5d4ffff5bc357658b3d00000
00056538b4768e85559ffff81c3287d07004883f8fd772b83ec0c8db3741d000056e8eb
4103008b47688b93701d000089108983701d0000893424e88a42030083c4105b5e5fc35
5575653e81159ffff81c3e47c070083ec188db3741d000056e8ad4103008b83701d0000
83c41085c0741383ec0c8b10508993701d0000e8794dffffebe083ec0c56658b3500000
000e8354203008b466883c41083f8ff740c83ec0c50e8534dffff83c410ff742424ff74
24246a006a00e82f0c030083c41083f8047305b80400000083ec0c8d680155e8d64cfff
f83c41089c785c07414ff742424ff7424245550e8fe0b030083c410eb0383cfff804e2e
02897e6883c40c5b5e5f5dc383ec0c8d442414525250ff74241ce82ffffff83c41cc35
7e8059dffff81c7127c070056538b5c24108b74241483ec0c53e84e0f040083c41085c0
752183fe02741450508d87e0d4ffff5650e8affffff83c410eb088b542418891aeb038
3c8ff5b5e5fc3ff3424ff74240cff74240ce8c310040083c40cc3ff3424ff74240cff74
240ce87112040083c40cc38b4004c38b400452658b1504000000518b088b400403048a5
9652b05000000005ac35653:ff0000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000
za sym.dlclose g cc=3 nbbs=8 edges=9 ebbs=1 bbsum=73
za sym.dlclose o 0x0001d226
za sym.dlclose x sym.dlclose
za sym.dlclose v s4
za sym.dlclose h
    fa3118103786d5cc98f23ca564e0e4b6cd68a6b554e807ad6250aff41776da19
```

A nomenclatura das assinaturas geradas para cada função reconhecida inicia-se com o prefixo *sym.* seguido do nome da função *dlclose*. As definições de cada item da saída estão descritas a seguir:

- `za sym.dlclose b`: É o padrão de *bytes* da função.
- `za sym.dlclose g`: São as métricas do grafo da função:
  - `cc`: complexidade ciclomática
  - `nbbs`: quantidade de blocos básicos
  - `edge`: quantidade de arestas
  - `ebbs`: blocos básicos finais (blocos básicos sem arestas de saída)
  - `bbsum`: soma do tamanho de todos os blocos básicos (em *bytes*)
- `za sym.dlclose o`: endereço de memória relativo ao início da biblioteca.
- `za sym.dlclose x`: Xref (*Cross-Reference*) utilizado para identificar o onde uma função ou objeto e chamado.
- `za sym.dlclose v`: variáveis.
- `za sym.dlclose h`: *hash* de todos os blocos básicos.

De acordo com Pancake, and Melchor, Álvaro Felipe (2020), o cálculo da complexidade ciclomática se dá por meio da seguinte fórmula:

$$CC = E - N + 2P$$

Em que E é o número de arestas do grafo, N é o número de nós e P o número de nós de saída.

O *hash* é gerado com o algoritmo SHA-256 (*Security Hash Algorithm*) aplicado sobre os *bytes* que compõe as instruções de todos os blocos básicos da função.

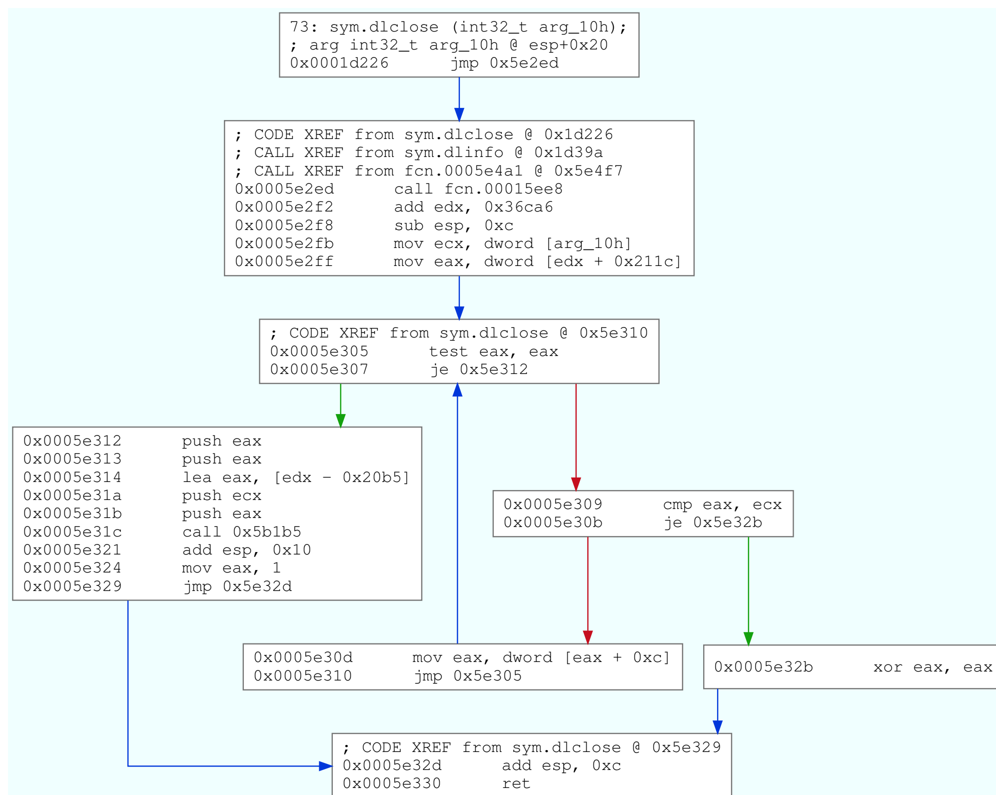


Figura 4.3: CFG da função dlclose.

Com os dados do grafo de controle de fluxo (CFG) da assinatura que foram gerados, pode-se exportar o grafo para o formato *.dot* e posteriormente renderizá-lo para uma imagem, como a exibida na Figura 4.3. Para gerar o arquivo `.dot` e exportá-lo para outros formatos foi utilizado o código a seguir:

```
$ r2 Alpine_Edge_i386.so #abre o arquivo a ser analisado
[0x0005ad37]> zo ./Alpine_Edge_i386.sdb #carrega a assinatura
[0x0005ad37]> aa #analisa o arquivo
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x0005ad37]> s sym.dlclose #procura o endereço de da função
    sys.dlclose
[0x0001d226]> agfd sym.dlclose > sym.dlclose.dot #exporta os
    dados do grafo para um arquivo .dot
[0x0001d226]> exit

$ dot -Tpdf sym.dlclose.dot -o sym.dlclose.pdf
```

O comando (`r2`) é responsável por carregar o arquivo no radare2 e, em seguida, é utilizado o comando (`zo`) para carregar a assinatura desejada. Logo após, é necessário analisar o arquivo para reconhecer as assinaturas utilizando o comando (`aa`) e, em sequência, o comando (`s`) é utilizado para procurar o endereço da função desejada podendo-se assim utilizar o comando (`agfd`) para exportar o grafo para um arquivo `.dot`.

Os casamentos das assinaturas usam, por padrão, todos os métodos: *bbhash*, *bytes*, *graph* e *offset*, mas eles podem ser desativados dependendo da necessidade do usuário. Por exemplo, o casamento com base no *offset* é dado por meio da distância em que se encontra a função com relação ao início do binário, tornando-o eficaz somente em binários gerados por *link*-editores mais antigos. Se a assinatura for gerada a partir de arquivos `.o` individuais, o `offset` não é inserido na saída.

4.5 Escolha das ferramentas

Uma vez estudados os mecanismos de geração de assinaturas, optou-se por escolher duas ferramentas como foco deste trabalho. O IDA Pro foi escolhido por ser a ferramenta mais madura, desenvolvida ativamente desde a década de 1990. O Radare2 foi escolhido por tornar disponíveis métricas de grafos para o casamento de assinaturas, mecanismo bastante distinto dos utilizados pelas demais ferramentas.

4.6 Previsão de versão da biblioteca padrão

Quando um binário é compilado, geralmente inclui-se nele uma *string* contendo a versão do compilador GCC que foi utilizada na compilação. Partindo do pressuposto que dificilmente um indivíduo vai atualizar parcialmente seu sistema (por exemplo, atualizar apenas o GCC, mas não atualizar a biblioteca padrão do C), a versão do GCC está correlacionada à versão da biblioteca padrão (libc) pela sua data de lançamento. Apesar desta ser uma ideia relativamente simples, desconhece-se qualquer trabalho publicado na literatura ou ferramenta de *software* que a tenha explorado até então.

Para explorar essa técnica, primeiramente foram coletados diversos arquivos `.diff.gz` e `.debian.tar.xz` de pacotes de código fonte do GCC e da GNU libc

do Debian e do Ubuntu, além de pacotes `.rpm` de binários do GCC e da GNU libc de diversas versões do RHEL.

Os arquivos `.diff.gz` ou `.debian.tar.xz` contém o diretório `debian` que contém instruções de compilação e metadados escritos pelos desenvolvedores do Debian ou Ubuntu. Dentro desse diretório, existe um arquivo denominado *changelog*, que contém diversas entradas com as seguintes informações, tanto a respeito da versão atual como de versões anteriores nas quais o pacote tenha sido baseado:

1. Versão do pacote.
2. Data em que a versão foi lançada.
3. Texto resumindo as novidades ou alterações daquela versão.

Os pacotes Red Hat também contém um *changelog*, que pode ser extraído diretamente do pacote precompilado com o comando `rpm -q --changelog -p`.

Para armazenar as informações dos *changelogs*, foi gerado um banco de dados SQLite[17] com uma tabela contendo seguintes informações: nome do pacote, versão, *timestamp* da data de lançamento da versão, a distribuição e a *release* (versão lançada da distribuição).

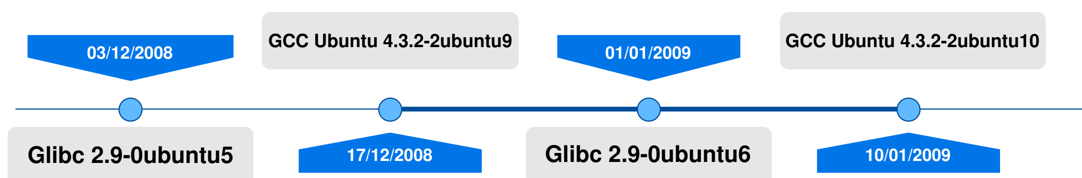


Figura 4.4: Versões da libc entre lancamentos do GCC.

Para determinar a versão de biblioteca padrão de um binário, é realizado o procedimento a seguir, que é ilustrado também na Figura 4.4:

1. Identifica-se a versão do GCC e o nome da distribuição Linux a partir do binário. Por exemplo, se existir a *string* `GCC: (Ubuntu 4.3.2-2ubuntu9) 4.3.3 20081217` no binário, sabemos que se trata da distribuição Ubuntu e da versão `4.3.2-2ubuntu9` do pacote `gcc-4.3`.

[17] <https://www.sqlite.org/index.html>

2. Procura-se no banco de dados o *release* da distribuição e a data de lançamento para essa versão de GCC. No caso do exemplo, *jaunty* e 17/12/2008, respectivamente.

3. Procura-se a versão do GCC imediatamente posterior a essa, com mesmo nome de pacote e no mesmo *release* da distribuição. No caso do exemplo, a versão `4.3.2-2ubuntu10`, lançada em 10/01/2009.

4. Procura-se e reporta-se: versão da libc com lançamento imediatamente anterior ao lançamento da primeira versão de GCC encontrada (no caso do exemplo, há a `2.9-0ubuntu5`, lançada em 03/12/2008); todas as versões de libc com lançamento entre a primeira e a segunda versão de GCC (no caso do exemplo, há apenas a `2.9.0-ubuntu6`, lançada em 10/01/2009).

Muitas vezes, esse procedimento não gera um resultado exato, como no próprio exemplo, que retornou como resultado duas versões possíveis de libc. No entanto, reduzir o conjunto de versões de libc a serem avaliadas pode reduzir drasticamente o tempo necessário para a análise, como será ilustrado na Seção 5.1.

4.7 Execução do Experimento

4.7.1 Especificações

Para o experimento foram utilizadas 3 instâncias do tipo `n2d-standard-8` da *Google Compute Engine*, serviço do tipo *Infrastructure as a Service* (IaaS), um dos pacotes oferecidos pela *Google Cloud Plataform*[18], possibilitando a utilização de máquinas virtuais definidas pelo usuário, o que permite escalonar as aplicações de acordo com as necessidades a um preço acessível. Cada instância desse tipo possui processador AMD EPYC 7B12 com 8 núcleos virtuais e 32GB de memória.

4.7.2 Análise com IDA Pro

Para aplicar as assinaturas do F.L.I.R.T nos binários utilizando o IDA Pro, o processo manual é abrir o binário com a ferramenta, aguardar a autoanálise terminar para poder aplicar a assinatura e coletar os resultados. Com o conhecimento do processo

[18]<https://cloud.google.com>

necessário para realizar a aplicação de uma assinatura, foi possível criar um *script* para o IDA Pro, responsável por automatizar essa tarefa.

O *script* foi escrito utilizando o IDAPython, um *plugin* do IDA Pro que integra a linguagem de programação Python, possibilitando o acesso ao *IDA plugin API*, IDC (*The Internal, C-Like Language*) *script* e todas as funcionalidades disponíveis no Python. Então essa etapa é repetida para todas as assinaturas disponíveis para o binário, de acordo com a arquitetura do processador utilizado na sua compilação. O tempo total consumido pela aplicação e análise de todas assinaturas também é armazenado em um arquivo de *log*.

O *script* segue as seguintes etapas:

1. Carrega o binário, aguarda a autoanálise realizada pelo IDA Pro e após isso tira uma *snapshot*. O tempo consumido pela autoanálise é armazenado em um arquivo de *log* gerado para o binário em questão.

2. A assinatura é aplicada ao binário e com isso foi possível armazenar as seguintes informações: quantidade de funções reconhecidas, além do nome e endereço de cada uma delas. Logo após esse processo, a *snapshot* inicial tirada no passo anterior é restaurada, para que não haja necessidade de carregar o binário novamente e aguardar a autoanálise. Então essa etapa é repetida para todas as assinaturas disponíveis para o binário, de acordo com a arquitetura do processador utilizado na sua compilação. O tempo total consumido pela aplicação e análise de todas assinaturas também é armazenado no arquivo de *log*.

Como o *script* do IDA analisa um único binário por vez, foi necessário a criação de um outro programa para automatizar as análises de todas as amostras de *malwares* e CTF coletadas. Para cada binário no diretório de amostras, esse programa obtém a arquitetura do processador e verifica se o binário foi compilado para 32 ou 64 *bits*. Essas informações são necessárias para para saber qual conjunto de assinaturas pode ser aplicado nesse binário.

Mesmo com essa automação, o processo levaria um tempo considerável para obter todos os resultados. Para otimizar o tempo, foi realizado um processo de paralelização com a biblioteca *multiprocessing* do Python, quebrando as chamadas do *script* escrito para o IDA Pro em subprocessos e reduzindo consideravelmente o tempo de análise.

Para que fosse possível executar esse experimento nos servidores da *Google Cloud Plataform*, foi utilizado o *xvfb-run*, que faz com que o experimento possa ser executado

sem interface gráfica, ou seja, o Xvfb executa todas as operações gráficas em uma saída de vídeo virtual, sem mostrar nada na tela.

4.7.3 Análise com Radare2

Para aplicar as assinaturas manualmente utilizando o radare2, o processo é semelhante ao IDA Pro: deve-se abrir o binário com o radare2, realizar a autoanálise e, então, aplicar a assinatura para coletar os resultados.

Como durante os testes foi constatado que seria demandado um tempo consideravelmente longo para o casamento de todos os binários com suas respectivas assinaturas da libc, foi realizada uma seleção de assinaturas para cada binário por amostragem estratificada com base nos casamentos obtidos com o IDA Pro. Para essa amostragem foram considerados as faixas de casamento de 100% a 90%, de 90% a 90% × 90%, de 90% × 90% a 90% × 90% × 90%, e assim por diante, até obter 30 amostras.

Com as amostras coletadas, foi possível a realização das análises utilizando um programa escrito em Python para automatizar as tarefas, seguindo as seguintes etapas:

1. Carrega o binário, executa a autoanálise e armazena o tempo consumido pela autoanálise no arquivo de *log* gerado para o binário.
2. Tenta remover alguma assinatura (se houver sido aplicada alguma) e então aplica uma nova assinatura. Então é realizada a busca por casamentos. Essa etapa é repetida para todas as assinaturas que haviam sido selecionadas por amostragem estratificada para o binário. O tempo total consumido pela aplicação e análise de todas assinaturas também é armazenado no arquivo de *log*.

4.7.4 Análise dos resultados

Para a avaliação dos dados coletados durante a análise dos binários, foi utilizada uma estimativa de acurácia denominada *F1 score*. Essa estimativa é muito utilizada na área Recuperação da Informação, e é adequada ao problema de classificação abordado por este trabalho pois ignora os verdadeiros negativos, que são muito comuns e camuflariam outras medidas de acurácia. O *F1 score* é calculado a partir de outras duas medidas, denominadas precisão e recuperação (POWERS, 2011)

Allen et al. (1955) definem os termos precisão e recuperação com base no conjunto de documentos recuperados e no conjunto de documentos relevantes. A precisão é calculada da seguinte forma:

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Já a recuperação é calculada como:

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

O *F1 score* é calculado como a média harmônica desses dois valores:

$$F_1 = \left(\frac{\text{recall}^{-1} + \text{precision}^{-1}}{2}\right)^{-1}$$

As variáveis utilizadas representam os seguintes valores:

1. `relevant_documents` = número de símbolos do binário original.
2. `retrieved_documents` = número de símbolos encontrados pelo IDA Pro ou radare2 (contabilizando os endereços com mais de um nome reconhecido).
3. `relevant_and_retrived_documents` = número de símbolos resultantes da interseção entre o binário original e os encontrados pelo IDA Pro ou radare2.

A função `fuzzy_match`, definida no código implementado para o *F1 score* disponível no Anexo A, resolve os seguintes problemas que eventualmente ocorrem no casamento entre nomes de símbolos:

1. Inserção de um *underline* seguido de um número no final do nome da função: geralmente ocorre quando a função é detectada em mais de um endereço do binário.
2. Inserção de um prefixo `libc_` ao nome da função. Ocorre em algumas versões de *link-editor*.
3. Inserção de um sufixo `_x86`, `_ia32` ou `_sse2`. Ocorre com algumas versões de biblioteca padrão para arquitetura Intel de 32 bits, quando o compilador escolhe uma versão otimizada da função, escrita em linguagem montadora.

4. Sinônimos comuns de símbolos: `_IO_getc` é sinônimo de `fgetc` e `_IO_putc` é sinônimo de `fputc`.

Em síntese, foi possível realizar a coleta de amostras de binários em uma vasta base de exemplares, considerando exemplares de *malwares* e de CTF. Também foram escolhidas as ferramentas utilizadas para gerar as assinaturas e analisar os binários previamente escolhidos. A partir das análises dos *changelogs* das bibliotecas padrão, foi desenvolvido um método de previsão de versão, como foi mostrado na Seção 4.6.

Capítulo 5

Resultados

Este capítulo primeiramente apresenta o tempo de execução das diferentes ferramentas utilizadas nos experimentos. Em seguida, mostram-se os resultados obtidos utilizando como referência os binários que originalmente possuíam símbolos de depuração e que, portanto, são tomados como verdade fundamental na primeira etapa dos experimentos. Nessa primeira etapa, avalia-se a acurácia de identificação de funções pelas ferramentas, além do decaimento dessa acurácia com o uso de versões de biblioteca padrão diferentes daquela que apresentou o melhor resultado. Em uma segunda etapa, avalia-se o impacto de tomar como referência o maior número de casamentos em vez da melhor acurácia, permitindo que os experimentos seguintes sejam estendidos a todo o conjunto de dados, incluindo amostras que não possuíam originalmente símbolos de depuração. Utilizando o conjunto de dados completo, avalia-se como o uso de assinaturas geradas a partir de bibliotecas de distribuições Linux diferentes impacta a acurácia. Em seguida, avalia-se como a versão estimada pela análise de changelogs varia com relação à versão com maior número de casamentos. Por fim, a data de lançamento das bibliotecas que geraram maior número de casamentos é utilizada para traçar uma linha do tempo das amostras coletadas.

5.1 Tempo de execução

Ao carregar o binário no IDA Pro ou Radare2, efetua-se uma análise inicial com o objetivo de encontrar funções que não estejam declaradas na tabela de símbolos, propagar referências cruzadas, dentre outras tarefas que tornam a desmontagem do binário mais legível. O tempo necessário para realizar essa análise nas amostras uti-

lizadas neste trabalho é apresentado na Figura 5.1. Os tempos foram medidos em instâncias do tipo `n2d-standard-8` da *Google Cloud Engine*.

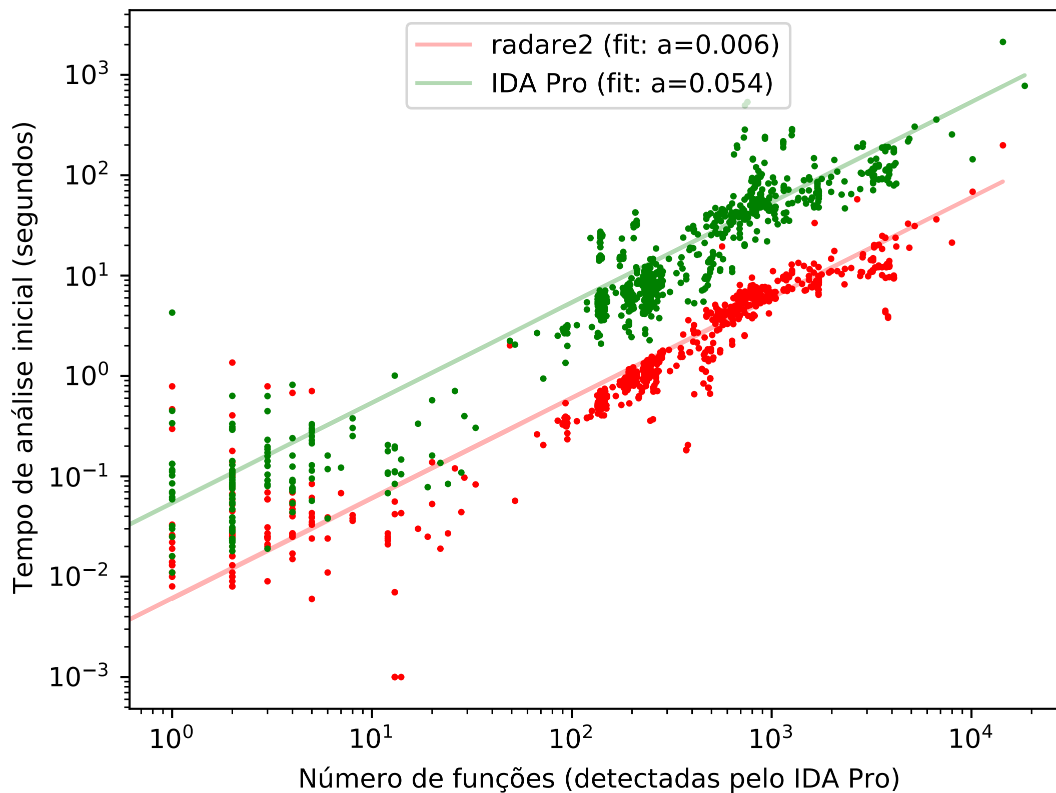


Figura 5.1: Tempo depreendido pelas ferramentas Radare2 e IDA Pro na análise inicial dos binários. Cada ponto corresponde a um dos binários. As retas correspondem a um ajuste linear desses pontos.

Devido ao fato de que a complexidade é uma característica do binário e não um dado extraído da ferramenta e definir uma complexidade está fora do escopo do projeto, para estimar a complexidade de cada binário foi utilizado o número de funções detectadas pelo IDA Pro, por ser o desmontador mais maduro e por fornecer uma *API* para facilitar a manipulação dos dados contidos no binário. Para que o eixo horizontal fosse consistente entre ambos os conjuntos de dados, esse número foi utilizado também para plotar o conjunto de dados do Radare2. Cabe esclarecer que se trata do número total de funções do binário, mesmo aquelas cujo nome é desconhecido. Esse valor não tem relação, portanto, com a aplicação de assinaturas e reconhecimento de funções de bibliotecas, que será realizada apenas posteriormente.

Foram ajustadas retas do tipo $f(x) = ax$ aos pontos medidos experimentalmente. Observando o coeficiente angular a dessas retas, pode-se notar que a análise inicial do Radare2 é cerca de 9 vezes mais rápida que a do IDA Pro.

Em seguida, mediu-se o tempo necessário para aplicar um grande conjunto de assinaturas sobre cada um dos binários. O tempo foi dividido pelo número de assinaturas, obtendo-se assim o tempo médio para aplicação de uma única assinatura. Os dados foram inseridos em um gráfico similar ao anterior, como mostrado na Figura 5.2.

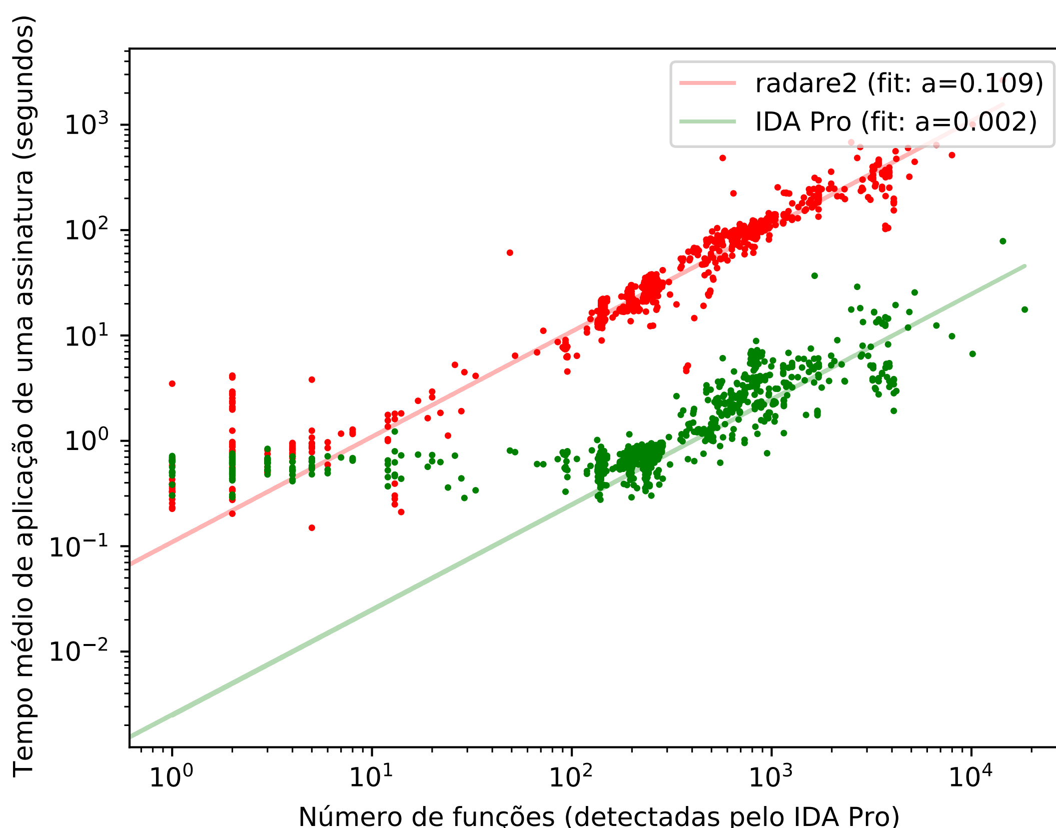


Figura 5.2: Tempo médio necessário para as ferramentas Radare2 e IDA Pro aplicarem uma das assinaturas. Cada ponto corresponde a um dos binários. As retas correspondem a um ajuste linear desses pontos.

Pode-se perceber que, apesar da análise inicial do Radare2 ser mais rápida que a do IDA Pro, o tempo necessário para aplicar uma assinatura é significativamente mais lento: em média 50 vezes mais lento que no IDA Pro. Esse resultado deve-se ao fato de que o Radare2 não utiliza uma estrutura de dados do tipo Trie para procurar pelas sequências de *bytes* presentes nas assinaturas. Além disso, os desenvolvedores

ainda não focaram[1] em otimizar o código em geral responsável pelo tratamento de assinaturas, incluindo o código dos mecanismos baseados em *hash* e em grafos.

Esse resultado reflete um grande problema que existe quando um analista precisa realizar engenharia reversa em um binário com uma versão desconhecida de biblioteca vinculada estaticamente. O tempo de análise inicial não é grande problema pois, em ambas as ferramentas, a análise inicial pode ser reaproveitada entre diversas tentativas de aplicação de assinatura. No entanto, o tempo elevado necessário para aplicação de uma assinatura no Radare2 torna proibitivo testar uma grande quantidade de assinaturas, tais como as mais de 400 assinaturas da base construída para o presente projeto.

Por esse motivo, todas as análises com o Radare2 apresentadas neste trabalho foram realizadas com uma amostra de apenas 30 assinaturas escolhidas aleatoriamente (de forma estratificada na quantidade de casamentos obtidos previamente pelo IDA Pro) para cada binário a ser analisado.

Pela analise dos gráficos da Figura 5.1 e Figura 5.2, nota-se que o tempo de reconhecimento de funções e aplicação das assinaturas não varia muito quando os binários apresentam uma quantidade de funções inferior a 100.

5.2 Acurácia da identificação

Para estimar a acurácia de um classificador, convém comparar os seus resultados a valores de referência conhecidamente verdadeiros, também chamados de "verdade fundamental". Dentre os 913 binários analisados, 459 estavam com os nomes de símbolos intactos, indicando que nunca haviam sofrido a ação da ferramenta `strip`. Foram, portanto, empregados como verdade fundamental para o cálculo dos índices de acurácia.

As Figuras 5.3 e 5.4 apresentam, respectivamente, o melhor *F1 score* obtido dentre todas as assinaturas aplicadas em cada um dos binários para os quais a verdade fundamental estava disponível. Nos gráficos, os binários estão categorizados pela arquitetura de conjunto de instruções do processador para o qual foram desenvolvidos. Tratam-se de 180 binários para arquitetura Intel de 32 bits (i386), 59 para Intel de 64 bits (amd64), 90 para MIPS, 101 para ARM e 29 para PowerPC. Em todos os gráficos de

[1] Comunicação pessoal de Pancake <https://github.com/radare>, fundador do Radare2, em 17 de Agosto de 2020, recebida pelo mensageiro instantâneo Telegram

box plot, o triângulo verde representa a média, e a linha horizontal laranja representa a mediana do conjunto de dados.

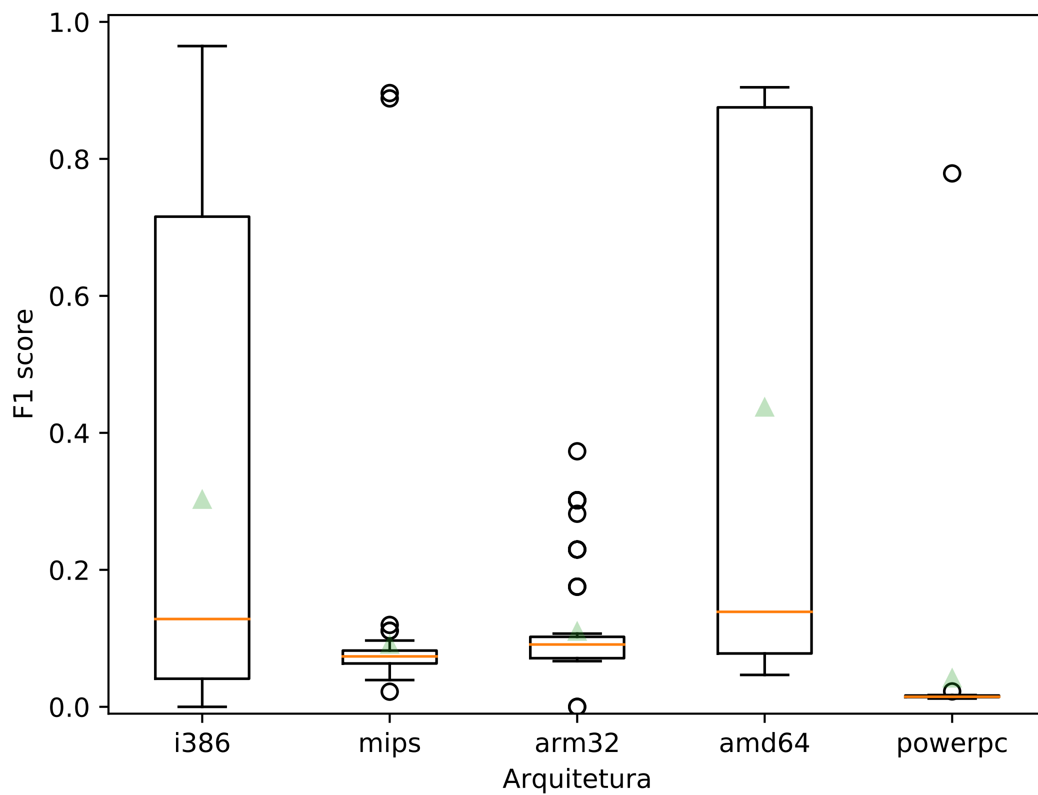


Figura 5.3: Melhor F1 score obtido pelo IDA Pro para o reconhecimento de funções de cada um dos 459 binários que originalmente continham nomes de símbolos.

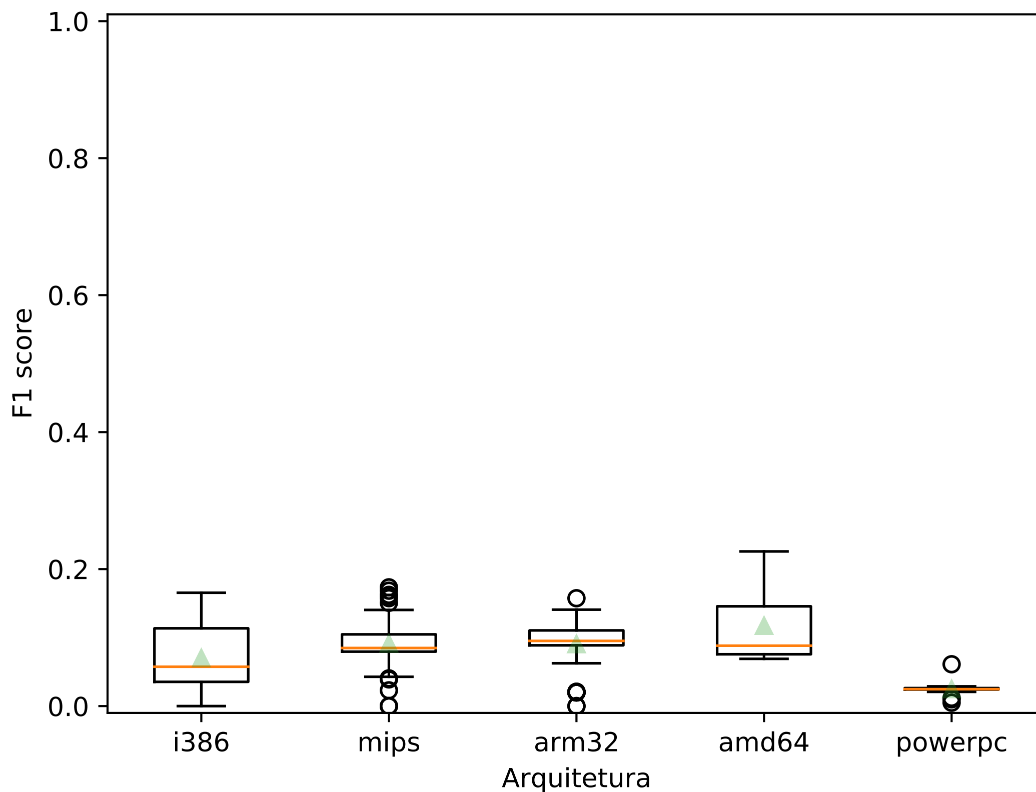


Figura 5.4: Melhor F1 score obtido pelo Radare2 para o reconhecimento de funções de cada um dos 459 binários que originalmente continham nomes de símbolos.

Nota-se que a acurácia de nenhuma das ferramentas testadas é consistente. O valor da mediana do *F1 score* fica sempre abaixo de 0,20. Ainda assim, algumas amostras puderam ser analisadas com boa acurácia pelo IDA Pro: 37% das amostras de amd64, 18% das de i386 e 2% das de MIPS obtiveram *F1 score* acima de 0,80.

A baixa acurácia deve-se, em grande parte, ao fato de que nem sempre é possível obter a versão exata da biblioteca padrão do C que havia sido utilizada para compilar os binários. Para ilustrar esse fato, os gráficos foram gerados novamente incluindo somente os binários para os quais a base de assinaturas possuía a versão exata da biblioteca, como estimada pela análise de *changelogs* a partir da versão de GCC embutida no binário. Os resultados são mostrados nas Figuras 5.5 e 5.6.

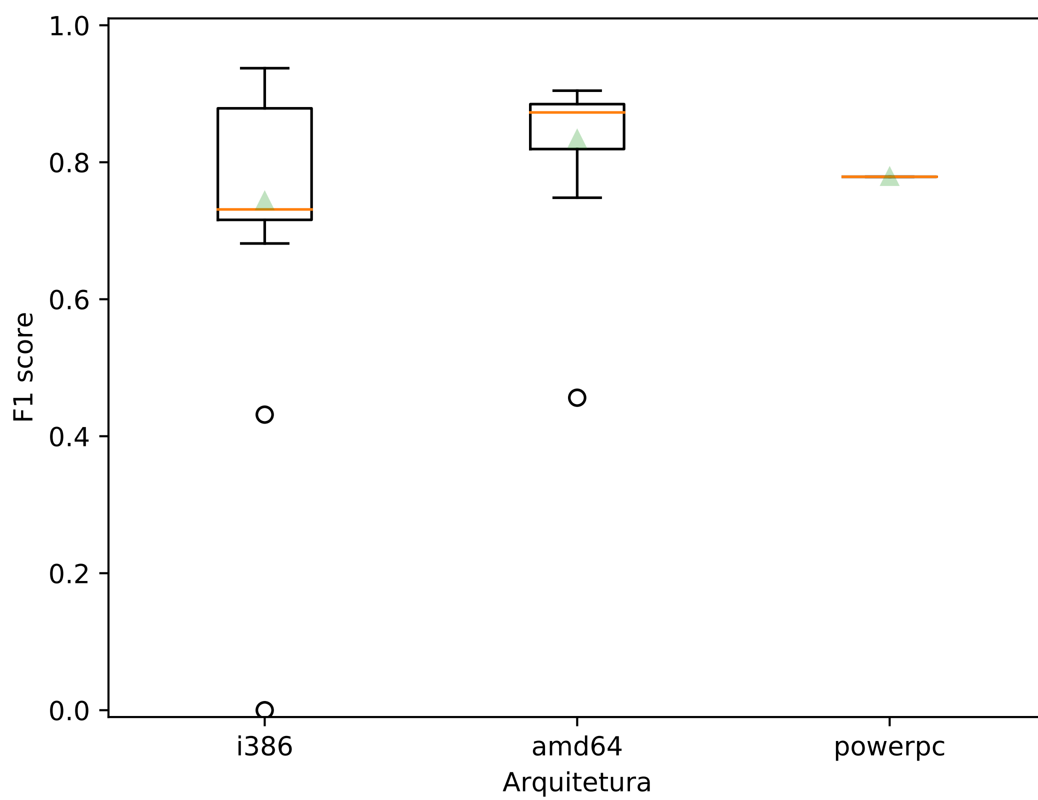


Figura 5.5: Melhor F1 score obtido pelo IDA Pro quando são considerados apenas binários para os quais a biblioteca padrão do C estava disponível na versão exata estimada via *changelogs*.

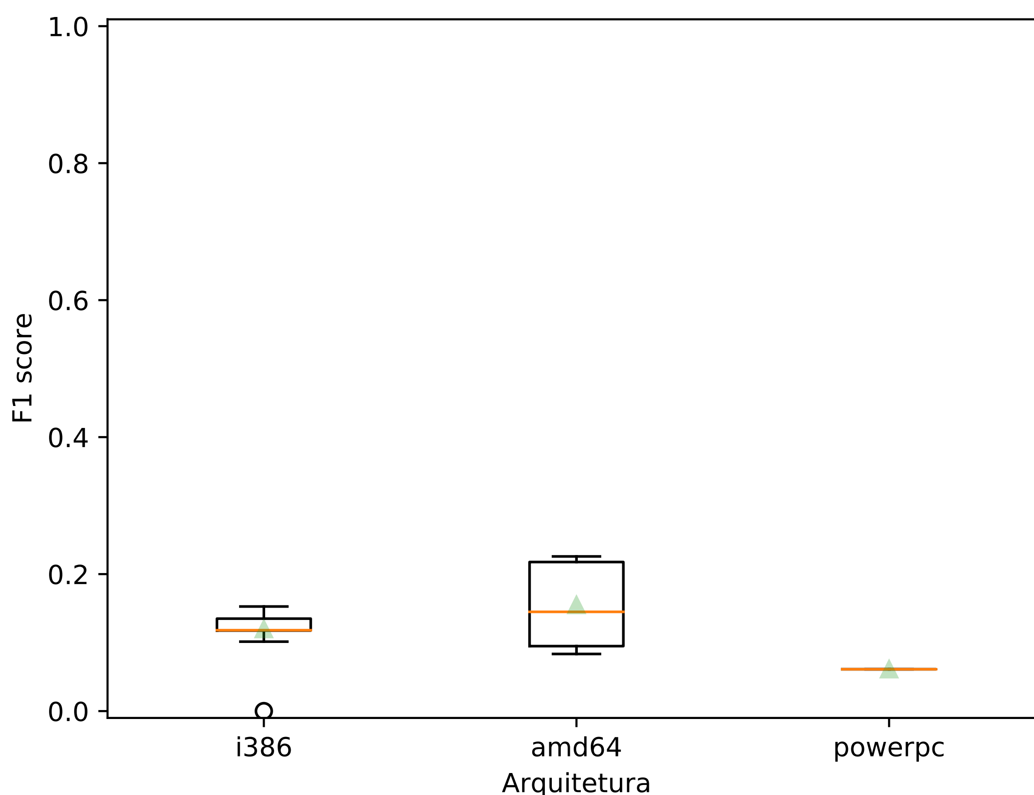


Figura 5.6: Melhor F1 score obtido pelo Radare2 quando são considerados apenas binários para os quais a biblioteca padrão do C estava disponível na versão exata estimada via *changelogs*.

Fazem parte do subconjunto de amostras apresentadas nesses gráficos 25 binários para i386, 19 para amd64 e 1 para PowerPC. Percebe-se que a acurácia do IDA Pro para a maioria dessas amostras é significativamente maior que a média e a mediana do conjunto completo.

Ainda assim, a acurácia do Radare2 continua consideravelmente baixa. Apesar de um dos métodos utilizados pelo Radare2 (o reconhecimento por sequências de *bytes*) ser funcionalmente equivalente ao F.L.I.R.T. utilizado pelo IDA Pro, a ferramenta `rasign2`, que gera as assinaturas para o Radare2, aparentemente ainda não produz assinaturas de qualidade tão alta quanto as geradas pelas ferramentas `pelf` e `sigmake`, do IDA Pro. Uma possível explicação é o fato de que o Radare2 ainda não utiliza as informações de relocação contidas em arquivos objeto para computar as máscaras das sequências[2].

[2]Comunicação pessoal de Pancake <https://github.com/radare>, fundador do Radare2, em 17 de Agosto de 2020, recebida pelo mensageiro instantâneo Telegram

5.3 Decaimento da acurácia com versões incorretas

Como o Radare2 implementa outros mecanismos de reconhecimento além do casamento de sequências de *bytes*, em particular a identificação de métricas similares em grafos de controle de fluxo (CFG), surge a seguinte questão de pesquisa: "as métricas baseadas em grafo têm potencial para melhorar o reconhecimento quando não é possível encontrar a versão exata de uma biblioteca vinculada estaticamente a um binário?"

Para responder a essa pergunta, foram selecionados os binários que possuíam pelo menos 50 símbolos de funções da biblioteca padrão do C, a fim de evitar que binários contendo poucas funções da biblioteca gerassem muito ruído na estatística. Dos 459 binários para os quais a verdade fundamental estava disponível, apenas 218 satisfaziam a esse requisito.

Os valores de F1 score das mesmas 30 assinaturas que foram amostradas para serem aplicadas pelo Radare2 foram selecionadas dentre os resultados obtidos pelo IDA Pro, para garantir que a comparação fosse o mais justa possível. O *F1 score* foi normalizado, dividindo-se o valor obtido com a aplicação de cada assinatura pelo valor obtido com a melhor assinatura do conjunto. Os dados foram dispostos em ordem decrescente e ajustados a uma curva exponencial do tipo $f(x) = \exp(-a(x-1))$.

Os resultados para o IDA Pro e Radare2 são apresentados, respectivamente, nas Figuras 5.7 e 5.8.

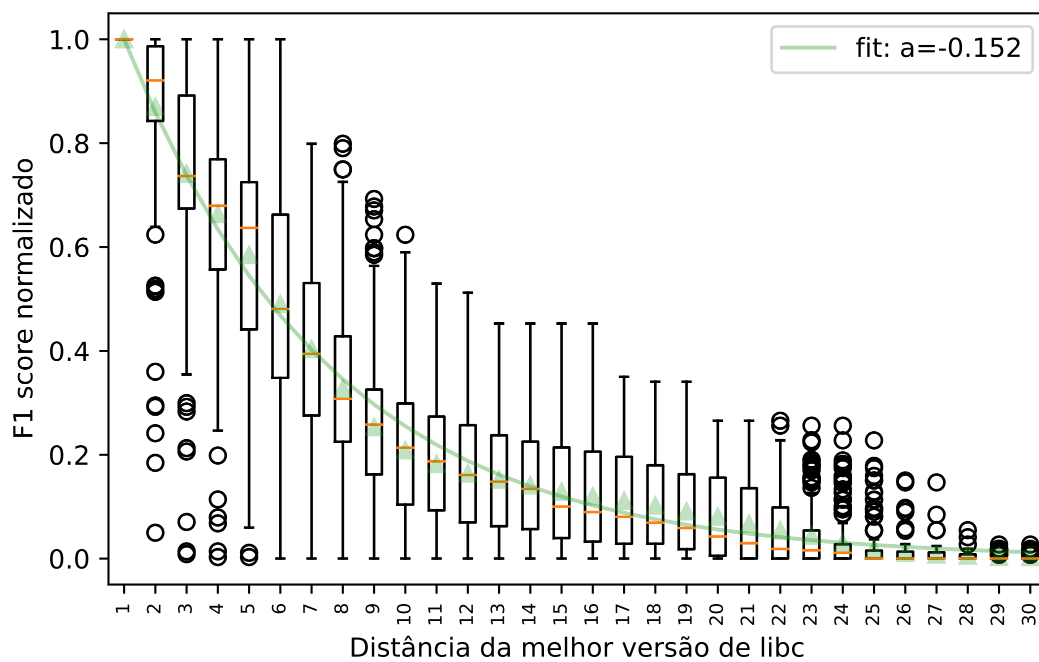


Figura 5.7: F1 score obtido pelo IDA Pro com assinaturas geradas a partir de diferentes versões da biblioteca libc, em ordem decrescente, e ajustado a uma curva de decaimento exponencial.

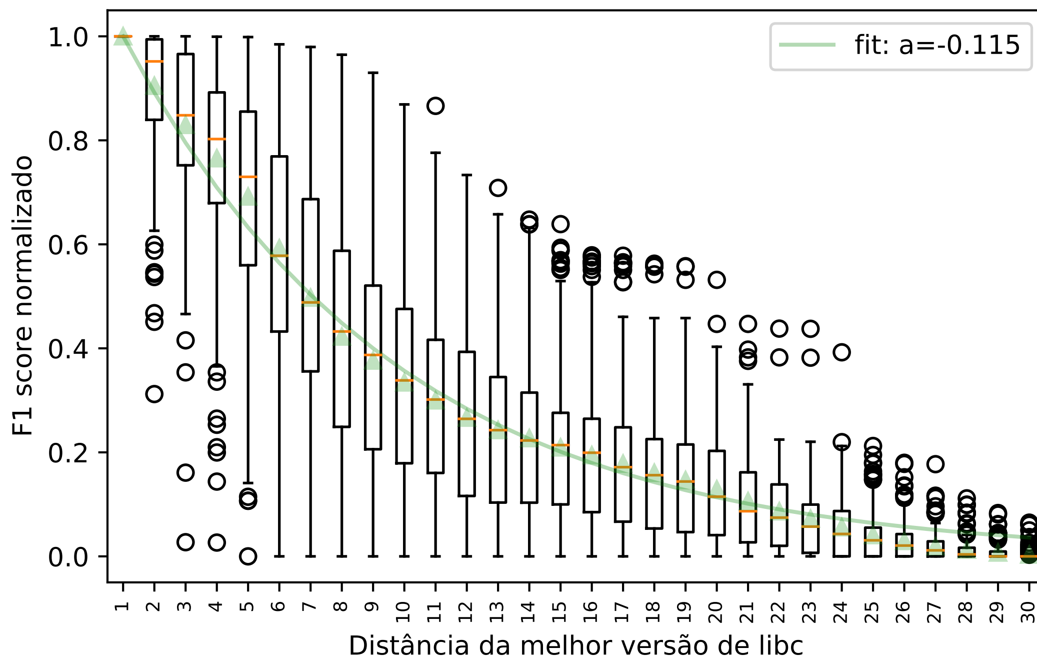


Figura 5.8: F1 score obtido pelo Radare2 com assinaturas geradas a partir de diferentes versões da biblioteca libc, em ordem decrescente, e ajustado a uma curva de decaimento exponencial.

Observa-se que, ao aplicar assinaturas de versões diferentes da versão ótima, o F1 score cai de forma um pouco mais lenta utilizando o Radare2 que com o IDA Pro. O coeficiente de decaimento com o Radare2 é de -0,115, comparado ao coeficiente de -0,152 obtido com o IDA Pro. A princípio, isso pode indicar que, uma vez corrigidos os problemas de acurácia que o Radare2 atualmente apresenta (mesmo quando é utilizada a versão correta da biblioteca), o Radare2 tem maior potencial de funcionar bem mesmo com versões de biblioteca ligeiramente diferentes daquela utilizada para compilar o binário.

No entanto, cabe ressaltar que são necessárias melhorias, também, nas métricas de grafo de controle de fluxo utilizadas pelo Radare2 para o reconhecimento das funções. Ao considerar para o cálculo do F1 score apenas as funções reconhecidas pelo mecanismo de grafos do Radare2, obtém-se a Figura 5.9.

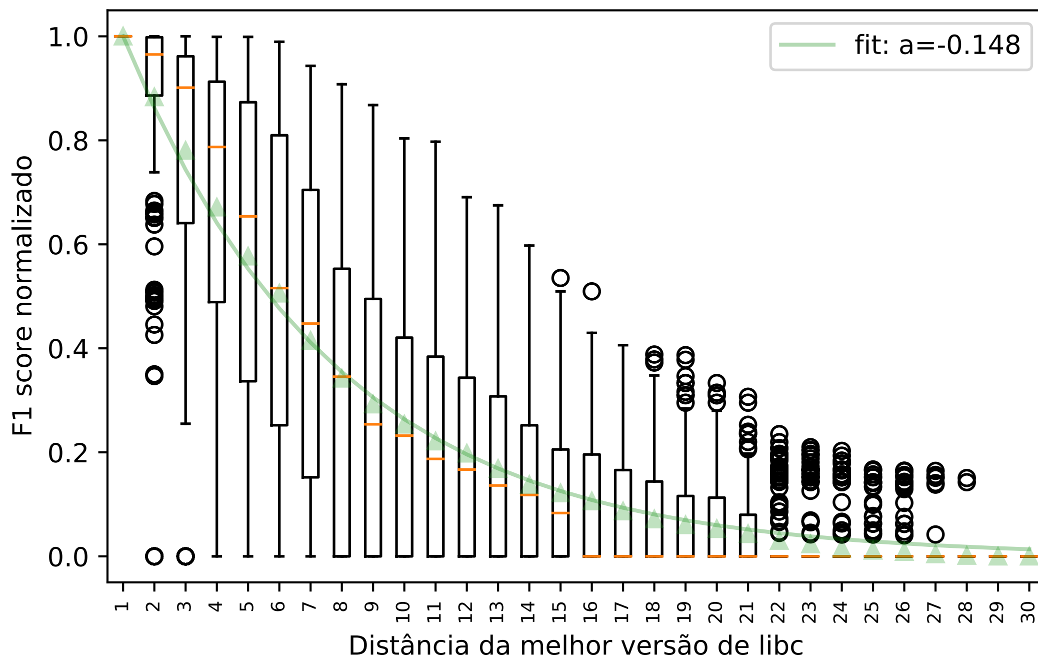


Figura 5.9: F1 score obtido pelo Radare2, isolando-se o mecanismo baseado em grafos, com assinaturas geradas a partir de diferentes versões da biblioteca libc, em ordem decrescente, e ajustado a uma curva de decaimento exponencial.

Apesar do mecanismo baseado em grafos do Radare2, funcionando isoladamente, obter um coeficiente de -0,148, que é melhor que o coeficiente -0,152 obtido pelo IDA Pro, a diferença entre esses coeficientes é baixa comparada à variância das medidas. Observa-se também uma queda abrupta na mediana do F1 score quando se atinge a décima sexta versão mais distante da versão ótima, comportamento que não ocorre nem com os resultados do IDA Pro, nem quando todos os mecanismos do Radare2 são considerados em conjunto.

5.4 Impacto de extrapolar para o conjunto completo

Para realizar análises sobres os 913 binários do conjunto completo, é necessário definir um critério para escolher a assinatura que gerou o melhor reconhecimento, permitindo que ela seja usada como referência, uma vez que a verdade fundamental não estará mais disponível.

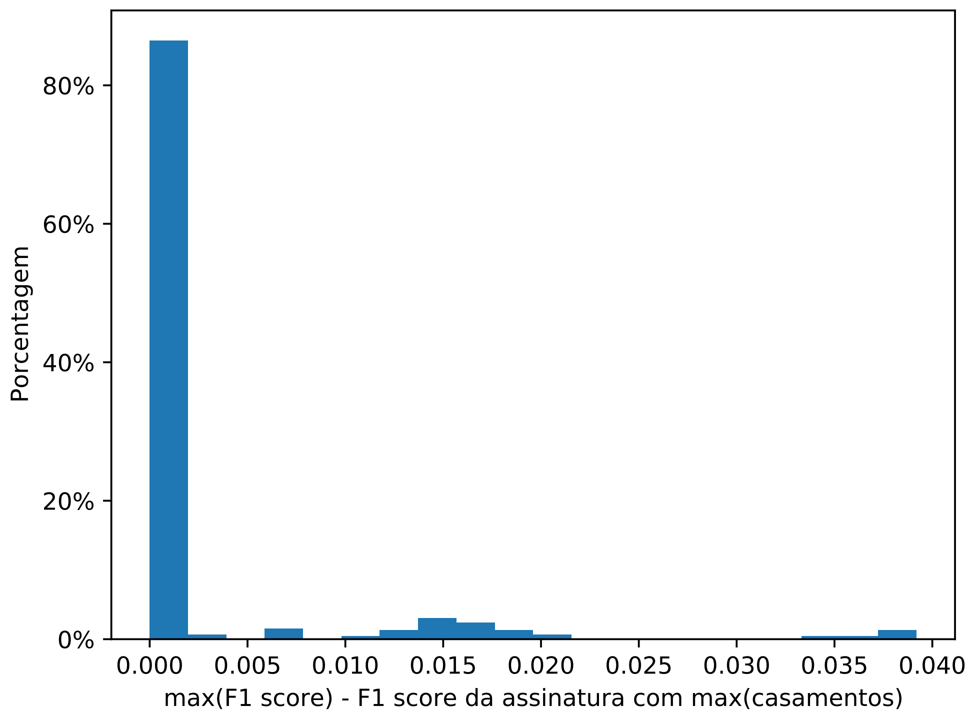


Figura 5.10: Histograma do desvio no F1 score obtido pelo IDA Pro com a escolha da assinatura que gerou o maior número de casamentos.

Propõe-se escolher como referência a assinatura que gerou o maior número de casamentos. O histograma da Figura 5.10 demonstra que utilizar essa métrica gera um desvio de no máximo 0,04 no F1 score quando comparado a escolher a assinatura que gerou o maior F1 score calculado com relação à verdade fundamental, sendo que em 89% dos casos o desvio é inferior a 0,01.

5.5 Acurácia com bibliotecas de outras distribuições

Muitas bases de bibliotecas padrão existentes na internet[3] disponibilizam apenas as versões publicadas pela distribuição Ubuntu Linux ou, quando muito, pelo Debian. Surge, portanto, a seguinte questão de pesquisa: "para alcançar uma boa acurácia, é necessário ter na base de assinaturas bibliotecas lançadas por diversas distribuições Linux, mesmo que as versões das bibliotecas sejam as mesmas?"

[3] <https://github.com/push0ebp/ALLirt>; <https://github.com/niklasb/libc-database>

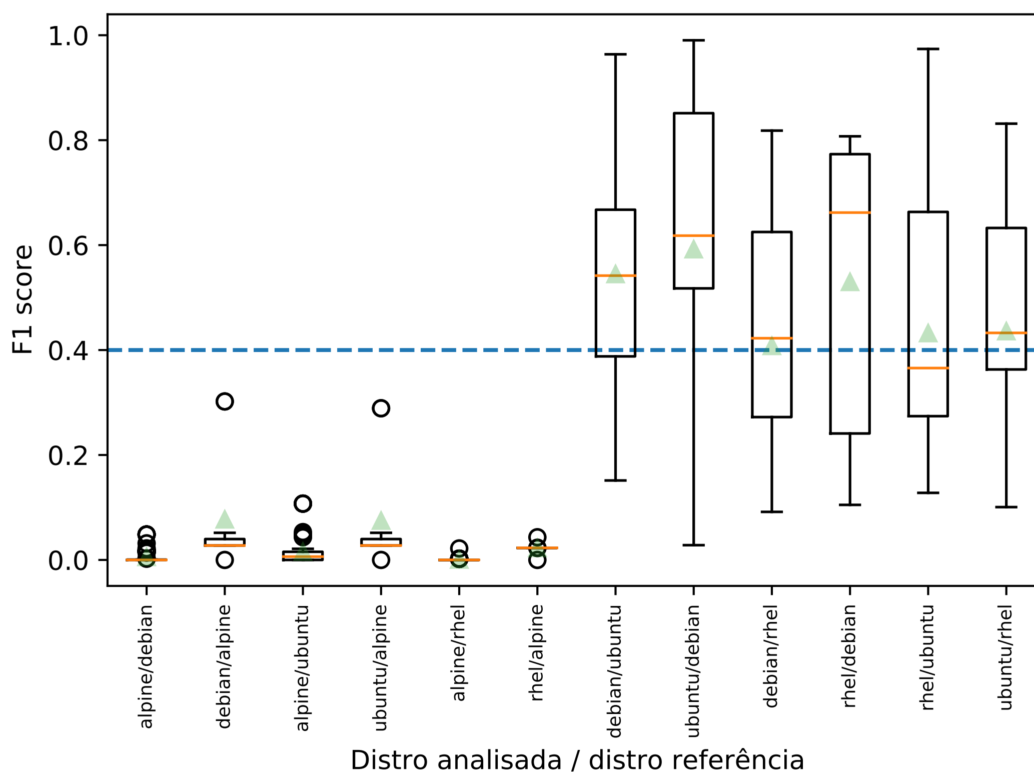


Figura 5.11: Melhor F1 score obtido com o IDA Pro usando assinaturas geradas a partir de bibliotecas de outra distribuição Linux.

A Figura 5.11 ilustra que analisar binários compilados no Red Hat Enterprise Linux (RHEL) com assinaturas geradas a partir de bibliotecas do Ubuntu ou do Debian produz, em média, cerca de 40% da acurácia que quando utilizadas assinaturas geradas a partir de bibliotecas do próprio RHEL. O mesmo ocorre quando binários compilados no Ubuntu são analisados com assinaturas geradas a partir de bibliotecas do RHEL.

Analisar binários do Debian com assinaturas do Ubuntu, ou vice-versa, também gera perda de acurácia, porém um pouco menos que nos casos anteriores. Esse fato era esperado, pois os pacotes do Ubuntu são construídos originalmente a partir de pacotes do Debian, apesar de muitas vezes usarem combinações diferentes com versão de compilador. No entanto, curiosamente, analisar binários do Debian com assinaturas do RHEL gera uma perda de acurácia comparável à da troca entre Ubuntu e Debian.

O pior caso ocorre quando binários do Alpine são analisados com assinaturas de alguma das outras distribuições, ou vice-versa. Esse fato também era esperado, pois o Alpine utiliza uma biblioteca padrão (denominada musl) completamente diferente

das outras distribuições. O gráfico mostra o caso de um *outlier* com cerca de 30% de acurácia quando analisado usando assinaturas do Debian ou Ubuntu. Analisando manualmente esse binário, notou-se que se tratava na verdade de um binário compilado no Debian, e que o grande número de casamentos com a assinatura do Alpine era devido a falsos positivos.

5.6 Avaliação da estimativa de versão

O presente trabalho propôs um método para estimar a versão da biblioteca padrão do C a partir da versão do compilador GCC, que muitas vezes pode ser encontrada nos binários, por meio de uma análise das datas de lançamento em *changelogs*.

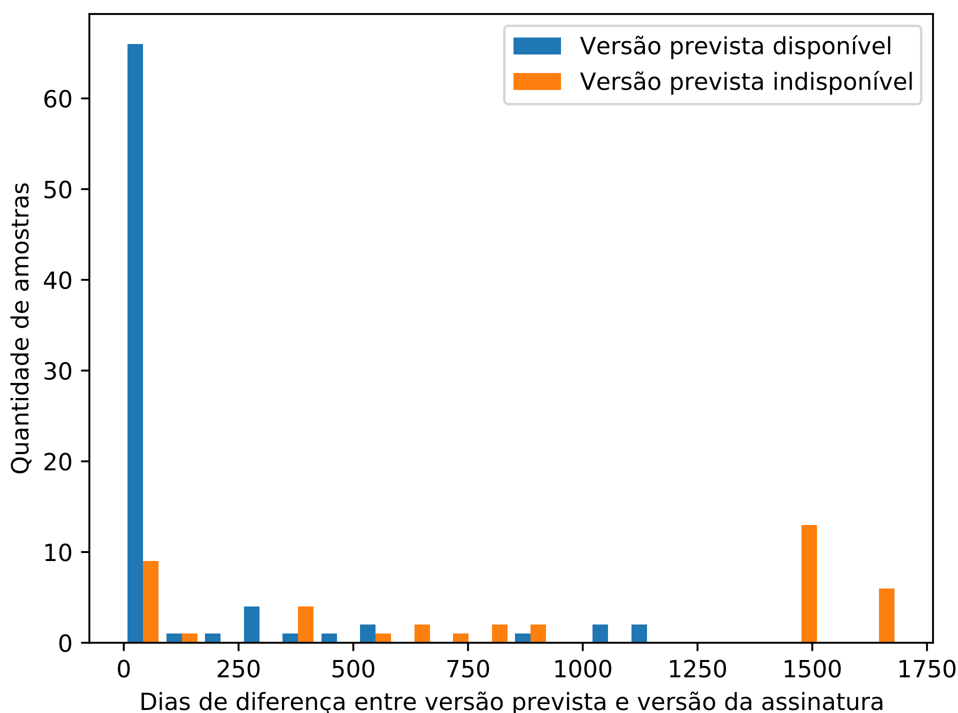


Figura 5.12: Histograma da quantidade de dias transcorridos entre o lançamento da versão de libc estimada pela análise de *changelogs* e o lançamento da versão que gerou o máximo de casamentos no IDA Pro.

Para verificar o quanto essa estimativa de versão concorda com a versão de assinatura que proporcionou o maior número de casamentos no IDA Pro, foi desenhado o histograma da Figura 5.12. O eixo horizontal representa a quantidade de dias que se

passaram entre o lançamento da versão da assinatura e o da versão estimada. Quanto mais próximo de zero, maior a concordância entre as medidas.

As barras em azul representam as medidas realizadas quando a versão estimada pela análise de *changelogs* estava disponível na base de assinaturas, o que gera, a princípio, resultados mais confiáveis. As barras em laranja representam as medidas realizadas quando apenas versões aproximadas estão disponíveis na base de assinaturas. Quando a versão prevista está disponível na base de assinaturas, ela gera o maior número de casamentos em 72% dos casos. Porém, quando não está disponível, uma versão lançada no mesmo mês gera o maior número de casamentos somente em 19% dos casos.

Os casos em que a diferença entre as datas de lançamento foi superior a 1250 dias correspondem a versões muito antigas da biblioteca, lançadas antes de 2004, que são muito difíceis de encontrar na internet.

5.7 Linha do tempo

Mesmo quando a versão do GCC não está disponível nas *strings* do binário, é possível estimar a data de compilação do binário (partindo do pressuposto que o sistema em que ele foi compilado está atualizado) com base na assinatura que proporcionou o maior número de casamentos no IDA Pro. Executa-se, para isso o caminho inverso: procura-se a versão da biblioteca padrão na base de *changelogs*.

A Figura 5.13 demonstra o resultado da aplicação dessa técnica para todos os binários da base coletada para a execução do presente trabalho. No entanto, cabe ressaltar que não é possível estimar a acurácia dessas medidas com os dados disponíveis, pois não existe uma verdade fundamental com a qual essa informação possa ser comparada. Para avaliar a acurácia dessa técnica, trabalhos futuros poderiam construir uma base de binários sintéticos com diversas versões de distribuições Linux, compiladores e bibliotecas.

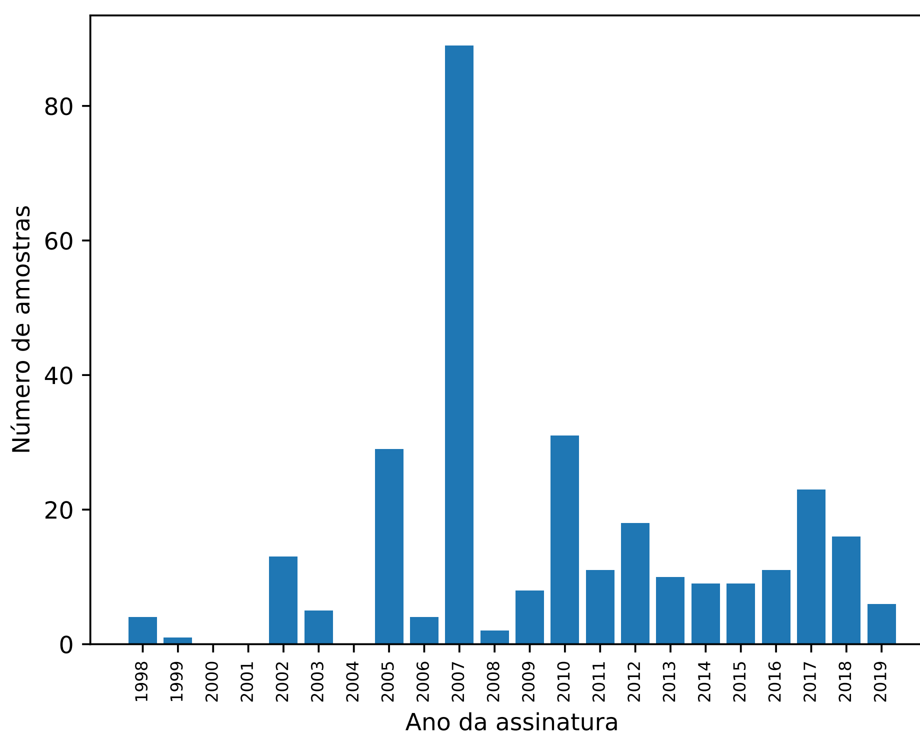


Figura 5.13: Quantidade de amostras da base analisada por ano em que se estima que elas foram compiladas.

É interessante o fato de ocorrer um pico no ano de 2007. Os binários desse ano pertencem a bases de *malware*, principalmente provenientes da VirusShare, e não às bases de CTFs. Procurou-se por algum relato de maior atividade de *malware* nesse ano, mas sem sucesso. A principal hipótese é, portanto, que tenha ocorrido algum pico de contribuições no repositório comunitário da VirusShare no mesmo período.

Capítulo 6

Considerações Finais

O presente trabalho realizou um estudo experimental e exploratório da técnicas utilizadas pelas ferramentas IDA Pro e Radare2 para tratar o problema do reconhecimento de funções de bibliotecas vinculadas estaticamente em binários. Como principais contribuições, destacam-se:

1. A construção de uma base de 913 binários ELF, dos quais 459 estão com os nomes de símbolos intactos.
2. A construção de uma base de 1601 versões de bibliotecas padrão, de diferentes distribuições Linux (Ubuntu, Debian, Red Hat Enterprise Linux e Alpine).
3. O desenvolvimento de um arcabouço de *scripts* para avaliar a acurácia de mecanismos de reconhecimento de funções vinculadas estaticamente.
4. A proposição de um método capaz de prever qual assinatura aplicar em um binário, com taxa de sucesso de 72% nos casos em que a versão do GCC está intacta no binário e em que a versão da biblioteca está disponível para geração da assinatura.

Estudando-se o funcionamento dos mecanismos de reconhecimento do Radare2, percebeu-se potencial na utilização de métricas de grafos de controle de fluxo para tornar o reconhecimento mais resiliente a pequenas variações nas versões de bibliotecas e, possivelmente, a alterações ocasionadas pelo *link-editor*. No entanto, a implementação do Radare2 é muito incipiente e apresenta baixa acurácia quando comparada a abordagens mais tradicionais, como o mecanismo F.L.I.R.T. do IDA Pro. Trabalhos futuros podem tanto resolver problemas de implementação do Radare2, avaliando

seu desempenho com base no arcabouço aqui desenvolvido, quanto propor novas métricas em grafos para melhorar a qualidade das assinaturas.

O método proposto neste trabalho para estimar a versão da biblioteca padrão a partir da versão do GCC pode ser aplicado no sentido inverso, de forma a estimar a data em que determinado binário foi compilado (pressupondo-se que o sistema em que ele foi compilado está atualizado). Essa técnica tem potencial de aplicação em forense digital, no entanto precisa ser melhor estudada e validada. Trabalhos futuros poderão utilizar a base de datas de lançamento levantada por este trabalho para validar essa técnica contra uma base de binários sintéticos gerados a partir de diversas versões de distribuições Linux, compiladores e bibliotecas.

Anexo A

Código para cálculo do *F1-Score*

```
relevant_documents = len(true_symbols)
retrieved_documents = 0
relevant_and_retrieved_documents = 0
for addr, names in recognized_symbols.items():
    true_name = true_symbols.get(addr)
    for name in names:
        retrieved_documents += 1
        if true_name and fuzzy_match(name, true_name):
            relevant_and_retrieved_documents += 1
precision = \
    relevant_and_retrieved_documents/retrieved_documents \
    if retrieved_documents else 0
recall = relevant_and_retrieved_documents/relevant_documents \
    if relevant_documents else 0
acc = 2/(1/precision + 1/recall) \
    if precision!=0 and recall!=0 else 0
```

Referências

AISAWA, W. et al. Breve comparação de ferramentas para análise estática de código malicioso. In: *SBRC 2019 - WSCDC ()*. [s.n.], 2019. Disponível em: <https://sol.sbc.org.br/index.php/wscdc/article/view/7706/7583>.

ALEXA CRAWLS. *DataRescue Home Page, home of IDA Pro*. 2020. <https://web.archive.org/web/19961102025534/http://www.datarescue.com/>. Acesso em: 14 Agosto de 2020.

ALLEN, F. E. Control flow analysis. In: ACM. *ACM Sigplan Notices*. [S.l.], 1970. v. 5, n. 7, p. 1–19.

ALLEN, K. et al. Machine literature searching viii. operational criteria for designing information retrieval systems. *American Documentation (pre-1986)*, Wiley Periodicals Inc., v. 6, n. 2, p. 93, 1955.

BAUMSTARK, N. *Building a libc offset database*. 2020. <https://github.com/niklasb/libc-database/>. Acesso em: 12 Agosto de 2020.

CERT.BR. *Cartilha de Segurança para Internet*. second. [S.l.]: Comitê Gestor da Internet no Brasil, 2012.

CHRISTODORESCU, M.; JHA, S. Static analysis of executables to detect malicious patterns. In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. Berkeley, CA, USA: USENIX Association, 2003. (SSYM'03), p. 12–12. Disponível em: <http://dl.acm.org/citation.cfm?id=1251353.1251365>.

FENG, J.; WANG, J.; LI, G. Triejoin: a triebased method for efficient string similarity joins. *The VLDB Journal*, Springer, v. 21, p. 437–461, 2012.

HEX-RAYS. *IDA F.L.I.R.T. Technology: In-Depth*. 2020. <https://www.hex-rays.com/products/ida/tech/flirt/in_depth/>.

HUANG, T.-M.; KECMAN, V.; KOPRIVA, I. *Kernel based algorithms for mining huge data sets*. [S.l.]: Springer, 2006. v. 1.

JACOBSON, E. R.; ROSENBLUM, N.; MILLER, B. P. Labeling library functions in stripped binaries. In: *Proceedings of the 10th ACM SIGPLANSIGSOFT workshop on Program analysis for software tools*. [S.l.: s.n.], 2011. p. 1–8.

LINUX MANPAGES. *Linux man page*. 2020. <https://linux.die.net/man/7/glibc>. Acesso em: 17 Setembro de 2020.

MCGRAW, G.; MORRISETT, G. Attacking malicious code: A report to the infosec research council. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 17, n. 5, p. 33–41, set. 2000. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/52.877857>.

MONARD, M. C.; BARANAUSKAS, J. A. Conceitos sobre aprendizado de máquina. *Sistemas inteligentes-Fundamentos e aplicações*, v. 1, n. 1, p. 32, 2003.

National Security Agency. *Ghidra*. 2020. <https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/FunctionID/src/main/doc/fid.xml>. Acesso em: 7 Agosto de 2020.

Pancake, and Melchor, Álvaro Felipe. *Radare2 Project*. 2020. <https://github.com/radareorg/radare2/blob/a023fe210c5d11888b50c1ea02539cfebb80327e/libr/anal/fcn.c#L1619>. Acesso em: 10 Agosto de 2020.

POWERS, D. M. Evaluation: from precision, recall and fmeasure to roc, informedness, markedness and correlation. Bioinfo Publications, 2011.

RINSMA, T. Automatic library version identification, an exploration of techniques. *arXiv preprint arXiv:1703.00298*, 2017.

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. [S.l.]: Malaysia; Pearson Education Limited,, 2016.

SHIRANI, P.; WANG, L.; DEBBABI, M. Binshape: Scalable and robust binary library function identification using function shape. In: SPRINGER. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. [S.l.], 2017. p. 301–324.

SIMON, P. *Too big to ignore: the business case for big data*. [S.l.]: John Wiley & Sons, 2013. v. 72.

VECTOR 35. *Binary Ninja*. 2020. <https://docs.binary.ninja/index.html>. Acesso em: 10 Agosto de 2020.

YIN, X. et al. Function recognition in stripped binary of embedded devices. *IEEE Access*, IEEE, v. 6, p. 75682–75694, 2018.