UNIVERSIDADE FEDERAL DO PAMPA

RONER DE CASTRO RODRIGUES

EXPLORAÇÃO DE CÓDIGO *ASSEMBLY* INTEL *X*64 OTIMIZADO PARA O *FUZZER* AFL

2023

RONER DE CASTRO RODRIGUES

EXPLORAÇÃO DE CÓDIGO *ASSEMBLY* INTEL *X*64 OTIMIZADO PARA O *FUZZER* AFL

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Bruno Silveira Neves

2023

Ficha catalográfica elaborada automaticamente com os dados fornecidos pelo(a) autor(a) através do Módulo de Biblioteca do Sistema GURI (Gestão Unificada de Recursos Institucionais).

Rodrigues, Roner de Castro

Exploração de código *assembly* Intel *x*64 otimizado para o *fuzzer* AFL / Roner de Castro Rodrigues.

101 p.

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Pampa, ENGENHARIA DE COMPUTAÇÃO, 2023.

"Orientação: Prof. Dr. Bruno Silveira Neves".

1. Análise dinâmica. 2. AFL. 3. AVX2. 4. ChatGPT. 5. Cybersegurança. 6. Fuzzing. 7. Paralelismo a nível-de-instrução. 8. SIMD. I. Título.

SERVIÇO PÚBLICO FEDERAL

MINISTÉRIO DA EDUCAÇÃO

Universidade Federal do Pampa

RONER DE CASTRO RODRIGUES

EXPLORAÇÃO DE CÓDIGO ASSEMBLY INTEL X64 OTIMIZADO PARA O FUZZER AFL

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação..

Trabalho de Conclusão de Curso defendido e aprovado em: 06, de dezembro de 2023.

Banca examinadora:

____

Prof. Dr. Bruno Silveira Neves
Orientador
UNIPAMPA

____

Prof. Dr. Fabio Luis Livi Ramos
UNIPAMPA

____

Prof. Dr. Gerson Alberto Leiria Nunes
UNIPAMPA

Assinado eletronicamente por **BRUNO SILVEIRA NEVES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 16/12/2023, às 02:26, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **FABIO LUIS LIVI RAMOS**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 18/12/2023, às 22:34, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **GERSON ALBERTO LEIRIA NUNES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 19/12/2023, às 00:04, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1332687** e o código CRC **F2EBC7BF**.

Referência: Processo nº 23100.025565/2023-71 SEI nº 1332687

"Coloquei meu coração e alma em meu trabalho e perdi minha mente no processo."
- Vincent Van Gogh

RESUMO

Nas últimas décadas, percebe-se um aumento inegável na popularização da tecnologia digital em todo o mundo, assim como uma crescente dependência dela em todos os setores da sociedade. Contudo, o nível de preocupação com *cybersegurança*, por parte das empresas e profissionais relacionados, geralmente não está em consonância com os atuais padrões de segurança recomendados e, devido a isso, empresas do setor têm observado, com frequência indesejada, violações de segurança, negações de serviço, vazamento de dados sensíveis e toda a sorte de ações indesejadas. Nesse contexto, fica evidente a necessidade do desenvolvimento de tecnologias que permitam, se não eliminar, ao menos mitigar consideravelmente a superfície de ataque de aplicações comerciais modernas. Nesse trabalho, realizou-se um estudo empírico, a fim de se verificar o espaço de otimização em nível *assembly* existente para um *fuzzer* de aplicação da atualidade. Através do uso do conjunto de instruções vetorizadas AVX2, explorou-se paralelismo a nível-de-instrução da CPU para otimização da principal função presente no *fuzzer* estudado, alcançando-se com isso ganhos consideráveis no desempenho dessa função específica. Dessa maneira, uma vez integrada na aplicação, obteve-se uma redução de 37.37% no *overhead* do tempo de execução da função otimizada, no melhor caso, enquanto que, para o pior caso, obteve-se uma redução desse *overhead* de 16.14%.

Palavras-chave: Análise dinâmica. AFL. AVX2. ChatGPT. Cybersegurança. Fuzzing. Paralelismo a nível-de-instrução. SIMD.

ABSTRACT

In recent decades, there has been an undeniable increase in the popularization of digital technology worldwide, along with a growing dependency on it in all sectors of society. However, the level of concern with cybersecurity, by companies and related professionals, usually does not align with the current recommended security standards. Consequently, companies in the sector have frequently observed undesirable occurrences of security breaches, denial of service, leakage of sensitive data, and all sorts of unwanted actions. In this context, it becomes evident the need for the development of technologies that allow, if not eliminating, at least significantly mitigating the attack surface of modern commercial applications. In this study, an empirical investigation was conducted to assess the optimization potential at the *assembly* level for a contemporary *fuzzer* application. By employing the AVX2 set of vectorized instructions, parallelism at the CPU instruction level was leveraged to optimize the main function present in the studied *fuzzer*, resulting in substantial performance improvements for this specific function. Consequently, once integrated into the application, a reduction of 37.37% in the execution time overhead of the optimized function was achieved in the best-case scenario, while in the worst-case scenario, the overhead reduction was of 16.14%.

Keywords: *AFL*, *AVX2*, *ChatGPT*, *ChatGPT*, *Dynamic Analysis*, *Fuzzing*, *Instruction-level parallelism* , *SIMD*.

LISTA DE FIGURAS

1 Primeiro caso reportado de um "*bug* de computador". 19
2 Tendências acompanhadas no aumento de relatórios de vulnerabilidades e incidentes, ao longo de 32 anos. 20
3 Cobertura de testes provida pela análise de código estática. 24
4 Representação da cobertura de testes provida pela análise de código dinâmica. 26
5 Diagrama idealizado para a exploração de um programa C vulnerável. 27
6 Fluxo de trabalho clássico de um *fuzzer*. 28
7 Custo relativo para se fixar *bugs*, com base em seu tempo de detecção. 29
8 SDL atualmente adotado pela Microsoft, onde verifica-se o uso de técnicas de *fuzzing* durante a etapa de Verificação dos seus produtos. 30
9 Diagrama arquitetural de um sistema de *fuzzing* clássico. 36
10 Representação do processo de mapeamento de arestas de transiçao entre blocos e o *bitmap* que é utilizado pelo AFL. 41
11 Representação do fluxo de execução de um programa, em termos de blocos básicos (vértices) e suas transições (arestas). 42
12 Processo de calibração de entradas do AFL. 44
13 Fluxo de trabalho do AFL. 45
14 Taxa de colisão de *hashes*, em função do tamanho do *bitmap*. 48
15 Velocidade de execução, em função do tamanho do *bitmap*. 49
16 Previsão para o mercado de IA Generativa. 52
17 Análise de *profiling* realizado para o AFL, ao longo de 10 minutos, onde destaca-se o total de chamadas para *has_new_bits()* nesse período, com um valor de centenas de milhares de execuções. 64
18 Captura de tela do Utilitário para identificação do processador Intel, listando as tecnologias disponíveis na CPU utilizada, destacando a disponibilidade do conjunto de instruções AVX2. 66
19 Captura de tela da ferramenta ChatGPT sendo utilizada para auxílio na geração da versão inicial do *assembly* otimizado proposto. 67
20 Representação conceitual da estratégia de *prefetching* utilizada, buscando disponibilizar na cache os dados dos *arrays* trabalhados, durante as iterações de *has_new_bits()*. 68
22 Fluxograma para validação das saídas esperadas para a função *has_new_bits()* original e sua versão *assembly* otimizada. 70
21 Algoritmo de *has_new_bits()* original, com destaque para as seções nas quais a versão otimizada faz uso das estratégias de otimização de laço (em azul), em conjunto com as instuções AVX2 (em verde). 71
23 Análise do desempenho para a função *has_new_bits()* otimizada com relação a sua versão original, para um *bitmap* de 64KB. 77
24 Análise do desempenho para a função *has_new_bits()* otimizada com relação a sua versão original, para um *bitmap* de 128KB. 77
25 Análise do desempenho para a função *has_new_bits()* otimizada com relação a sua versão original, para um *bitmap* de 256KB. 78
26 Tempos de execução percentuais sobre o tempo de execução total da aplicação, para as duas versões de *has_new_bits()*, normal e otimizada, para os diferentes tamanhos de *bitmap* testados. 79

27 Total de execuções do SUT alcançadas para um determinado intervalo de tempo, antes da integração com a versão *assembly* otimizada de *has_-new_bits()*. 80
28 Total de execuções do SUT alcançadas para um determinado intervalo de tempo, após a integração com a versão *assembly* otimizada de *has_new_-bits()*. 80
29 Código da função *has_new_bits()*. 90
30 Função *has_new_bits()* otimizada desenvolvida. 96

LISTA DE TABELAS

1 Resumo das informações e referências utilizadas 58
2 Dados de *profiling* do AFL para diferentes tamanhos de *bitmap*. 62
3 Análise estatística das instruções do código *assembly* padrão para a função *has_new_bits()*. 66
4 Análise das instruções do código *assembly* otimizado para a função *has_new_bits()*. 70
5 Ganho médio percentual para diferentes tamanhos de *bitmap*. 74
6 Tempos médio de execução e variância para *has_new_bits* original e sua versão otimizada, em microssegundos, para um *bitmap* de 64KB. 75
7 Tempos médio de execução e variância para *has_new_bits* original e sua versão otimizada, em microssegundos, para um *bitmap* de 128KB. 76
8 Tempos médio de execução e variância para *has_new_bits* original e sua versão otimizada, em microssegundos, para um *bitmap* de 256KB. 76
9 Dados comparativos de *profiling* do AFL para diferentes tamanhos de *bitmap*, já integrado com a versão otimizada de *has_new_bits()* desenvolvida.. 79

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AFL	*American Fuzzy Lop*
AVX2	*Advanced Vector Extensions 2*
CPU	*Central Processing Unit*
DoD	*Department of Defense*
ETSI	*European Telecommunications Standards Institute*
GCC	*GNU C Compiler*
HTML	*Hypertext Markup Language*
IA	*Inteligência Artificial*
IEEE	*Institute of Electrical and Electronics Engineers*
IETF	*Internet Engineering Task Force*
NASM	*Netwide Assembler*
NIST	*National Institute of Standards and Technology*
NSA	*National Security Agency*
NVD	*National Vulnerability Database*
OS	*Operating System*
RFC	*Request For Comments*
SAGE	*Scalable Automated Guided Execution*
SDK	*Software Development Kit*
SIMD	*Single Instruction, Multiple Data*
SMB	*Server Message Block*
SUT	*System Under Test*
WSL2	*Windows Subsystem for Linux, versão 2*

SUMÁRIO

1 Introdução 15
- 1.1 Justificativa e contribuições desse trabalho 16
- 1.2 Objetivo geral 17
- 1.3 Objetivos específicos 17
- 1.4 Organização do trabalho 17

2 Referencial Teórico 18
- 2.1 *Bugs* de computador e vulnerabilidades em *softwares* 18
- 2.2 Robustez de *softwares* 21
- 2.3 Metodologias de análise de vulnerabilidades 22
- 2.4 História e motivações do *fuzzing* 27
  - 2.4.1 Vantagens do *fuzzing* 31
  - 2.4.2 Desvantagens do *fuzzing* 33
- 2.5 Anatomia de sistemas *fuzzing* 34
- 2.6 Taxonomia de sistemas *fuzzing* 35
  - 2.6.1 Baseados-em-mutação 35
  - 2.6.2 Baseados-em-geração 38
  - 2.6.3 Evolucionários 38
- 2.7 O *fuzzer* American Fuzzy Lop (AFL) 39
  - 2.7.1 Funcionamento e modos de operação 39
  - 2.7.2 Fluxo de trabalho 42
  - 2.7.3 Limitações e gargalos conhecidos 45
- 2.8 *Benchmarking* de *fuzzers* 48
- 2.9 O cenário atual de otimização em *software* através do uso de IA Generativa 50
- 2.10 O conjunto de instruções Advanced Vector Extensions 2 (AVX2) 53

3 Metodologia 54
- 3.1 Problema de Pesquisa 54
- 3.2 Hipótese de Pesquisa 54
- 3.3 Caracterização da Pesquisa 55
- 3.4 Procedimento Metodológico 56
- 3.5 Revisão da Bibliografia 57

4 Desenvolvimento do *assembly* otimizado 59
- 4.1 Delimitação do Problema 59
  - 4.1.1 Seleção de um *fuzzer* para análise 59
  - 4.1.2 Seleção de um corpo de testes 60
  - 4.1.3 Seleção de um *profiler* para análise 61
- 4.2 *Profiling* do *fuzzer* AFL 62
  - 4.2.1 Análise dos gargalos encontrados 62
- 4.3 Projeto e implementação do *assembly* otimizado na CPU 65
  - 4.3.1 Análise do *assembly* original de *has_new_bits()* 65
  - 4.3.2 O uso da ferramenta ChatGPT no processo de desenvolvimento do código otimizado 67
  - 4.3.3 Análise do *assembly* otimizado para *has_new_bits()* 67

5 Análise dos Resultados 72

5.1 Laboratorio de Experimentos ... 72
5.2 Análise de desempenho para as funções *has_new_bits()* original e otimizada, de maneira isolada ... 73
5.3 Análise de *profiling* do AFL após a integração com a versão otimizada de *has_new_bits()* desenvolvida ... 78

6 Considerações Finais **82**
6.1 Trabalhos Futuros ... 82
6.2 Contribuições desse trabalho ... 83

Referências **84**

APÊNDICE A – Código *assembly* original para *has_new_bits()*, sem otimização **90**

APÊNDICE B – Código *assembly* otimizado desenvolvido para *has_new_bits()* **95**

APÊNDICE C – Cálculos para determinação do ganho de desempenho comparativo para as versões normal e otimizada de *has_new_bits()*. **99**

1 INTRODUÇÃO

A exploração de vulnerabilidades em *softwares*, decorrentes dos mais variados tipos de *bugs*, tem sido uma realidade há mais de 30 anos, assim como técnicas que se valem disto com os mais diversos fins (WEBER, 2010). Sendo assim, crimes virtuais acontecem de forma cada vez mais frequentes, a ponto do prejuízo estimado das empresas já ter ultrapassado a margem de dois trilhões de dólares em 2020 (FORBES.COM, 2016). Dessa forma, fabricantes de *hardware* e *software* têm implementado ao longo dos anos diversas mitigações contra vetores de ataques comumente explorados por atacantes, visando dessa forma uma mitigação total ou parcial dos possíveis impactos que podem ser causados por um usuário mal-intencionado, por meio de falhas em seus produtos. Isto soma-se ao fato de que tais atacantes ganham cada vez mais notoriedade, uma vez que a popularização de moedas digitais acaba por fomentar a prática crescente de técnicas de estelionato, através dos já famigerados *ransomware* e de toda sorte de técnicas de engenharia social.

No Brasil, medidas de regulação de privacidade como a LGPD[1], visam desestimular essa casuística de crimes envolvendo vazamento de informações pessoais. No exterior, parceiros da indústria e agências reguladoras já adotam e recomendam a testagem de segurança de recursos críticos via-fuzzing, dentre elas: IEEE, IETF, ETSI, NIST, DoD e a própria NSA. Com a adoção cada vez maior da modalidade de trabalho *home-office* e em vista da quarta revolução digital, a pesquisa acerca do tema cresce à medida em que a preocupação com segurança digital é essencial do planejamento estratégico das empresas (GLOBO, 2021).

Nesse contexto, verifica-se ainda que a busca por *bugs* em *softwares* é uma disciplina que desde muito tempo abrange múltiplos segmentos. Muitas vezes, tais *bugs* são tidos como inofensivos, provocando apenas mudanças sutis no fluxo de execução esperada do programa, podendo assim permanecer indetectáveis por um longo período de tempo. Porém, algumas vezes, tais *bugs* acarretam em falhas de segurança graves, vindo a representar uma ameaça à integridade de toda uma empresa, com impactos que podem variar desde a violação de mecanismos de segurança até cenários que podem criar risco de vida às pessoas. Por exemplo, em 1987 *bugs* do tipo *race condition* na máquina de radioterapia Therac-25, um equipamento muito moderno para sua época, foram responsáveis pela morte de quatro pessoas, além de deixar outras duas com lesões permanentes (COMPANY, 2021b). Em 1996, *bugs* no sistema de referência inercial, controlados por compu-

[1]Lei Geral de Proteção de Dados (Lei nº 13.853, de 2019)

tador no foguete Ariane-5, foram diretamente responsáveis pela sua explosão. Estima-se a perda na ordem dos 370 milhões de dólares ao programa espacial americano, além de atrasar as pesquisas científicas esperadas do investimento em cerca de quatro anos. Esses exemplos não-usuais são parte dos "piores *bugs* de computador da história"(COMPANY, 2021a), mas por muito menos, grandes desastres decorrentes de falhas de computador são percebidos diariamente por profissionais de segurança no mundo todo.

Dessa forma, este trabalho tem como objetivo contribuir com a comunidade de segurança, através de um estudo empírico que visa verificar se existe e, se existir, qual é o espaço de exploração de paralelismo em nível-de-instrução que possa vir a fornecer um aumento do desempenho na execução de um *fuzzer* moderno.

1.1 Justificativa e contribuições desse trabalho

A crescente popularização de técnicas de *hacking*, o crescimento na demanda por profissionais de segurança e a migração para uma infraestrutura totalmente digital, nos traz a oportunidade de aprimorar as tecnologias que se revelam como essenciais para toda uma nova geração de cyberarmas. Em vista disso, ferramentas de *fuzzing* se revelam, dentre as ferramentas do arsenal de profissionais da área, uma ferramenta fundamental para exploração de *softwares* comerciais modernos, em especial, o *fuzzer* AFL[2], conhecido como um divisor de águas entre as ferramentas desse tipo. Dessa forma, exploraram-se nesse trabalho as seguintes contribuições para essa ferramenta:

- Exploração dos gargalos e do espaço de otimização disponível, identificando-se o ponto mais viável e propenso à otimização através do uso de conjuntos de instrução vetorizados;
- Validação da capacidade de ferramentas de Inteligência Artificial (IA) modernas em auxiliar no projeto e implementação de código *assembly* Intel *x*64, otimizado para uma microarquitetura especificada;
- Resultados teóricos e práticos com a CPU, exibindo a viabilidade de exploração de paralelismo-a-nível-de-instrução do gargalo escolhido. Além disso, implementou-se uma versão *assembly* otimizada do trecho de código escolhido que, por sua vez, trouxe uma redução considerável no *overhead* do tempo execução desta, quando comparada a sua versão original.

[2]American Fuzzy Lop, disponível em https://github.com/google/AFL

1.2 Objetivo geral

Como objetivo geral, esse trabalho se propõe a identificar e explorar possíveis gargalos no *fuzzer* escolhido. Para a execução dessa proposta, busca-se a exploração de paralelismo em nível-de-instrução em uma CPU da atualidade. Dessa forma, busca-se encontrar uma maneira de fornecer, ao *fuzzer* AFL, alternativas de implementação de funções que se revelem custosas em termos de tempo de processamento, permitindo que a ferramenta em questão se aproveite melhor dos recursos providos por uma microarquitetura Intel específica.

1.3 Objetivos específicos

- Determinar, no *fuzzer* escolhido para ser trabalhado, qual é o possível espaço de otimização existente neste;
- Determinar qual tecnologia é a mais propícia para exploração de um possível espaço de otimização que seja encontrado;
- Fazer uso de ferramentas de *profiling* que permitam determinar, de fato, qual foi e se houve, de fato, ganho de desempenho para o *fuzzer* escolhido.

1.4 Organização do trabalho

O Capítulo 1 apresenta os motivadores acerca deste trabalho; O Capítulo 2 apresenta o Referencial Teórico sobre o qual esse trabalho se baseia, além de detalhar as tecnologias envolvidas no projeto; O Capítulo 3 apresenta a metodologia científica que norteou este trabalho; O Capítulo 4 apresenta os detalhes acerca da implementação do código *assembly* otimizado desenvolvido; No Capítulo 5 são analisados os dados colhidos durante os experimentos realizados; Por fim, o capítulo o Capítulo 6 fala sobre a contribuição científica dessa pesquisa, idealiza trabalhos futuros correlatos e exibe as conclusões obtidas ao final deste trabalho de conclusão de curso.

2 REFERENCIAL TEÓRICO

Neste capítulo, será realizada a revisão da literatura acerca dos conceitos e tecnologias que norteiam este trabalho. A seção 2.1, aborda a questão de vulnerabilidades em *softwares*, examinando sua natureza e história, a fim de introduzir o leitor no contexto de segurança acerca do *fuzzer* sobre o qual o trabalho se baseia; A seção 2.2, fala do conceito de robustez em *softwares*, que traz ao leitor definições mais formais de segurança para o tipo de aplicação que é utilizada como corpo de testes nesse trabalho; Na seção 2.3 são elencadas algumas das metodologias de análises de vulnerabilidades mais conhecidas, análise estática e dinâmica, exibindo as diferenças entre essas duas abordagens; A seção 2.4, conta um pouco da história acerca da tecnologia de *fuzzing*, listando algumas das vantagens e desvantagens na adoção desta técnica para a análise de segurança em aplicações modernas; A seção 2.5 detalha os componentes considerados essenciais para um *fuzzer*, destacando o processo de cobertura de código, cujo *overhead* de processamento é bem-conhecido e é tratado nesse trabalho; Na seção 2.6 são listados as principais categorias de *fuzzers* existentes, explicando os conceitos e casos de uso mais comuns; A seção 2.7 fala sobre o *fuzzer* sobre o qual esse trabalho foi realizado, descrevendo seu fluxo de trabalho e gargalos conhecidos, com enfoque no gargalo que fora escolhido para o presente trabalho; A seção 2.8 fala sobre o processo de *benchmarking* para *fuzzers* modernos, estabelecendo dessa forma as métricas necessárias para avaliação de desempenho do *fuzzer* otimizado desenvolvido no presente trabalho; Na seção 2.9 é explorado o cenário atual de otimização de código *assembly* Intel através do uso de recursos de IA, demonstrando casos de uso bem sucedidos dessa recente tecnologia, no contexto de otimização de código e, por fim, na seção 2.10 é detalhado o conjunto de instruções AVX2, utilizado para o desenvolvimento do *assembly* customizado desenvolvido neste trabalho.

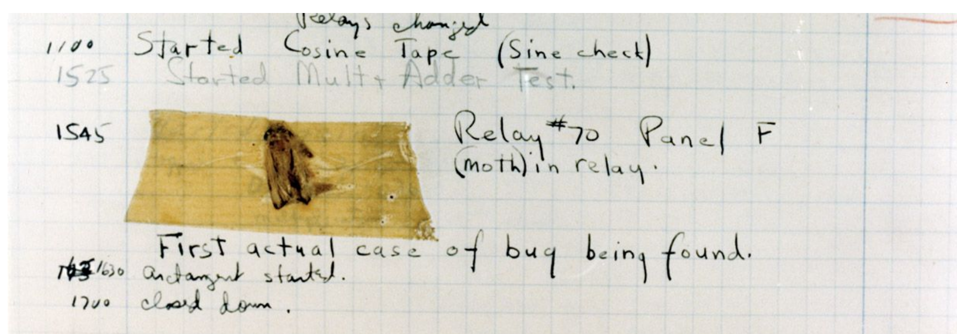
2.1 *Bugs* de computador e vulnerabilidades em *softwares*

A etimologia do termo "*bug*" remonta por volta de 1870, quando o então engenheiro eletricista Thomas Edison, utilizava-o para *design* ar o mal-funcionamento de suas máquinas. Contudo, fora no ano de 1947 que o termo *bug* popularizou-se como atualmente o conhecemos. Na ocasião, o computador Mark-II da Universidade de Harvard (EUA) quebrou e, quando o problema foi inspecionado, os engenheiros surpreenderam-se com a causa encontrada: uma mariposa, que entrou no computador (em busca talvez de

luz e calor) e, com isso, provocara um curto-circuito na placa (BBVA, 2021).

Segundo (JUUSO; TAKANEN, 2011), pode ser definido como um erro, uma falha ou uma anomalia em um computador ou em um sistema de computadores que por sua vez, podem vir a produzir um resultado indesejado ou levar a um comportamento indevido. Esses *bugs*, por sua vez, derivam de erros no código-fonte da aplicação ou até mesmo de seu *design* (CHEN et al., 2018). Para eliminação de tais erros, diversos métodos de testes são atualmente escritos manualmente ou com ajuda de *frameworks*, principalmente com o objetivo de se validar a funcionalidade esperada dos componentes da aplicação. Visualiza-se na Figura 1 um extrato do relatório gerado pelos técnicos, de quando o termo *bug* passou a ser utilizado com o significado atual.

Figura 1 – Primeiro caso reportado de um "*bug* de computador".

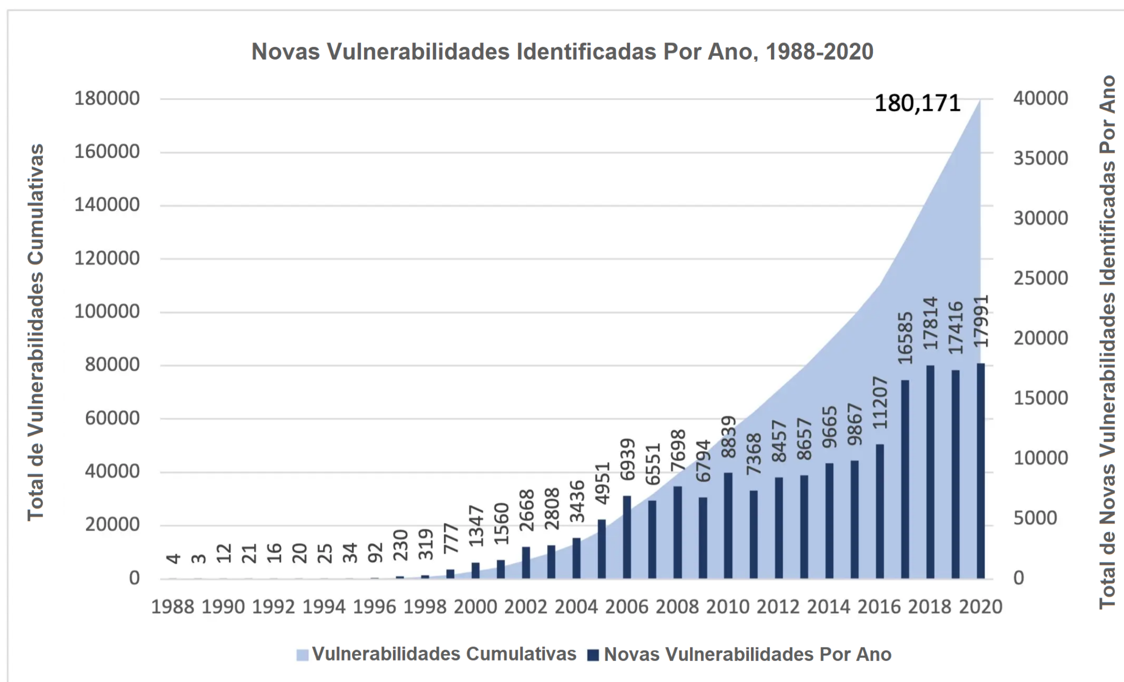


Fonte: National Museum of American History, Smithsonian Institution (WA),2022.

À medida em que o número de famílias e empresas que fazem uso de computadores pessoais continuam a crescer, o número de *bugs* reportados tem aumentado consideravelmente e, com isso, aumenta-se também as preocupações acerca da segurança dos *softwares* utilizado. De acordo com o NVD, o número de vulnerabilidades de *softwares* reportado cresceu de 25 em 1995 para quase 5000 em 2005 ((SPARKS et al., 2007) e (NIST, 2021)). Na Figura 2, pode-se verificar facilmente essa tendência do aumento de relatórios de falhas de segurança, assim como o aumento esperado para incidentes de segurança correlatos.

Dentre esses *bugs*, existem aqueles que são considerados críticos, ou seja, mais propensos a desencadear eventos que podem vir a comprometer a segurança da infraestrutura sobre a qual executam e por isso, passam a ser considerados vulnerabilidades. Essas vulnerabilidades perigosas, por sua vez, permitem que atacantes executem código arbitrário, escalem privilégios, corrompam estruturas de dados sensíveis ou até mesmo, assumam todo o controle do sistema sobre o qual o programa executa (CHEN et al., 2018). Segundo (JUUSO; TAKANEN, 2011), pode-se afirmar que vulnerabilidades não são cri-

Figura 2 – Tendências acompanhadas no aumento de relatórios de vulnerabilidades e incidentes, ao longo de 32 anos.



Fonte: (Data Privacy Manager, 2023).

adas no momento em que o sistema está sendo atacado e sim, são os mecanismos que possibilitaram o ataque.

Nesse sentido, vulnerabilidades são consideradas como erros de implementação que são introduzidos no momento de desenvolvimento de uma aplicação, sendo causadas em geral por erros do programação. Esses erros, por sua vez, tornam-se vulnerabilidades no momento de *release* da aplicação, quando a superfície de ataque desta aplicação é exposta ao público para ataques. Dessa forma, pesquisadores de segurança, companhias de segurança e hackers acabam por descobrir estas vulnerabilidades e, caso estes agentes decidam reportar seus achados, então isso possibilitará os desenvolvedores de *software* a criarem *patches*[3] para as vulnerabilidades reportadas. Contudo, haverá vulnerabilidades que não seriam descobertas durante os testes ou seus descobridores decidem não reportá-las por qualquer motivo e dessa forma, passam a ser consideradas vulnerabilidades *zero-day* - os fabricantes do *software* não estão cientes da falha e portanto, não estão prontos para lançar *patches* para elas. Uma vez que tais vulnerabilidades *zero-day* são exploradas, inicia-se uma verdadeira corrida contra o tempo para fixá-las, a fim de mitigar[4] um dano

[3]Correções lançadas para corrigir *bugs* reportados em uma determinada aplicação, geralmente na forma de atualizações gratuitas.

[4]No caso, uma vez que a vulnerabilidade *zero-day* já foi explorada, não resta mais nada a fazer, senão a

irreversível nas vendas ou reputação da empresa responsável pelo *software*. Em suma, uma vulnerabilidade pode ser caracterizada como a interseção de três elementos principais: a suscetibilidade do sistema a apresentar uma falha, o acesso a falha por um atacante e a capacidade do atacante de explorar essa falha (CHEN et al., 2018).

2.2 Robustez de *softwares*

Como resultado do crescente aumento do número de relatos de *bugs* e suas consequências por vezes desastrosas, surgiu um grande interesse por parte das indústrias e universidades, no sentido de se desenvolver e aprimorar ferramentas automatizadas para testes de segurança e com isso, surgiu a necessidade de definir-se o conceito de robustez em *softwares*. Segundo (COMMITTEE et al., 1990), essa robustez caracteriza-se como *"o grau através do qual um sistema ou componente pode sustentar o seu funcionamento adequado, quando na presença de entradas indevidas ou seja, quando exposto a um ambiente estressante, para o qual não fora devidamente testado"*.

Neste último caso, interpreta-se o termo "estressante" como qualquer cenário em que se verifique, de maneira proposital, controlada ou não, a capacidade de um programa de computador em se manter estável quando em execução, sem apresentar *bugs*, *crashs*[5], congelamentos ou ainda, que apresente comportamentos que possam vir a comprometer toda a infraestrutura sobre a qual executa. Ainda, de acordo com (BRADEN, 1989), é importante que os *softwares* sejam desenvolvidos para lidar com todos os tipos de erros concebíveis, mesmo que improváveis. Eventualmente, um pacote de dados com uma combinação específica de erros e atributos será recebido, e se o *software* não estiver preparado para lidar com isso, problemas podem surgir. Geralmente, é recomendado assumir que a rede possui entidades mal-intencionadas que enviarão pacotes com o objetivo de causar os piores erros possíveis. Essa premissa leva a um *design* protetivo adequado, mesmo que muitos dos problemas mais graves na internet tenham sido causados por mecanismos desavisados, acionados acidentalmente por eventos de baixa probabilidade.

Segundo (JUUSO; TAKANEN, 2011), pode-se também afirmar que todos os *softwares* que não estão em concordância com as boas práticas de segurança, assim como conceito de robustez (2.2), são exploráveis por atacantes. Cabe apenas aos atacantes en-

mitigação dos problemas causados.

[5]Situação na qual um programa de computador apresenta uma falha crítica, da qual não consegue recuperar sua execução por conta própria.

contrar entradas que provoquem respostas indevidas do sistema que é submetido ao teste (também chamado de SUT[6]), a fim de vislumbrar um ataque possível. Basicamente, qualquer *bug* passível de suscitar *crashs* é passível de ser explorado no processo de ataque a um sistema ou aplicação. Por sua vez, os atacantes enviam entradas mal-formatadas até que o SUT responda de uma certa maneira desejada. Algumas vezes, *bugs* podem ser expostos através de simples entradas individuais, enquanto que, algumas vezes, atacantes tem de comunicar longas sequências especificamente mal-formatadas, a fim de ganhar acesso à camadas mais profundas do SUT. Outras vezes, essas vulnerabilidades podem ainda ser desencadeadas por eventos do próprio sistema sobre o qual a aplicação executa, como, por exemplo, a própria manutenção desse sistema. Portanto, a melhor forma de se garantir a segurança de seus sistemas é desenvolver *software* tendo sempre em vista a concordância com boas-práticas de segurança, buscando e fixando os *bugs* mais críticos, antes que esses passem a serem considerados como vulnerabilidades de segurança.

2.3 Metodologias de análise de vulnerabilidades

Segundo (SPARKS et al., 2007), a análise de vulnerabilidades envolve a descoberta de um subconjunto do espaço de entradas, nas quais um usuário malicioso pode explorar erros de lógica em uma aplicação que, por sua vez, levarão esta aplicação a um estado inseguro. À medida em que um *software* se torna maior e mais complexo, a exploração de todo o espaço de estados de uma aplicação comercial torna-se um problema intratável. Com isso, a redução do escopo de exploração torna-se fundamental e dessa forma, pesquisadores de segurança tem desenvolvido numerosas metodologias de testagem. Segundo (ERNST, 2003), (INTEL, 2021) e (JUUSO; TAKANEN, 2011), embora essas metodologias existam em grande número, é possível classificá-las em duas grandes vertentes: análise estática e análise dinâmica.

- **Análise Estática**: Também conhecida como análise estrutural, *white-box* ou "teste de caixa-branca", consiste na análise detalhada do SUT através de seu código-fonte, além possivelmente de especificações de *design*, *disassembly* estático e informações providas em tempo de compilação, se disponíveis - quanto mais informação acerca da aplicação, maior é a eficiência possível deste tipo de processo. Baseia-se na presunção de que o testador possui todo o conhecimento interno possível do SUT,

[6]System Under Test ou Sistema Sob Teste, em português. No caso, é a aplicação/sistema-alvo de qualquer tipo a ser testada pelo *fuzzer*.

quando no momento de geração de casos de testes (SPARKS et al., 2007). Ainda, segundo (MCNALLY et al., 2012), nesse tipo de análise, o testador caso deve possuir um entendimento relativamente completo das operações internas do SUT.

Segundo (JUUSO; TAKANEN, 2011), um dos problemas com a análise estática é a de reportar *bugs* que o *software* muitas vezes não possui, ou que não possuem potencial de exploração viável, fato esse que também é chamado de falso-positivo. Devido a isto, torna-se impossível testar todo o sistema, o que leva as ferramentas de testagem relacionadas a realizarem algumas deduções mais ou menos aproximadas, o que leva seu resultado a imprecisões. Falsos-positivos são problemáticos, devido à sua verificação de validade consumir muitos recursos. Portanto, algumas ferramentas têm adotado heurísticas ou aproximações estatísticas para reduzir a quantidade de falso-positivos que passarão para a fase de análise e seleção de *bugs* realmente problemáticos, sem comprometimento com a cobertura dos testes. Como efeito, essas técnicas podem aprimorar ainda mais a habilidade de capturar vulnerabilidades críticas.

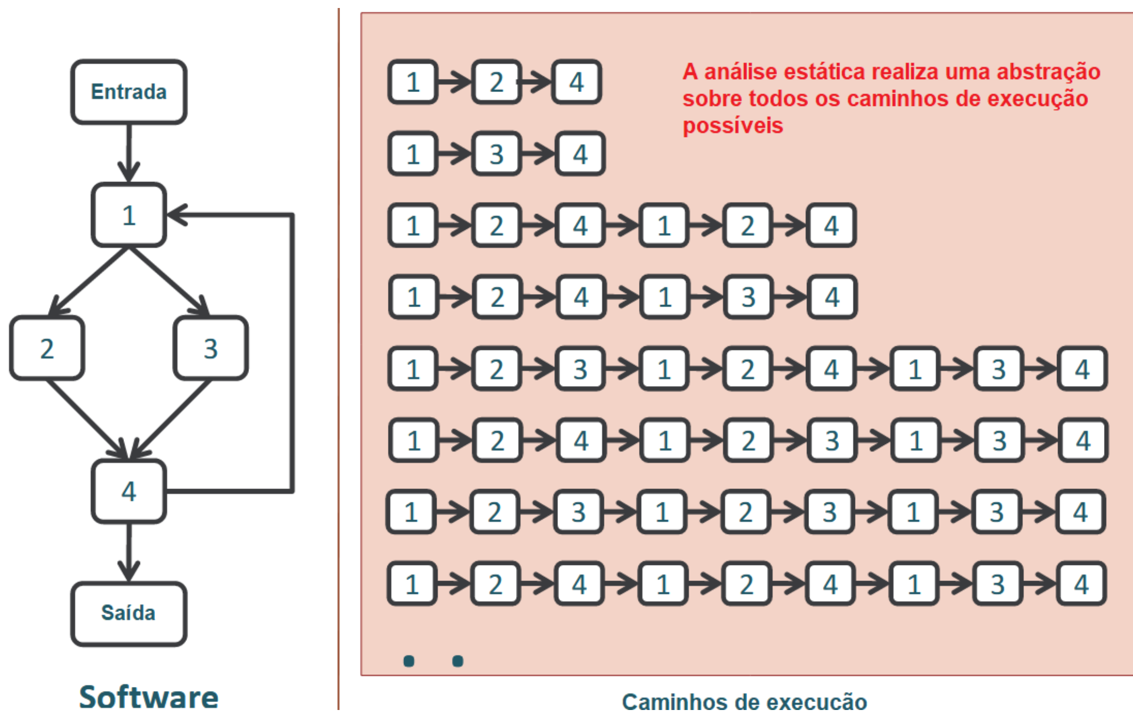
Com relação à cobertura de código (JUUSO; TAKANEN, 2011), ferramentas de análise estática são capazes de identificar práticas inadequadas de programação e estruturas de código arriscadas. Seu propósito no SDL[7] seria o de auxiliar o processo de revisão de código a fim de garantir a conformidade com políticas de codificação seguras. Sua implementação consiste no exame minucioso de todo código-fonte do SUT, em busca de correspondências sintáticas de problemas de codificação comuns, através do uso de regras e padrões. Ferramentas mais avançadas de análise estática adotam uma abordagem multi-camadas para análise do código-fonte, combinando análise sintática e semântica com outras técnicas de análise se disponíveis, o que pode incluir a identificação de "códigos-espaguete"[8], análise de dependências e acoplamento, questões e métricas acerca de *threading*, etc. Todas essas questões podem ser vinculadas à questão de segurança do sistema.

A Figura 3 ilustra a representação da cobertura de testes provida pela análise de código estática. Nessa abordagem, o SUT é analisado diretamente através do código-fonte da aplicação. Dessa forma, *bugs* e possíveis falso-positivos são encontrados tanto nas abstrações de fluxos de execução visíveis quanto diretamente no código,

[7]Software Development Lifecycle, ou Ciclo de vida de desenvolvimento de *software*, é o processo que contempla desde a concepção até a implementação de um produto digital qualquer.
[8]Código de computador que segue pouco ou não segue nenhuma das boas-práticas de programação.

pela identificação de padrões de codificação vulneráveis. Ainda, também é possível verificar como que como que ferramentas de análise estática são capazes de encontrar *bugs* em todo o *software*, ainda que nem todos esses *bugs* sejam considerados críticos ou prováveis de serem explorados por um potencial atacante.

Figura 3 – Cobertura de testes provida pela análise de código estática.



Fonte: (MITCHELL, 2023)

- **Análise Dinâmica**: Conhecido como testagem funcional, *black-box* ou "teste de caixa-preta", consiste na testagem e a avaliação de uma aplicação em tempo de execução, sem depender tanto do código-fonte do SUT ou do seu *disassembly* em tempo de execução. Ao invés disso, baseia-se na alimentação de dados randômicos ou semi-randômicos diretamente na entrada do SUT[9] e, então, monitora seu comportamento a fim de buscar comportamentos inesperados que podem vir a surgir, como consequência destas entradas propositalmente mal-formatadas. *fuzzers black-box* têm se tornado popular nos anos recentes devido ao custo-benefício favorável, a sua simplicidade e potencial de automação. Ainda, segundo (MCNALLY et al., 2012), nesse tipo de análise, considera-se que o SUT é uma *black-box*, ou seja, que apenas as suas entradas e saídas podem ser determinadas, sem nenhuma pista inicial acerca dos estados e da implementação interna do SUT.

[9]Em tempo de execução, a entrada do SUT é também chamada de "superfície de ataque".

Segundo (JUUSO; TAKANEN, 2011), a vantagem da análise dinâmica consiste no fato de que, diferentemente da análise estática, não há tantos falso-positivos: cada *bug* é descoberto mediante um ataque simulado. Portanto, é maior a probabilidade de que esses *bugs* representem ameaças de segurança mais consistentes. Com relação à cobertura de código (SPARKS et al., 2007), o que *fuzzers black-box* frequentemente não dispõem, é de uma seleção de entradas para o SUT baseada em um *feedback* acerca do progresso alcançado por cada uma dessas entradas, do ponto de vista da lógica da execução do SUT. Na prática, pesquisadores de segurança frequentemente se deparam com situações nas quais estes analisaram e identificaram uma localização potencialmente explorável no SUT e essa localização, por sua vez, tem sua alcançabilidade dependente apenas de uma entrada controlada pelo usuário. Um exemplo prático simplificado, seria um pacote recebido através de uma conexão de rede, que então é enviado a alguma função de API que por sua vez, é reconhecidamente vulnerável a *buffer-overflows*.

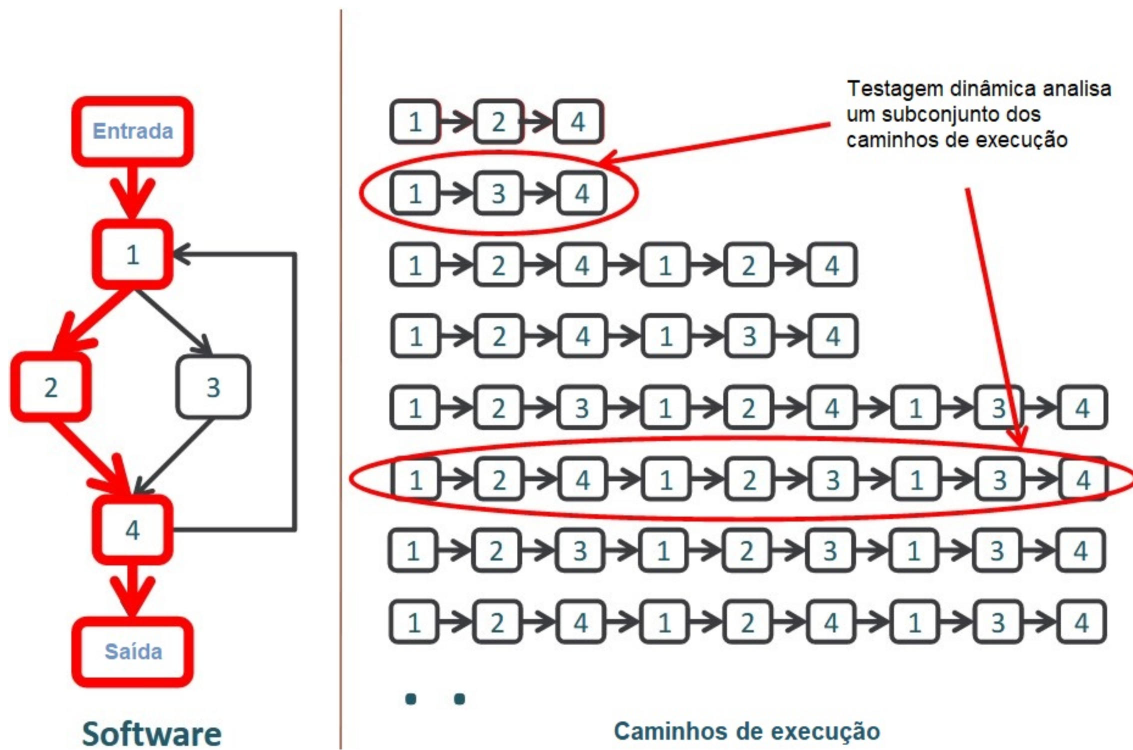
Na Figura 4, por sua vez, pode-se conferir a representação da cobertura de testes provida pela análise de código dinâmica. Nessa outra abordagem, o SUT é analisado através de entradas fornecidas em tempo de execução. Dessa forma, *bugs* são encontrados diretamente, em tempo de execução, o que elimina a probabilidade de se reportar falso-positivos. Ainda, segundo (JUUSO; TAKANEN, 2011), em contraste com a análise estática, técnicas de *fuzzing* apenas são capazes de encontrar *bugs* que, por sua vez, estejam acessíveis em tempo de execução, ou seja, ao longo de sua superfície de ataque exposta.

Explorar uma falha, contudo, implica em alcançabilidade da seção de código vulnerável. Ou seja, a fim de se determinar se uma vulnerabilidade é de fato uma ameaça explorável, é necessário que antes disso garanta-se a alcançabilidade do recurso / trecho de código a ser explorado, mediante uma entrada específica. O formato exato dessa entrada, dependerá exclusivamente da lógica subjacente ao controle de fluxo do programa - exclusivamente, do processo entre o *parsing* da entrada fornecida e a chamada do trecho de código a ser analisado. A Figura 5 nos provê uma abstração gráfica dessa ideia: nela, pode-se ver um exemplo de exploração da função ***recv***[10], a qual recebe, propositalmente, uma entrada mal-formatada com o objetivo de explorar a função ***strcpy***[11] que, por sua vez, é bem conhecida por ser vulnerável

[10]Exemplo de implementação disponível em: https://learn.microsoft.com/en-us/windows/win32/api/winsock/nf-winsock-recv.

[11]Exemplo de implementação disponível em: https://www.ibm.com/docs/en/i/7.4?topic=functions-strcpy-

Figura 4 – Representação da cobertura de testes provida pela análise de código dinâmica.

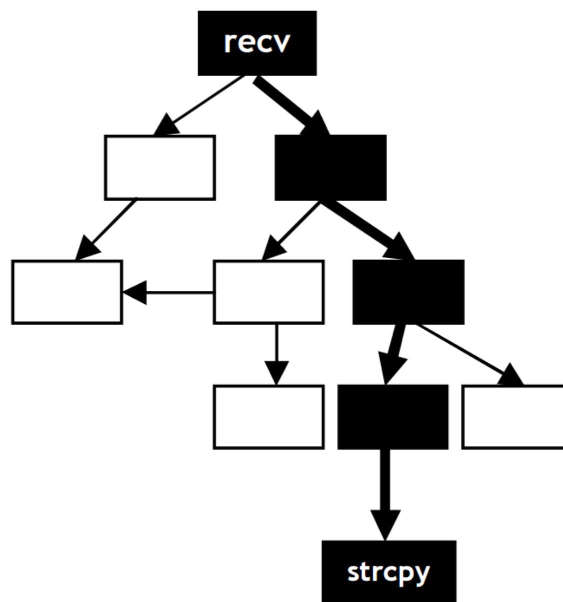


Fonte: (MITCHELL, 2023)

a erros de formatação. É através dessa abordagem que muitos *fuzzers* de aplicação descobrem pontos de entrada vulneráveis em seus alvos.

copy-strings.

Figura 5 – Diagrama idealizado para a exploração de um programa C vulnerável.



Fonte: (SPARKS et al., 2007)

Por fim, segundo (SPARKS et al., 2007), verifica-se que ambas as metodologias de análise são complementares, uma vez que a cobertura de código de nenhuma delas é capaz de, isoladamente, detectar todos os possíveis cenários vulneráveis em uma aplicação.

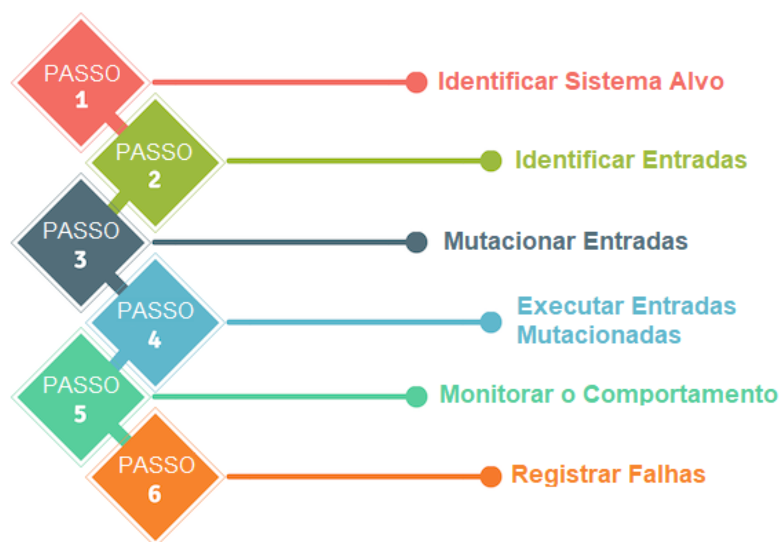
2.4 História e motivações do *fuzzing*

A análise de *softwares* com técnicas de *fuzzing* não é algo novo (BOEHME; CADAR; ROYCHOUDHURY, 2021), tendo suas prováveis origens na universidade de *Wisconsin* por volta de 1988, como parte de um projeto na disciplina de sistemas operacionais. Mais especificamente, o acadêmico inventor da técnica queria descobrir o motivo pelo qual o ruído elétrico de tempestades, que ao se propagar pelas linhas telefônicas, afetava terrivelmente a funcionalidade dos seus programas utilizados quando estava conectado à internet. A proposta se revelou muito mais promissora do que o esperado e com isso, surgiu o conceito de *fuzzer*, cuja proposta é de um programa para testes de segurança em implementações abertas ou fechadas, injetando dados semi-randômicos a fim de desencadear uma condição vulnerável no fluxo de execução esperado do programa-alvo e, dessa forma, encontrar *bugs* de maneira total ou parcialmente automatizada. O sistema mais simples de *fuzzing* deve consistir, basicamente, de uma geração de entradas randômicas

e da alimentação dessas entradas para o SUT, além do monitoramento do SUT mediante essas entradas e do respectivo *logging* de possíveis erros, conforme pode-se conferir na Figura 6:

No passo 1, define-se o sistema a ser testado; No passo 2, tem-se a identificação da superfície de ataque; No passo 3, gera-se os dados mal-formatados de acordo com a aplicação; No passo 4, executa-se a aplicação com os dados mal-formatados no passo anterior; No passo 5, monitora-se o comportamento da aplicação e no passo 6, é feito o *logging* de possíveis *crashs*. Dependendo da natureza do SUT, métodos mais complexos podem levar ao desenvolvimento de um modelo de *fuzzer* mais eficiente, contudo modelos mais básicos são relativamente simples de serem desenvolvidos, especialmente nos casos de uso onde o *fuzzer* é suscetível a uma abordagem mutacional ou ainda, é possível valer-se de algum *framework* de *fuzzing* que se revele mais adequado ao SUT ((CHEN et al., 2018) e (MCNALLY et al., 2012)).

Figura 6 – Fluxo de trabalho clássico de um *fuzzer*.



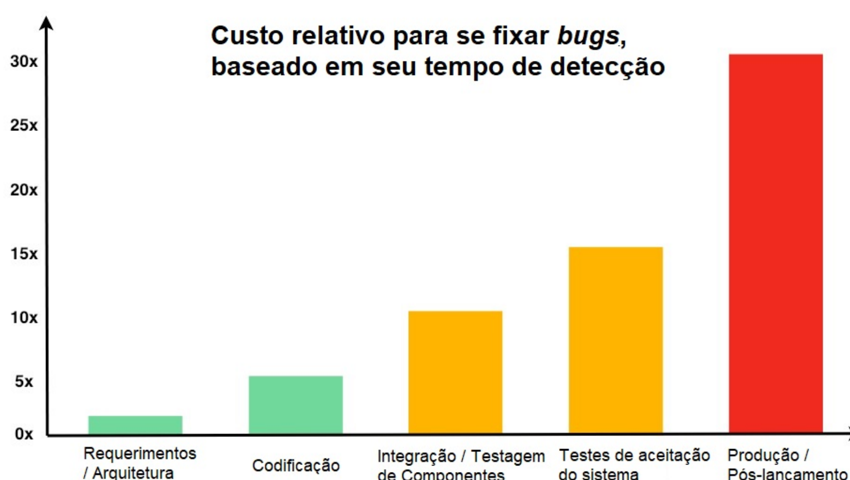
Fonte: (TAVARES, 2020)

Desde então, conceito de *fuzzing* vem sendo amplamente utilizado por setores de Controle da Qualidade em empresas de desenvolvimento de *software*, já sendo uma parte essencial do ciclo de desenvolvimento de seus produtos, enquanto que, por outro lado, *hackers* e pesquisadores de segurança tem cada vez mais se valido técnicas de *fuzzing* para a exploração de *softwares* comerciais fechados (ou seja, sem código-fonte ou informações de depuração disponíveis), o que por muitas vezes acaba por possibilitar diretamente a escrita dos seus *exploits*[12] correspondentes. Atualmente técnicas de *fuzzing*

[12] Trecho de código responsável por suscitar um comportamento vulnerável, em uma aplicação específica

estão, são mais utilizadas pelas maiores empresas de desenvolvimento da atualidade, tais como Google (ZERO, 2014), Adobe (ADOBE, 2021), IBM (IBM, 2021), Apple (APPLE, 2019), Cisco (CISCO, 2017), Microsoft (MICROSOFT, 2020a) (MICROSOFT, 2020b). Essas empresas, consideram o fato evidente de que a adoção de técnicas de *fuzzing* representa um grande retorno sob o seu investimento inicial, já que o custo relacionado à detecção e correção de *bugs*, em tempo de desenvolvimento, é muito menor quando comparado ao cenário em que tais *bugs* são descobertos e explorados após o *release* da aplicação (Figura 7), além da capacidade dessa técnica de reduzir a superfície de ataque para *blackhats*[13].

Figura 7 – Custo relativo para se fixar *bugs*, com base em seu tempo de detecção.



Fonte: Adaptado de: (MICROSOFT, 2023a))

No contexto do mercado do trabalho, percebe-se que as oportunidades para os profissionais que possuem expertise em tais técnicas são muito amplas. Considerando-se a clara vantagem na capacidade de encontrar e corrigir *bugs* em seus produtos em tempo de produção, pode-se considerar hoje que técnicas de *fuzzing* são parte essencial no ciclo de vida do desenvolvimento de qualquer aplicação comercial da atualidade. Na Figura 8 verifica-se que a implementação do SDL[14] da Microsoft incorpora técnicas de *fuzzing*, como parte da etapa de verificação de erros em seus produtos.

Além disso, em vista da crescente onda de *cyberataques* que diversas empresas ao redor do mundo inteiro tem reportado, o investimento em segurança digital nunca foi tão grande quanto nos últimos anos e, com isso, aumentou também a demanda por pro-

para qual foi escrito.

[13]Hackers antiéticos.

[14]*Software Development Lifecycle* - Ciclo de Desenvolvimento de Software de uma aplicação.

Figura 8 – SDL atualmente adotado pela Microsoft, onde verifica-se o uso de técnicas de *fuzzing* durante a etapa de Verificação dos seus produtos.

Fonte: Adaptado de: (MICROSOFT, 2023b)

fissionais do ramo. Empresas como a Zerodium chegam a pagar até US$ 2,5 milhões por *exploits zero-day* (aqueles para os quais ainda não existem *patchs*) que violem mecanismos de segurança de sistemas operacionais e aplicações modernas. Um exemplo, prático e atual, foi o domínio de tais técnicas que permitiu agências como a NSA[15] desenvolver sua nova geração de cyberarmas, como o exploit Eternal Blue que visa especificamente falhas de implementação do protocolo SMB[16] em sistemas WinNT da atualidade ou ainda, o malware Stuxnet, cujo objetivo era atingir especificamente as controladoras das centrífugas de enriquecimento de urânio Iranianas em 2003 (CHEN; ABU-NIMEH, 2011). Ainda, segundo (CHEN et al., 2013), dentre as diversas técnicas utilizadas em testagem automatizada para se encontrar erros de programação e geração de casos de teste, ferramentas de testagem semi-randômicas agressivas como os *fuzzers*, têm se revelado impressionantemente efetivas. Contudo, segundo (MCNALLY et al., 2012), técnicas de *fuzzing* são apenas um dos muitos métodos que são utilizados para se descobrir falhas dentro de um sistema ou *software*, uma vez que diversos outros métodos igualmente relevantes também existem, tais como: revisão de código-fonte, análise estática, testes unitários, verificação de modelo, *beta testing*. Tendo em vista a abundância de metodologias existentes, é válido esclarecer quais foram as motivações para a ampla adoção de técnicas de *fuzzing* ao longo das últimas décadas, assim como definir em quais casos de uso o uso de *fuzzing* é conveniente ou não.

[15]National Security Agency - Agência de Segurança Nacional norte-americana.
[16]Server Message Blocks - Blocos de Mensagem do Servidor, protocolo da Microsoft comumente utilizado para compartilhamento de arquivos em rede.

2.4.1 Vantagens do *fuzzing*

Nessa subseção são examinadas algumas das vantagens mais conhecidas ao se adotar técnicas de *fuzzing* no processo de análise de segurança em aplicações modernas. São analisados diversos fatores acerca dessa técnica, tais como: Eficiência dessa técnica para exposição de falhas, dependência de código-fonte para a realização desse tipo de análise, custos envolvidos, alcance desse tipo de técnica, portabilidade, facilidade de implementação em sistemas distribuídos e escalabilidade.

- **Eficiência para exposição de falhas:** Como mencionado anteriormente, a geração automatizada de entradas pode inicialmente parecer uma abordagem contraintuitiva para se descobrir falhas; a história tem demonstrado que essa abordagem revela-se surpreendentemente viáveis, quando na descoberta de erros mais sutis que, por sua vez, não foram detectados em outras abordagens. Segundo (GODEFROID; LEVIN; MOLNAR, 2012), quase um terço dos *bugs* no Windows 7 entre 2007 e 2019 foram descobertos pelo *fuzzer* SAGE[17], um fato considerável, visto a quantidade de *bugs* que não foram cobertos pelo processo de controle de qualidade de uma empresa já bem estabelecida no ramo de desenvolvimento.

- **Não necessita do código-fonte:** Uma das áreas, nas quais a testagem por *fuzzing*, ganhou notoriedade dentro da comunidade *blackhat*, foi devido ao fato de viabilizar a busca por vulnerabilidades *zero-day* em *softwares* comerciais fechados. Diferente de muitas outras técnicas, *fuzzing* não requer acesso ao código-fonte do SUT, o que quase sempre é o caso para as tecnologias que são vendidas. A falta desse código-fonte restringe ou inviabiliza totalmente as técnicas e a superfície de análise coberta em abordagens como análise estática e verificação de modelos. Ainda, no processo de compilação de *softwares* comerciais, costuma-se remover informações simbólicas de depuração, que servem especialmente para o processo de produção do código e, portanto, são removidas na versão final de *release*. A falta dessas informações dificulta o processo de depuração para *blackhats* e podem até mesmo frustrar completamente a análise do programa.

- **Testes manuais são caros:** Historicamente, verifica-se que equipes de controle da qualidade tem dependido quase que predominantemente sobre conjuntos de casos

[17] Scalable Automated Guided Execution, ferramenta de *fuzzing* da Microsoft disponível em (GODEFROID; LEVIN; MOLNAR, 2008).

de teste desenvolvidos manualmente. A testagem de *software* tradicional é voltada para uma "abordagem positiva" (ou seja, considerando-se os casos em que os recursos comportam-se como o esperado), ao invés de uma "abordagem pessimista" (ou seja, considerando-se os casos onde o sistema faz coisas que supostamente não deveria estar fazendo) e com isso, reduz-se consideravelmente a área de cobertura das falhas que podem ser encontradas. Em suma, *fuzzing* é um método com um bom custo-benefício para uma testagem pessimista, com uma boa área de cobertura de código. Como tal, técnicas de *fuzzing* são frequentemente utilizadas para complementar suítes de testes manuais, fazendo uso de um *fuzzer* que é apto a gerar automaticamente uma grande quantidade de casos de teste que por sua vez, irão sondar por assunções inseguras dentro do sistema.

- **Testadores humanos têm pontos-cegos:** As práticas de engenheiros de *software* tem demonstrado que a produção de casos de testes para seus programas, quando produzidos por outros programadores, são mais efetivas, uma vez que pontos cegos na implementação tendem a serem replicados nos testes. Segundo (MYERS; SANDLER; BADGETT, 2011), deve-se atentar a dois princípios: primeiro, o de que desenvolvedores devem evitar testar os seus próprios programas; segundo, o de que companhias de desenvolvimento não deveriam testar seus próprios programas. Para que testadores humanos sejam efetivos, eles também precisam compreender a implementação do sistema que testam, incluindo suas restrições de segurança e casos-limites. Essa abordagem é onerosa de todas as formas possíveis, além de exigir que dois times separados entendam as operações e regras de negócio.

- **Portabilidade:** Em muitos casos, *fuzzers* não estão restritos a um SUT específico e portanto, podem ser aplicados a outros sistemas. Por exemplo, um *fuzzer* para o protocolo HTML[18] pode muito bem ser utilizado para testagem de diferentes navegadores, incluindo-se diferentes versões e fabricantes;

- **São distribuíveis:** Trabalhos como (JANG et al., 2022), (LI; FENG; TANG, 2018) e (ZHOU et al., 2020) demonstram a possibilidade de valer-se de sistemas de computação distribuída em sistemas *fuzzing*, tanto a fim de valer-se da capacidade de computação de sistemas heterogêneos, quanto para se se alcançar um aumento de vazão do *fuzzer*. Ainda, ferramentas como ClusterFuzz[19] permitem a execução do

[18] Hypertext Markup Language - Linguagem de Marcação de Texto.
[19] Disponível em https://github.com/google/clusterfuzz.

processo de *fuzzing* através de sistemas distribuídos;

- **São escaláveis:** Devido a sua característica modular (veja Figura 9), *fuzzers* podem ser otimizados em termos dos seus componentes individuais, a fim de se alcançar um aumento na vazão da aplicação como um todo. Por exemplo, em trabalhos como (SEAL, 2016), verifica-se a possibilidade do uso de estratégias como algoritmos genéticos para se alcançar um aumento de vazão para *fuzzers* baseados-em-geração 2.6.2, cuja performance está diretamente atrelada a sua capacidade de gerar entradas válidas, no menor tempo possível. Ainda, observa-se que *fuzzers* evolucionários tais como o AFL (utilizado nesse trabalho) (LCAMTUF.COREDUMP.CX, 2021) fazem uso de estratégias heurísticas em diferentes módulos, a fim de se obter ganho de performance.

2.4.2 Desvantagens do *fuzzing*

- **Podem ser redundantes:** *fuzzers* podem ser frustrantes no sentido de que, com frequência, estes encontram repetidamente falhas que, ou não têm severidade o suficiente para serem devidamente exploradas, ou não revelam o suficiente sobre o motivo de tais falhas para que o desenvolvedor as corrija. Atualmente, usuários de *fuzzers* mais sofisticados filtram os casos de teste gerados utilizando métodos *ad-hoc*, tais como desabilitar recursos problemáticos nos testes e filtrar os resultados manualmente. Essa filtragem por sua vez, é feita tendo em vista o fato de que, dado um número potencialmente grande de casos de teste que engatilham falhas, é possível ordenar esses casos de forma que estes sejam ranqueados com base em uma métrica de exploração que se revele mais conveniente ao usuário do *fuzzer*.

- **São computacionalmente custosos:** Segundo (GODEFROID; LEVIN; MOLNAR, 2008), dependendo de qual estratégia de geração de entradas para o SUT for adotada, *fuzzers* podem se revelar extremamente ineficientes, especialmente se tratando de *fuzzers* puramente mutacionais, os quais dependem quase que exclusivamente da sua capacidade de permutação de entradas para o SUT. Ainda, caso essas entradas obedeçam a uma lógica estrita, como *checksums*, situação que impacta consideravelmente ou até mesmo inviabiliza qualquer estratégia de *fuzzing*, a menos que o algoritmo de *checksum* seja extraído do SUT e então incorporado ao processo de *fuzzing* .

- **São conceitualmente limitados:** A fraqueza talvez ainda irremediável de *fuzzers* clássicos é a conceituação do que uma "falha" realmente significa. Por exemplo, *fuzzers* típicos não são capazes de detectar violações de acesso para usuários não-autenticados, quando estes acessam recursos que apenas deveriam estar disponíveis em um processo de *logging* válido. Ao invés disso, *fuzzers* estão tipicamente limitados a detectar situações nas quais a aplicação sofreu *crash* ou travou. Dessa forma, o que existem são várias possibilidades para aumentar a área de cobertura que pode ser oferecida por um *fuzzer*, incluindo suporte aprimorado para detecção de falhas de execução, suporte para um espectro mais amplo de classes de erros e melhorias no algoritmo de validação para os diferentes tipos de estados internos possíveis do SUT.

2.5 Anatomia de sistemas *fuzzing*

Ao longo dos últimos 30 anos, sistemas de *fuzzing* foram aprimorados e incorporaram cada vez mais técnicas, o que algumas vezes, acabou por torná-los excessivamente complexos. Contudo, a estrutura-geral de sistemas de *fuzzing* pode ser decomposta em basicamente três componentes principais segundo (CHEN et al., 2018) e (MCNALLY et al., 2012):

- **Gerador de entradas**: Conhecido por "gerador de casos de teste", é responsável pela geração das amostras de entradas que serão utilizadas para se testar a aplicação-alvo (SUT). Este, por sua vez, realiza mutações distintas e apropriadas de diversas entradas de testes para o SUT, ou para quaisquer um de seus componentes (bibliotecas, classes, funções, etc). A amostra de testes que é gerada (a saída do gerador) pode ser, por exemplo, um tipo de arquivo ou um *stream* de *bytes* em um fluxo de rede. O gerador podem se utilizar de uma variedade de estratégias de mutação para essa geração de amostras, a fim de aumentar sua eficiência sobre um determinado tipo de aplicação.

- **Sistema de alimentação**: É responsável por recolher as entradas de teste produzidas pelo gerador e, então, alimentar o SUT para processamento. O sistema de alimentação está diretamente em função da natureza esperada da entrada pelo SUT. Por exemplo, o mecanismo de alimentação de uma aplicação que consome um arquivo de entrada, é diferente do mecanismo utilizado por uma aplicação que con-

some um *stream* da rede.

- **Sistema de monitoramento**: É responsável pelo monitoramento e coleta de informações das instâncias em execução do SUT, conforme cada instância destas consome as entradas que vão sendo fornecidas, a fim de se detectar quaisquer erros que possam suscitar desta ação. Tais erros, por sua vez, podem revelar informações importantes acerca da estrutura do SUT e talvez, determinar se um *bug* foi efetivamente encontrado e para tal, já se encontram disponíveis ferramentas na literatura com esse propósito (HASTINGS, 1991). Contudo, o processo de se filtrar, dentre os *bugs* encontrados, aqueles mais ou menos passíveis de exploração[20] costuma ser feito de maneira manual, visto a complexidade de se identificar esses pontos vulneráveis, mas já existem ferramentas atuais que servem bem ao propósito de se determinar a superfície de ataque e a severidade dos *bugs* encontrados. Sistemas de monitoramento são uma parte crítica no processo de *fuzzing*, com um reflexo direto sobre o tipo de falhas que o *fuzzer* em questão estará apto a revelar.

Na Figura 9, verifica-se a estrutura de um sistema de *fuzzing* genérico, na qual constam os três elementos supracitados, dentre outros.

2.6 Taxonomia de sistemas *fuzzing*

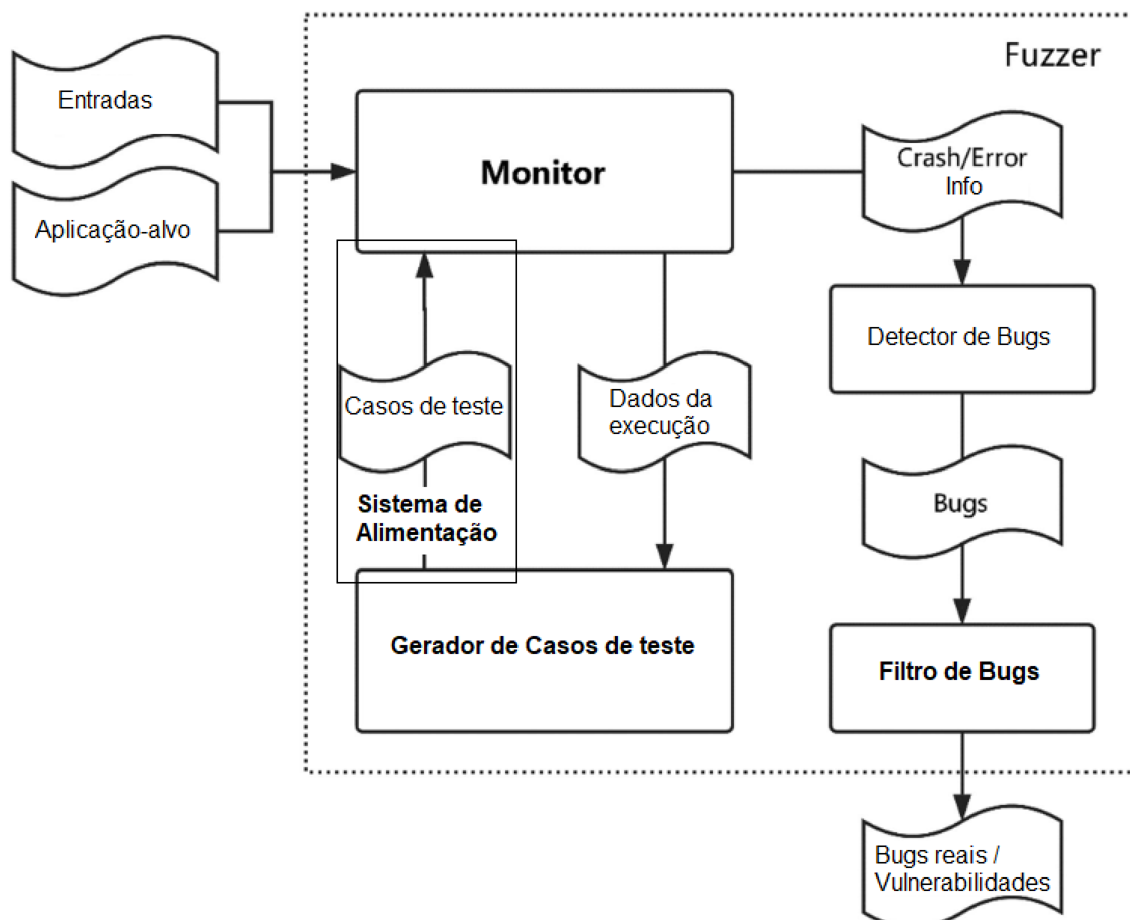
Devido ao amplo espectro de atuação que a tecnologia de *fuzzing* permite abordar, suas diversas implementações podem ser classificadas sob múltiplas perspectivas. Ainda que *fuzzers* sejam especificamente projetados para a exploração de um tipo específico de aplicação, segundo múltiplos autores ((MCNALLY et al., 2012), (CHEN et al., 2018) e (WU et al., 2022)), sistemas de *fuzzing* podem, no geral, serem bem classificados sob o princípio pelo qual efetivamente exploram falhas e recolhem informação de *feedback* de seus alvos. Essas classificações mais comuns são as que seguem:

2.6.1 Baseados-em-mutação

Um *fuzzer* mutacional recebe como entrada dados fornecidos pelo usuário (comumente chamado de *seed*) e então, realiza mutações semi-randômicas sobre essa entrada.

[20]Ou seja, passível de sofrer com a ação de *exploits*.

Figura 9 – Diagrama arquitetural de um sistema de *fuzzing* clássico.



Fonte: (MCNALLY et al., 2012)

Essa mutação, por sua vez, vai gerar as entradas a serem testadas na aplicação-alvo, que após serem triadas mediante algum critério, podem ou não vir a serem descartadas logo no começo da execução. Tal critério poderia ser, por exemplo, o excesso de dissonância com o formato de entrada esperado do SUT, o nível de propensão dessa entrada de desencadear um determinado tipo de comportamento faltoso ou ainda, a capacidade dessa entrada de provocar a execução de uma linha específica no código do SUT. *Fuzzers* baseados em mutação, no geral, requerem muito menos esforço de programação do que os baseados-em-geração, uma vez que não requerem nenhum esforço de compreensão da sintaxe esperada para protocolos e interfaces por exemplo, dependendo dessa forma quase que exclusivamente de um esforço computacional. A exceção dessa regra é, por exemplo, quando a mutação de um determinado tipo de arquivo (*.ISO* ou *.rar*, por exemplo) envolve algum tipo de cálculo de *checksum* que deve ser re-inserida nos metadados do arquivo, a fim de evitar o descarte desses por uma aplicação que, provavelmente, irá por padrão validar esses metadados antes de realizar qualquer tipo de processamento sobre

eles. Em (WU et al., 2022) e (CHEN; CHEN, 2018), verifica-se que *fuzzers* modernos tem utilizado estratégias mutacionais em comuns, tais como:

- ***Byte / Bit flipping***: Consiste na inversão de valores lógicos de um determinado grupo de *bits* ou *bytes*, dentro dos dados de entrada do SUT. Esta técnica é fundamental porque permite testar como diferentes variações de dados podem afetar a estabilidade e segurança de um *software* qualquer. Com isso, o *fuzzer* pode descobrir vulnerabilidades ocultas que só se manifestariam de outra forma sob condições de dados inesperadas ou anômalas. Esta abordagem é especialmente eficaz para identificar problemas em locais onde o *software* espera dados em formatos específicos ou restritos, pois mesmo uma pequena mudança pode resultar em comportamentos inesperados. A simplicidade e eficiência dessa técnica a torna uma técnica fundamental no arsenal de ferramentas de *fuzzing*;

- **Inserção de valores interessantes**: Certos valores são reconhecidos pela sua capacidade de suscitar erros no processo de *parsing* das aplicações, tais como os valores máximo/mínimo de uma variável do tipo inteiro (cujo tamanho em *bytes* depende da implementação), o que pode induzir a erros aritméticos ou, ainda, erros do tipo *off-by-one*. Um erro (ou *bug*) *off-by-one*, é um erro lógico que decorre de um laço iterativo que, por qualquer razão, itera uma vez a mais ou a menos do previsto. Dessa forma, escolhe-se randomicamente alguns desses valores a partir de uma lista e, então, insere-se tais valores no corpo de testes do SUT;

- **Inserção de valores randômicos**: Consiste na escolha de valores randômicos a serem inseridos no corpo de testes que é consumido pelo SUT;

- **Incremento / Decremento aritmético**: Consiste no incremento ou decremento de certos valores aritméticos;

- **Clonagem / Inserção de *bytes***: Consiste na cópia e inserção de blocos de tamanho randômico no corpo de testes do SUT, que por sua vez, acaba por aumentar o tamanho do corpo de testes;

- **Sobrescrita / Deleção de *bytes***: Consiste na deleção e sobrescrita de blocos de tamanho randômico do corpo de testes do SUT, que por sua vez, diminui o tamanho do corpo de testes.

2.6.2 Baseados-em-geração

Também conhecido como "*fuzzers* espertos", um *fuzzer* geracional produz entradas com base em um modelo de entradas (expressões regulares, RFC[21], documentação, etc). Dessa forma, procura-se reduzir drasticamente o número de entradas inválidas, que, de outra forma, seriam descartadas pelo *parser* da aplicação logo de início e portanto, quanto maior o conhecimento do formato das entradas, maior será o rendimento esperado do *fuzzer*. Em contrapartida, *fuzzers* desse tipo possuem um custo maior de tempo de implementação, dependendo diretamente da complexidade relacionada ao formato de entrada da aplicação-alvo, tais como RFC's, documentações, etc. Difere de *fuzzers* mutacionais no sentido de que, ao invés de realizar mutações sobre todo o conjunto de entrada possível, *fuzzers* geracionais inserem pequenas anomalias apenas em seções específicas das entradas produzidas. Seu rendimento (em termos de cobertura de código) costuma estar diretamente relacionado ao nível de conhecimento da formatação esperada das entradas da aplicação-alvo e além disso, diferente do *fuzzer* mutacional, consegue lidar com aplicações-alvo mais bem protegidas, uma vez que conseguem atender a mecanismos de mitigação comuns, tais como *checksums* e dependências externas.

2.6.3 Evolucionários

Esses *fuzzers* são conhecidos como "baseados-em-*feedback*" ou "guiados-por-cobertura", *fuzzers* evolucionários produzem entradas com base em informações que são obtidas em tempo de execução, utilizando a cobertura de código da aplicação-alvo como métrica de desempenho. Ou seja, essa abordagem revolucionária utiliza *feedback* do código do programa alvo para orientar a geração de novos testes, aumentando a probabilidade de descobrir áreas de código não testadas anteriormente e, consequentemente, potenciais vulnerabilidades. Possuem um tempo de implementação superior se comparado aos anteriores, contudo, a despeito de uma cobertura de código consideravelmente maior. *fuzzers* desse tipo geralmente exigem alguma instrumentação binária sobre a aplicação-alvo, devido às técnicas de análise dinâmica que costumam ser implementadas a fim de se aumentar a cobertura de código. Além disso, segundo (DING et al., 2021) esse tipo de *fuzzing* costuma envolver um alto *overhead*, em vista da necessidade instrumentação do

[21] Request For Comments (Requisição por Comentários, em português.), um tipo de especificação formal e bem-conhecida, utilizada para descrição detalhada de uma determinada tecnologia.

código-fonte e de possíveis traduções dinâmicas do código do SUT em tempo de execução.

2.7 O *fuzzer* American Fuzzy Lop (AFL)

Segundo (LCAMTUF.COREDUMP.CX, 2021) e (AFL++, 2023), American Fuzzy Lop (AFL) é um *fuzzer greybox*, ou seja, que pode fazer uso combinado tanto de estratégias de instrumentação do SUT em tempo de compilação, quanto de estratégias para instrumentação de binários para as quais não se possui o código-fonte. Seu princípio de funcionamento consiste na descoberta de casos de teste que levam a novos estados internos de sua aplicação-alvo, através de uma técnica inovadora, fazendo uso tanto de técnicas de instrumentação em tempo-de-compilação. Esses novos estados, por sua vez, aumentam a cobertura de código no binário-alvo. Entre suas principais vantagens, verifica-se a sua facilidade de uso, o que lhe permite ser utilizado sem um conhecimento prévio do domínio ou de sua aplicação-alvo em si. Com isso, o AFL permite a descoberta de vulnerabilidades com um esforço mínimo para analistas de segurança. A centralidade de seu modelo de *fuzzing* guiado-por-cobertura permitiu uma exploração de código mais eficiente e eficaz em comparação com as abordagens anteriores, que eram, em sua maioria, baseadas em estratégias mutacionais 2.6.1. Ainda, trabalhos como (LAKSHMINARAYAN, 2023), (CHEN et al., 2018) e (MCNALLY et al., 2012) e outros demonstram que o *fuzzer* AFL desempenha um papel fundamental na história dos *fuzzers*, introduzindo técnicas inovadoras que remodelaram o cenário do *fuzzing* moderno. O AFL não apenas propiciou uma mudança no paradigma do *fuzzing*, mas também estabeleceu um novo padrão de eficácia no campo da detecção de *bugs*. Portanto, verifica-se que o AFL mantém uma posição de destaque na história dos *fuzzers*, moldando a direção do desenvolvimento de *fuzzers* e estabelecendo uma base sólida para a evolução futura dos estudos nessa área.

2.7.1 Funcionamento e modos de operação

Uma vez que o AFL é um *fuzzer* guiado-por-cobertura (vide seção 2.6.3), faz-se necessário compreender os fundamentos e algoritmos utilizados em seu processo de mapeamento de estados do SUT que ocorre durante o processo de *fuzzing*. Segundo (LI; ZHAO; ZHANG, 2018), para se alcançar uma análise profunda e completa no alvo, *fuz-*

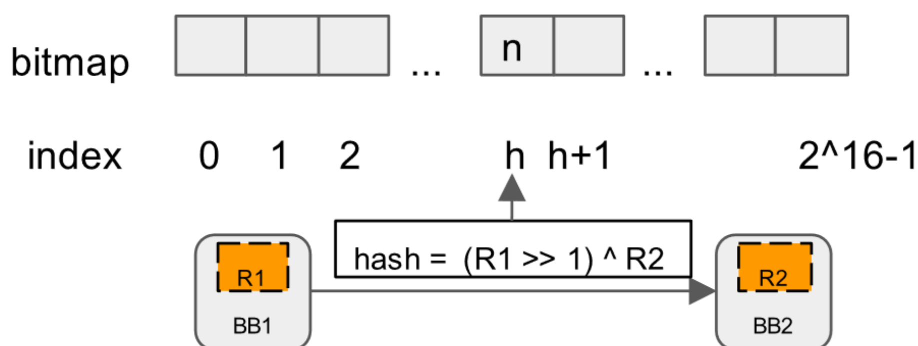
zers devem explorar o maior número de estados possíveis no SUT. Contudo, não existe uma única métrica para os estados do SUT, devido à incerteza inerente no comportamento do SUT (do ponto de vista do *fuzzer*). Dessa forma, a adoção da métrica de cobertura de código se torna uma solução alternativa viável para se mapear novos estados no SUT, sendo facilmente medidade através de instrumentação em tempo-de-compilação ou instrumentação inserida diretamente no binário analisado. Contudo, sabe-se que existe uma certa de perda de informação associada a essa métrica, uma vez que uma cobertura de código constante não implica, necessariamente, em um número constante de estados do SUT.

Dessa forma, durante a análise do SUT, esse é visto como sendo constituído por blocos básicos. Blocos básicos, por sua vez, são trechos de código que possuem um único ponto de entrada e uma saída no fluxo de execução do SUT, tendo suas instruções executadas sequencialmente e uma única vez. No contexto de cobertura de código, diversos *fuzzers* do estado-da-arte (AFL incluso) adotam esses blocos básicos como sendo o melhor nível de granularidade para uma boa análise de cobertura, devido a (dentre outros motivos), o fato de que um bloco básico pode facilmente ser identificado pelo endereço da primeiro instrução que o delimita, sendo essa informação facilmente extraída através de instrumentação do código-fonte do SUT, além de que uma simples contagem de blocos básicos implicaria em uma grave perda de informação importante.

Atualmente existem duas métricas que podem ser adotadas com base nos blocos básicos identificados: contagem de instruções básicas executadas e contagem de transições entre os blocos. Nessa última, o fluxo de execução do SUT é tratado como um grafo, onde cada vértice representa um bloco básico e cada aresta representa uma transição entre cada bloco: No primeiro método são rastreados apenas vértices e no último, apenas arestas. Na Figura 11 pode-se verificar a representação do fluxo de execução de um programa, em termos de blocos básicos (vértices) e transições entre os blocos (arestas). O AFL foi o primeiro *fuzzer* a introduzir esse método de medição de transições de blocos no conceito de *fuzzing* guiado-por-cobertura. Segundo (LCAMTUF.COREDUMP.CX, 2023), o AFL possui dois modos de instrumentação do SUT, no caso, em tempo-de-compilação (modo *greybox*) e instrumentação externa ao SUT (modo *blackbox*, também conhecido como modo sem-instrumentação). No primeiro modo, o AFL provê sua própria versão de compiladores que inserem essa informação para rastreio diretamente no código-fonte SUT; No último, o AFL provê um mecanismo de emulação binária, capaz de inserir informação de instrumentação diretamente no SUT, em tempo-de-execução.

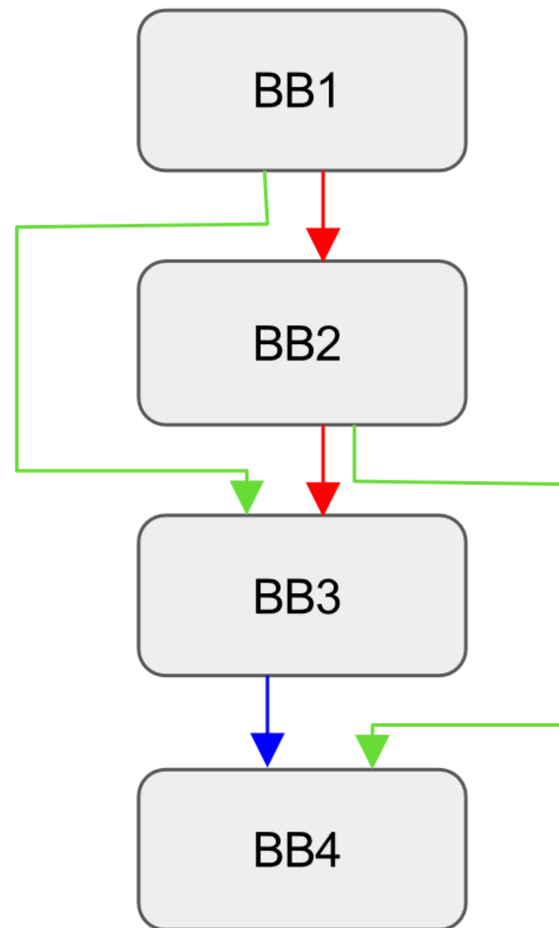
Segundo (LI; ZHAO; ZHANG, 2018) e (PANG et al., 2021), no seu modo de instrumentação (modo *greybox*), um identificador randômico é atribuído a cada bloco básico no SUT, sendo cada transição entre esses blocos mapeada para um *array* de 64KB (podendo ser aumentado), onde cada *byte* desse vetor representa uma transição (aresta) única entre dois blocos. Na prática, um número de *hash* é calculado para cada transição de blocos que é identificada e então, um valor no mapa de *bits* (*bitmap*) é atualizado. Na Figura 10 é representado esse processo de mapeamento de *hashes* no *bitmap* utilizado pelo AFL.

Figura 10 – Representação do processo de mapeamento de arestas de transiçao entre blocos e o *bitmap* que é utilizado pelo AFL.



Fonte: (LI; ZHAO; ZHANG, 2018)

Figura 11 – Representação do fluxo de execução de um programa, em termos de blocos básicos (vértices) e suas transições (arestas).



Fonte: (LI; ZHAO; ZHANG, 2018)

2.7.2 Fluxo de trabalho

Uma vez que o AFL é amplamente conhecido, seu fluxo de trabalho (Figura 13) encontra-se bem representado e descrito por diversos autores, servindo como base para diversos outros *fuzzers*. Segundo (PANG et al., 2021), o fluxo de *fuzzing* realizado pelo AFL pode ser resumido em duas etapas principais:

1. Instrumentação da aplicação-alvo (o SUT nesse caso); Na prática, a função *_afl_-maybe_log* é utilizada para se construir informações de cobertura de código para o alvo;

2. Faz a leitura de entradas iniciais para a fila de entradas através da função *read_-testcases*;

3. Executa a função *perform_dry_run* para testar o arquivo de entrada e garantir que a aplicação executa sem problemas. A função *perform_dry_run*, por sua vez, chama a função *calibrate_case* para testar o arquivo de entrada; Esse processo pode ser visto na Figura 12. O programa então é executado e começa a ser rastreado pela função *run_target()*, dentro do laço de repetição principal do programa (Fim da primeira etapa);

4. Seleciona uma entrada e a simplifica[22]. Chama a função *run_target()* para testar se as entradas simplificadas têm algum impacto no caminho de execução; se não tiverem impacto, salva a assinatura dessa entrada simplificada, com o objetivo de se evitar o tempo de execução gasto para uma entrada já previamente considerada "inútil";

5. Avalia a qualidade de execução do teste realizado, no caso, em termos de métricas como o total de caminhos de execução exercitados no SUT, profundidade nos caminhos de execução alcançados, severidade dos erros suscitados e outras. O AFL também utiliza um algoritmo de "escalonamento de esforço"para determinar a pontuação para uma determinada entrada. Quanto maior a pontuação, mais promissoras são as mutações realizadas sobre essa entrada;

6. Mutaciona a entrada. O AFL utiliza diversos algoritmos de mutação (vide seção 2.6.1) para gerar diferentes casos de teste;

7. Testa e rastreia a execução do SUT. Alimenta o SUT com as entradas previamente selecionadas e monitora o SUT através de sua instrumentação[23];

8. Rastreia a cobertura de código no SUT e reporta vulnerabilidades, se quaisquer violações na segurança forem detectadas;

9. Salva os casos de teste considerados interessantes para serem novas entradas. Se uma nova aresta de transição (vide seção 2.7.1) ou a contagem de incidência sobre essas for aumentada, então adiciona esse caso de testes exercitado à fila de entradas, caso contrário, vá para 6;

[22]No caso, o termo original em inglês seria "trim seeds", que consiste no processo de simplificação/redução de uma entrada específica, em vista dos prováveis caminhos de execução que essa entrada é capaz de executar.

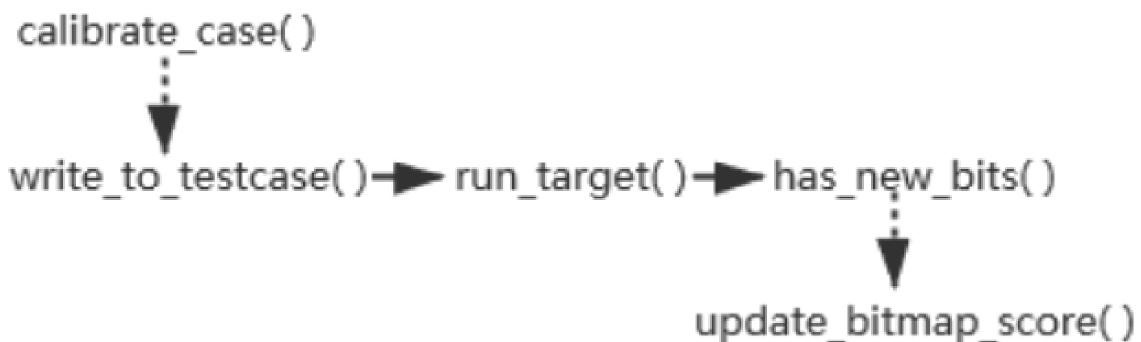
[23]Nesse caso, o autor considera o modo instrumentado do AFL (também chamado de *greybox*, em inglês), contudo, o AFL também é capaz de ser executado diretamente sobre binários

10. Faz a calibragem de novas entradas e reorganiza a fila de entradas. Se o programa encontrar uma nova entrada favorecida, define a variável *score_changed* para 1, dentro da função *update_bitmap_score*;

11. Resume a fila de entradas para entradas equivalentes, ou seja, elimina entradas que exercitam os mesmos caminhos no SUT. Esse resumo é realizado na função *cull_-queue* que, por sua vez, é apenas chamada quando *score_changed* for igual a 1, momento em que *score_changed* é então chamado para selecionar as entradas favorecidas por meio do algoritmo de seleção de entradas;

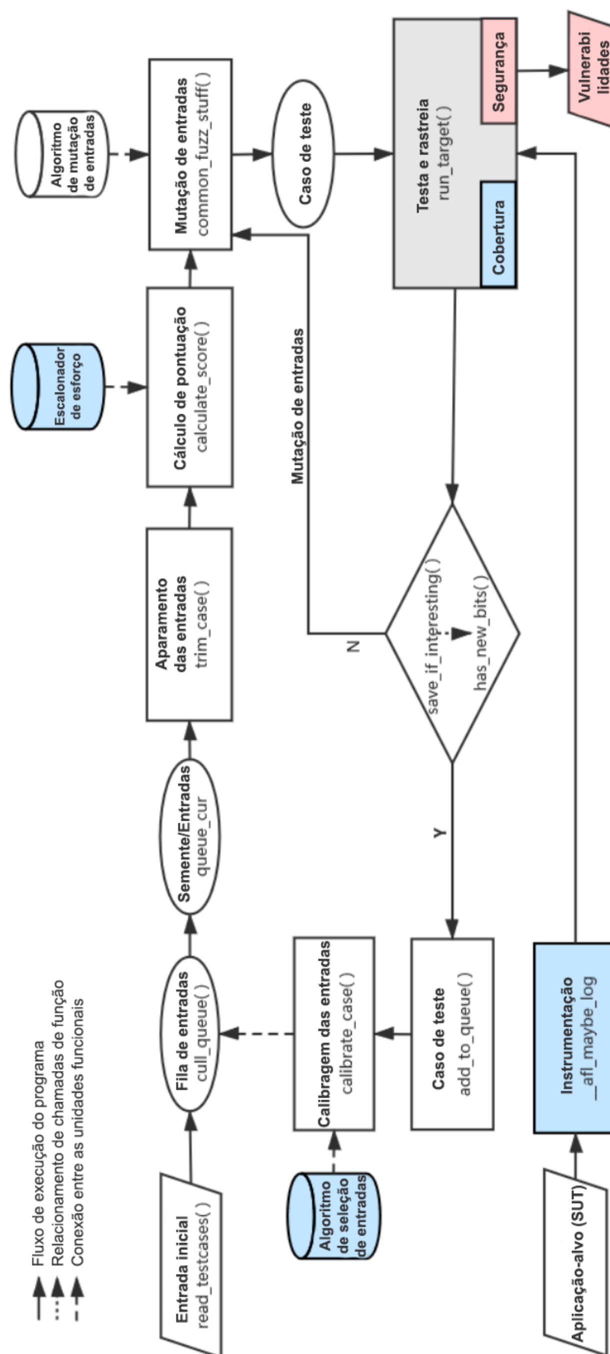
12. Se todos os casos de teste gerados pela entrada atualmente analisada, já tiverem sido consumidos pelo SUT, vai para o passo 4. O laço de repetição apenas termina quando manualmente especificado ou quando o tempo de execução (ou outra condição de término suportada) do AFL é definido.

Figura 12 – Processo de calibração de entradas do AFL.



Fonte: (PANG et al., 2021)

Figura 13 – Fluxo de trabalho do AFL.



Fonte: (PANG et al., 2021)

2.7.3 Limitações e gargalos conhecidos

Como visto anteriormente, o AFL é um *fuzzer* bem conhecido no mundo inteiro e que serviu como base para muitos outras versões aprimoradas dele próprio. Dessa forma,

suas limitações e gargalos já são bem conhecidos e têm sido abordados por diversos autores de diferentes maneiras. Segundo (XU et al., 2017), (GAN et al., 2018) e (AHMED et al., 2021), podem-se resumir os seguintes gargalos para se obter uma maior escalabilidade na versão padrão do AFL:

1. O alto *overhead* da função *fork()* (SANTOS, 2023), necessário durante o processo de instância do SUT para testagem de cada caso de teste individual. A chamada de sistema associada à função é bem conhecida pelo seu elevado custo de processamento e falta de escalabilidade, quando implementada sobre múltiplos núcleos de processamento. Nesse ponto, ao invés do tradicional mecanismo de *fork server* utilizado pelo AFL, o autor propõe a implementação complementar de um mecanismo de *snapshot* de memória, ou seja, um mecanismo que permita gravar e, quando preciso, restaurar os valores dos registradores e do *stackframe* [24], quando no momento de criação de uma nova instância do SUT para teste. Dessa forma, evita-se o *overhead* contínuo da chamada de sistema associada ao *fork()*, substituindo-a por operações menos custosas;

2. *Fuzzers* tipicamente dependem de operações de sistema de arquivos como *open/create* (para gerar o caso de teste mutado), *write* (para salvar casos de teste interessantes) e *read* (para carregar casos de teste) de pequenos arquivos em cada ciclo de *fuzzing*, especialmente quando rodando em modo paralelo. Especificamente, o processo de criar e escrever pequenos arquivos altera fortemente os metadados do sistema de arquivos (por exemplo, alocando *inode* para criação de arquivos e blocos de disco para escrita de arquivos). Esta é uma seção crítica na maioria das implementações de sistemas de arquivos e não é escalável;

3. Para *fuzzers* sincronizados executando em paralelo, analisar diretórios de casos de teste durante a fase de sincronização não é escalável pelas seguintes razões: Primeiro, o número de operações de enumeração de diretórios para descobrir novos casos de teste não sincronizados aumenta de forma não linear com mais *fuzzers*. Isso resulta em uma fase de sincronização mais longa. Por exemplo, cada *fuzzer* levará $O(f \times t)$, onde f é o número de *fuzzers* e t é o número de casos de teste em um diretório de casos de teste e; Segundo, a enumeração do diretório interfere seriamente na criação de um novo caso de teste. Isso porque uma operação de escrita

[24]Estrutura de dados que representa o quadro da pilha para uma determinada função.

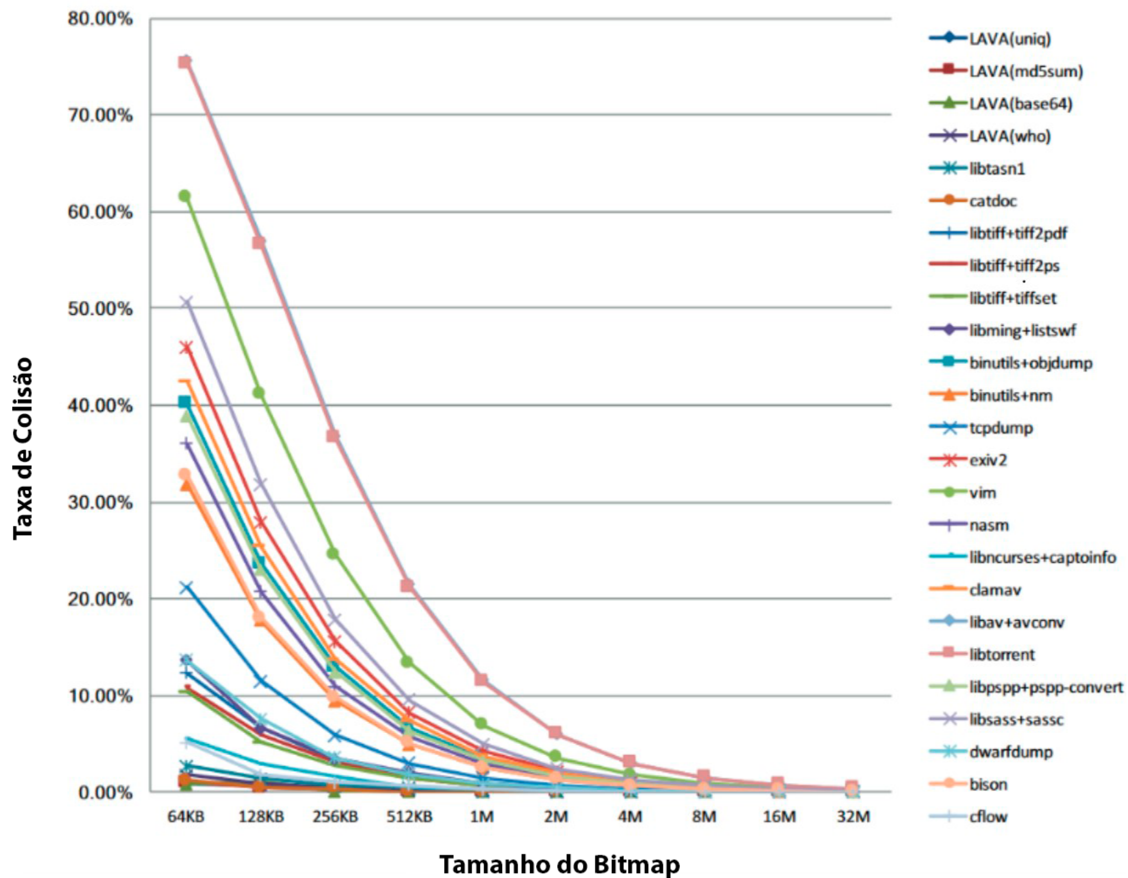
em diretório (ou seja, criar um arquivo) e uma operação de leitura (ou seja, enumerar arquivos) não podem ser realizadas simultaneamente em sistemas de arquivos típicos.

4. Um outro problema inerente e bem conhecido do AFL, é o fato que seu algoritmo de mapeamento de cobertura de código é sujeito à colisão dos *hashes* que, por sua vez, são utilizados como índices no processo de mapeamento de cobertura do SUT. Segundo o próprio autor, o tamanho escolhido para o mapa de cobertura (também chamado de *bitmap*) utilizado pelo AFL é relativamente pequeno (64KB), sendo esse tamanho escolhido de forma que tais colisões sejam esporádicas para a maioria dos alvos originalmente entendidos de serem exploráveis através do AFL, ao mesmo tempo que mantém em vista um tamanho hábil de caber por completo na maioria das caches L2 dos processadores de sua época. Nesse ponto, algumas soluções se apresentam para esse problema. Segundo (GAN et al., 2018) e (AHMED et al., 2021), a solução mais direta seria simplesmente a de se aumentar o tamanho do *bitmap* utilizado, a fim de se reduzir a probabilidade de colisões nos *hashes* utilizados no processo de mapeamento de cobertura que, dado o tempo o devido de execução para o *fuzzer*, se tornam inevitáveis.

   Contudo, embora essa solução reduza consideravelmente, ainda não elimina a probabilidade de colisão, uma vez que esta sempre será maior que zero (em decorrência inerente na forma como é construído o mapa de cobertura) e, além disso, aumentar o tamanho do *bitmap* introduz um *overhead* significativo no tempo de execução da aplicação (Figuras 14 e 15). Segundo (GAN et al., 2018), uma outra solução consiste em uma versão do AFL que implementa técnicas melhores de instrumentação do SUT, provendo uma instrumentação melhor em tempo-de-compilação (vide seção 2.7.2) e resolvendo de vez o problema de colisões. Contudo, essa solução, assim como outras que trabalham com melhores técnicas de instrumentação do SUT, dependem do código-fonte do SUT para funcionarem. Ainda, autores como (AHMED et al., 2021) exploram o uso de algoritmos de *hashing* em dois níveis, o que mitiga consideravelmente o problema de colisões, enquanto viabiliza o uso de *bitmap* maiores com baixo *overhead*. Por fim, esse problema de probabilidade de colisão de associado ao baixo tamanho padrão para o *bitmap* é, de acordo com (TEAM, 2023a), *"esse problema é subestimado na comunidade de fuzzing"*, com o autor também provendo uma solução livre-de-colisão através do uso de melhores técnicas de instrumentação, que por sua vez, também dependem do código-fonte do

SUT.

Figura 14 – Taxa de colisão de *hashes*, em função do tamanho do *bitmap*.



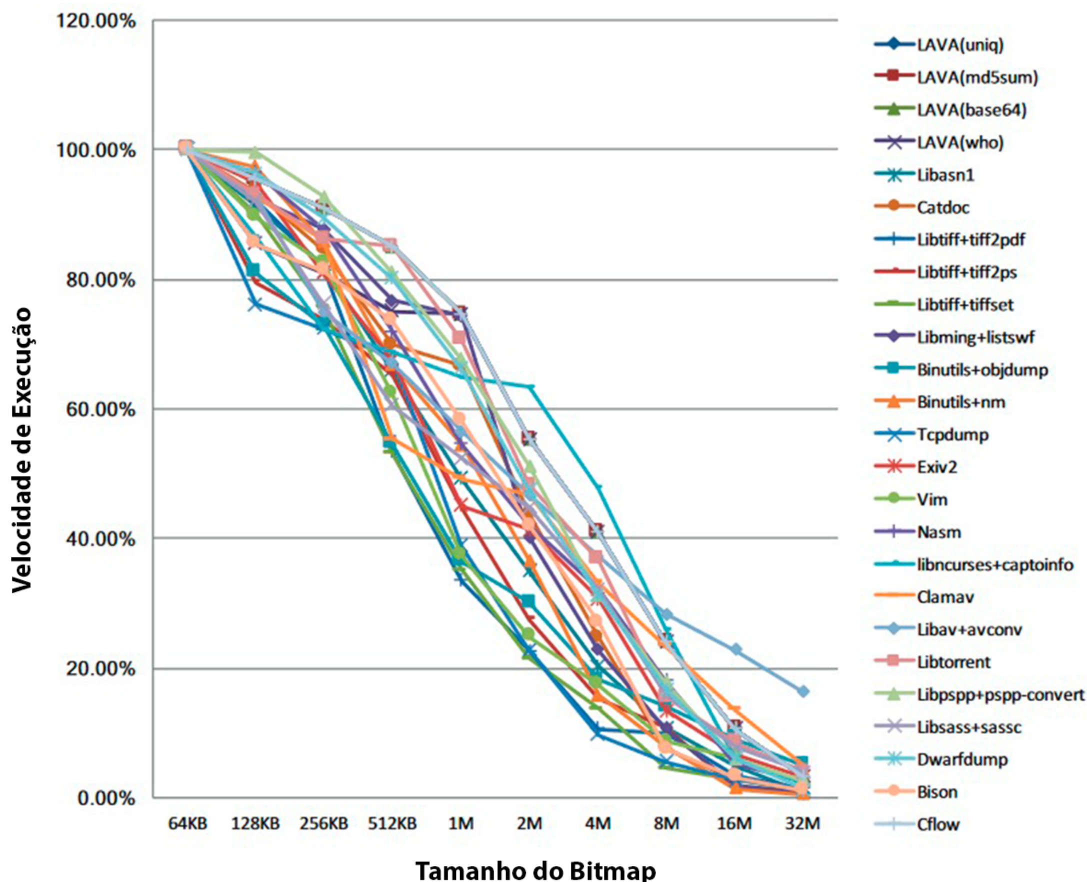
Fonte: (GAN et al., 2018)

2.8 *Benchmarking* de *fuzzers*

O *benchmarking* de *fuzzers* é um processo crítico para se medir a eficácia dessas ferramentas na detecção de falhas de software. Este procedimento permite aos pesquisadores e profissionais de segurança digital comparar diferentes *fuzzers* e suas variantes, fornecendo uma base para a escolha da ferramenta mais adequada para uma determinada aplicação ou contexto (KLEES et al., 2018). A importância do *benchmarking* de *fuzzers* é amplamente reconhecida na literatura, especialmente devido à variedade de técnicas de *fuzzing* disponíveis e à necessidade de entender seus pontos fortes e fracos (METZMAN et al., 2021a).

Várias métricas podem ser usadas para avaliar o desempenho de um *fuzzer*. Estas

Figura 15 – Velocidade de execução, em função do tamanho do *bitmap*.



Fonte: (GAN et al., 2018)

incluem, mas não estão limitadas, ao número de falhas únicas encontradas, o número de novos caminhos de código descobertos e a cobertura de código alcançada. Além disso, a eficiência de um *fuzzer* pode ser medida em termos do número de execuções do programa alvo que podem ser realizadas em um determinado período de tempo. Em algumas situações, a capacidade de encontrar falhas raras ou difíceis de detectar pode ser uma métrica importante. Essas métricas, no entanto, devem ser consideradas em conjunto para fornecer uma avaliação abrangente do desempenho do tipo de ferramenta em questão.

Ainda, segundo (METZMAN et al., 2021a), a avaliação minuciosa de ferramentas de *fuzzing* é difícil e, muitas vezes, exige tempo e recursos dos quais a maioria dos pesquisadores não dispõem. Dessa forma, surgiram ao longo do tempo algumas ferramentas com o objetivo de prover mais facilidade no processo de *benchmarking* de um *fuzzer* qualquer, tais como:

- **FuzzBench**: Plataforma de *benchmarking* para *fuzzers* em nuvem, de código aberto e gratuita para *fuzzers* de diversos tipos. É desenvolvido e mantida pela equipe

de análise de segurança do Google (METZMAN et al., 2021b). Funciona através de um modelo de "*fuzzing* como um serviço", ou seja, disponibilizando uma API para integração de *fuzzers* customizados que, após sua integração com a plataforma ser aprovada, podem ser executados para se testar o impacto de possíveis otimizações sobre cenários realistas, gerando diversos gráficos e estatísticas pertinentes, permitindo-se comparar soluções customizadas com outras existentes;

- **LAVA-M**: Ferramenta de *benchmark* desenvolvida pelo MIT Lincoln Laboratory. Segundo (DOLAN-GAVITT et al., 2016) LAVA-M (Large-Scale Automated Vulnerability Addition) funciona pela adição automatizada de vulnerabilidades em *softwares* conhecidos, permitindo dessa maneira a testagem eficiente de diferentes tipos de *fuzzers*, ainda que esse processo exija um certo grau de adaptação dos alvos pretendidos de serem explorados, nos moldes esperados pela ferramenta;
- **Fuzzgoat**: Aplicação bem-conhecida para avaliação de *fuzzers*, em especial, do AFL. Consite em um *parser* JSON[25] escrito em linguagem C, contendo aproximadamente 1300 linhas de código, deliberadamente projetada para desencadear erros de corrupção de memória, a fim de testar a eficácia de *fuzzers* como o AFL e outras ferramentas de análise de segurança (FUZZSTATI0N, 2021);

2.9 O cenário atual de otimização em *software* através do uso de IA Generativa

Com o recente avanço de tecnologias de Inteligência Artificial (IA), abriu-se um leque de possibilidades para otimização nas mais diversas áreas da ciência e da tecnologia. Mais especificamente, técnicas e modelos de aprendizado de máquina tem permitido a descoberta de novos algoritmos e otimização de código já existente para algoritmos já conhecidos há décadas, os quais, até então, já eram considerados suficientemente otimizados por cientistas e engenheiros ao longo de décadas. Segundo (DEEPMIND, 2023), nos últimos cinquenta anos, observou-se uma crescente demanda por redução energética e poder computacional que, por sua vez, tem se baseado na sua maior parte sobre otimizações a nível de *hardware* a fim de suprir a crescente demanda por poder de processamento. Contudo, à medida em que *microchips* atingem seus limites físicos, torna-se imperativo a otimização do código que executa sobre eles, a fim de tornar o processo de computação mais eficiente e sustentável. E isso torna-se especialmente importante para algoritmos

[25] JavaScript Object Notation, padrão bem-conhecido para armazenamento de informações.

que, por sua vez, são a base de códigos que são executados trilhões de vezes todos os dias.

Em um estudo recente (MANKOWITZ et al., 2023), por exemplo, pesquisadores apresentam uma IA capaz de refinar algoritmos de ordenação e torná-los mais rápidos, através da análise minuciosa em um nível onde a maioria dos seres humanos não costuma analisar com a devida propriedade: instruções em nível *assembly*, no caso, *assembly* Intel x86-64. Segundo o autor do estudo, embora a intuição humana e expertise tenham papel crucial na otimização de algoritmos, muitos desses algoritmos atingiram um grau de complexidade no qual especialistas humanos não têm sido capazes de otimizá-los ainda mais, o que, por sua vez, tem levado a um gargalo computacional crescente em todas as áreas. Valendo-se de modelos de aprendizagem de máquina, o autor conseguiu obter otimizações relevantes em nível *assembly* para algoritmos de ordenação da biblioteca de ordenação LLVM LIBC++[26]. Dessa forma, o autor ainda extrapola o potencial de aplicação da sua técnica para diversos outros algoritmos utilizados em nosso cotidiano. Ainda, em (SHYPULA et al., 2021) também verifica-se espaço de otimização, através do uso de IA, para otimização de código *assembly* Intel gerado por compiladores bem conhecidos como o GCC (GNU C Compiler), mesmo com esse último fazendo uso de *flags* de compilação mais abrangentes como *-O3*. Ainda, segundo (JONES, 2005), *"todas as aplicações são diferentes e, portanto, não existe uma configuração simples ou flags de otimização capazes de render o melhor resultado para todas."*.

Além disso, observa-se a crescente popularização de ferramentas como o ChatGPT, uma ferramenta de IA Generativa[27], desenvolvida pela OpenAI, representando atualmente um marco significativo na jornada rumo à criação de IA de conversação de alto desempenho. A ferramenta é treinada por meio de um procedimento de duas etapas, que envolve aprendizado não supervisionado seguido de um ajuste fino supervisionado. Este processo aproveita a amplitude dos dados disponíveis na internet para criar um modelo de linguagem robusto e versátil, com aplicações variadas, desde assistentes virtuais à geração de código. Trabalhos como (BISWAS, 2023), (LIU et al., 2023) e outros, demonstram que o ChatGPT tem o potencial de ser uma ferramenta valiosa para identificação de gargalos e sugestão de código otimizado, embora não tenha sido especificamente treinado para essa tarefa. Ainda, trabalhos recentes como (FU et al., 2023) demonstram que o ChatGPT já tem capacidade suficiente para *design* de *microchips*, a fim de auxiliar os projetistas da

[26]Implementação atual padrão da biblioteca padrão C++, versão 11 e superiores.

[27]Ferramenta de IA especializada em geração de conteúdo dinâmico, voltada para um tema especificado pelo usuário.

indústria de semicondutores em seus projetos. Por fim, segundo (RESEARCH, 2023), observa-se uma tendência evidente uso cada vez maior do mercado para ferramentas de IA (Figura 16), o que demonstra a relevância e, sobretudo, a confiabilidade cada vez maior de ferramentas desse tipo, em favor dos profissionais da indústria e universidades.

Dessa forma, enxerga-se nesse tipo de tecnologia a possibilidade de auxílio para o desenvolvimento da otimização proposta, uma vez que, embora uma linguagem de baixo-nível, como o *assembly* Intel *x*64 utilizado no trabalho, forneça um leque maior para otimizações do que outras linguagens de nível mais alto, também é bem conhecido o fato da dificuldade inerente de projeto, desenvolvimento e depuração nesse nível de abstração lógico (vide parágrafo anterior) e, nesse ponto, ferramentas automatizadas de análise e desenvolvimento revelam-se um recurso valioso, possibilitando o vislumbre de otimizações a nível-de-instrução *assembly*, dentro do orçamento previsto para o projeto, assim como demonstrado ao longo dessa seção.

Figura 16 – Previsão para o mercado de IA Generativa.

Fonte: (RESEARCH, 2023)

2.10 O conjunto de instruções Advanced Vector Extensions 2 (AVX2)

O conjunto de instruções Advanced Vector Extensions 2 (AVX2) [28] foi introduzido pela Intel a partir da microarquitetura Haswell em 2013, representando um avanço significativo na computação vetorizada. Ele expande as capacidades da arquitetura de conjunto de instruções *x86* ao permitir operações SIMD [29] com vetores de 256 bits, o dobro da largura dos vetores suportados pelo AVX original. Ainda, o AVX2 introduz instruções para permutação e manipulação de *bits*, além de versões vetorizadas de instruções que aumentam significativamente a eficiência em tarefas que exigem processamento paralelo, como gráficos 3D, processamento de imagem e som, codificação e decodificação de vídeo, aplicações científicas e de engenharia, entre outros.

Além disso, o AVX2 estendeu as operações SIMD para operações inteiras, o que foi uma adição significativa já que o AVX original só suportava operações de ponto flutuante. Isso se traduz em um aumento dramático na velocidade de muitas operações em aplicações que processam grandes volumes de dados, especialmente porque os dados podem ser processados em blocos muito maiores de uma vez. No entanto, é importante observar que o uso eficaz do AVX2 requer que os programas sejam cuidadosamente otimizados para tirar proveito dessas instruções. As ferramentas modernas de compilação podem realizar alguma vetorização automaticamente mas, em muitos casos, compiladores ainda deixam margem para otimização manual, conforme é demonstrado neste trabalho.

[28] Extensões Vetorizadas Avançadas, em português.
[29] Single Instruction, Multiple Data, ou Única Instrução, Múltiplos Dados, em português.

3 METODOLOGIA

Neste capítulo são detalhadas a caracterização da pesquisa e a metodologia adotada para o desenvolvimento da pesquisa proposta. Na seção 3.1 é apresentado o problema a ser explorado nesse trabalho, em vista do referencial teórico previamente apresentado; Na seção 3.2 é detalhada as premissas sobre as quais esse trabalho se desenvolveu; Na seção 3.3 é feita uma caracterização formal da metodologia de pesquisa utilizada e, por fim, na seção 3.4 são detalhadas as etapas envolvidas no procedimento metodológico adotado.

3.1 Problema de Pesquisa

Como pode-se verificar em 2.7.3, o AFL, embora já bem conhecido e estudado ao longo dos anos, ainda possui espaço de otimização e gargalos não-resolvidos em sua versão base. Mais especificamente, o problema de colisão de *hashes* que ocorre durante seu processo de mapeamento de cobertura de código é propenso a falhas, tendo sido parcialmente resolvido por diversos autores, de diferentes maneiras, ao longo do tempo. Ainda, de acordo com o observado em 2.9, ferramentas de IA abriram um novo leque de possibilidades de otimização de código gerado por compiladores como o GCC, permitindo empurrar ainda mais os limites do desempenho para algoritmos já bem estudados e otimizados manualmente ao longo das últimas décadas. Por fim, verifica-se em 2.10 que conjuntos de instruções vetorizadas, como o mais recente Intel AVX2, vêm sendo utilizados para exploração de paralelismo a nível-de-código em diversos tipos de aplicações. Dessa forma, o problema da pesquisa desse trabalho consiste em encontrar, dentre as tecnologias mais recentes estudadas, uma maneira de, senão resolver, ao menos mitigar o *overhead* em seções do AFL que, segundo a literatura vigente (vide seção 2.7.3), se revelam custosas em termos de processamento. Essas seções, bem como seu *overhead* de processamento relativo podem ser vistas no Capítulo 4.

3.2 Hipótese de Pesquisa

Esse trabalho parte do pressuposto de que é possível e viável a exploração do código *assembly* que é gerado pelo GCC, durante o processo de *build* nativo do *fuzzer* AFL. Com isso, esse trabalho busca um aumento de desempenho neste *assembly* gerado

nativamente, através do uso de técnicas de programação em nível *assembly* em vista de conjuntos de instruções modernos, que permitam explorar melhor o paralelismo a nível de instrução em uma microarquitetura específica. Dessa maneira, seria possível fornecer uma solução de otimização que exigiria pouca ou até nenhuma refatoração do código-fonte do AFL em sua versão original.

3.3 Caracterização da Pesquisa

Nesta pesquisa, baseou-se na metodologia Hipotético-Dedutiva, que segundo (PRODANOV; FREITAS, 2013), parte de uma delimitação do problema, seguido de uma precisa descrição deste, a fim de se preencher, através de hipóteses e inferências dedutivas, as lacunas que se revelem necessárias ao longo do processo. Ainda, segundo (PRODANOV; FREITAS, 2013), esta pesquisa classifica-se em relação aos diferentes aspectos:

- Em relação à abordagem, caracteriza-se como qualitativa, uma vez que trata da análise da qualidade do desempenho de seções específicas do *fuzzer* AFL, em busca de gargalos que se revelem propícios à aceleração em *software*. Ainda, também trata da análise da qualidade dos dados de desempenho, colhidos durante a análise do *fuzzer* proposto;

- Em relação à natureza, caracteriza-se como aplicada, pois como objetivo desenvolver e implementar um *assembly* otimizado para uma seção específica do AFL;

- Em relação ao objetivo, caracteriza-se como exploratória e explicativa: Exploratória, uma vez que consiste em pesquisa bibliográfica acerca da possibilidade de otimização do código gerado pelo GCC durante o processo de montagem nativo do AFL, valendo-se de paralelismo em nível-instrução, a fim de se otimizar seções críticas e bem-conhecidas do *fuzzer* estudado; Explicativa, pois também consiste em identificar a explicar os motivos pelos quais esse *assembly* otimizado é viável na prática;

- Por fim, com em relação aos procedimentos, caracteriza-se como bibliográfica e experimental: bibliográfica em razão dos levantamentos de trabalhos e estudos relacionados, a fim de se constituir a base científica necessária para realização da tarefa; experimental devido ao trabalho prático que foi executado, com objetivo de se alcançar os objetivos descritos nas seções 1.2 e 1.3.

3.4 Procedimento Metodológico

Neste trabalho, faz-se uso da metodologia ágil Scrum (S., 2023), devido à experiência do autor em projetos ágeis e ao fato de que esta metodologia em questão possibilita realizar possíveis re-definições dos requisitos da aplicação, à medida em que o projeto se desenvolve. O fluxo previsto para o projeto e suas etapas constituintes são as que seguem:

1. **Revisão da Bibliografia**:

   Esta etapa tem como objetivo o levantamento de informações técnicas que servirão como base para o desenvolvimento do referencial teórico que é utilizado neste trabalho;

2. **Delimitação do Problema**

   Devido ao amplo espectro de atuação que a tecnologia de *fuzzing* permite abordar, suas diversas implementações podem ser classificadas sob múltiplas perspectivas, assim como oferecem um amplo espectro de possíveis otimizações. Dessa forma, nessa etapa será definida a superfície de atuação da proposta, delimitando-se: O *fuzzer* a ser estudado; O corpo de testes mais adequado para validação da proposta; A seção crítica de código a ser otimizada no *fuzzer* escolhida e por último, a estratégia de otimização mais adequada para otimização da seção crítica identificada.

3. **Determinação do espaço de otimização no *fuzzer* escolhido**

   Essa etapa tem como objetivo identificar, no *fuzzer* escolhido, gargalos que possam efetivamente valer-se de paralelismo a nível-de-instrução na microarquitetura-alvo;

4. **Projeto e implementação do *assembly* otimizado na CPU**

   Nesta etapa, é projetado e implementado o *assembly* otimizado a ser executado na CPU-alvo, em vista dos possíveis gargalos identificados;

5. **Análise de Resultados**

   E, finalmente, nesta etapa é realizada a análise dos resultados obtidos na implementação do *assembly* customizado desenvolvido, a fim de se determinar se houve, de fato, o ganho de desempenho pretendido no trabalho.

3.5 Revisão da Bibliografia

Em vista da necessidade de se obter o material necessário para fornecer o embasamento científico necessário para o trabalho, realizou-se uma extensa pesquisa em repositórios acadêmicos e páginas relacionadas ao tema. Conforme mencionado na seção 2.7.3, trabalhos como o de (XU et al., 2017) demonstram que o AFL possui gargalos bem conhecidos e que vêm sendo tratados de diferentes maneiras ao longo do tempo. Ainda, como visto na seção 4, é bem-conhecido o problema de colisão de *hashes* durante o processo de mapeamento de cobertura de código do SUT e trabalhos como os de (GAN et al., 2018) e (AHMED et al., 2021), entre outros, evidenciam que a mitigação mais direta para esse problema consiste em aumentar o tamanho do seu *bitmap* utilizado no processo de cobertura de código, ao custo entretanto de um aumento considerável do *overhead* de seu tempo de execução.

Além disso, conforme descrito na seção 2.9, verifica-se a existência de espaço de otimização em compiladores com o GCC (JONES, 2005), utilizado pelo AFL em seu processo de instalação e, por último, observa-se que ferramentas de IA da atualidade têm evidenciado um espaço de otimização em diversos segmentos ((DEEPMIND, 2023), (MANKOWITZ et al., 2023) e outros). Dessa forma, estipulou-se um plano de ação que visa, sobretudo, estabelecer em que local no código do AFL é viável para a implementação de uma otimização significativa neste, além de se determinar qual é a melhor maneira de se implementar essa otimização. Na Tabela 1 pode-se encontrar um resumo dessas informações levantadas.

Informação	Fonte
Gargalos bem conhecidos do AFL e abordagens variadas para tratá-los ao longo do tempo.	(XU et al., 2017)
Problema de colisão de hashes no mapeamento de cobertura de código do SUT e estratégias para mitigação, incluindo aumento do tamanho do bitmap, resultando em maior overhead de tempo de execução.	(GAN et al., 2018), (AHMED et al., 2021)
Espaço de otimização em compiladores como o GCC, utilizado pelo AFL.	(JONES, 2005)
Otimizações emergentes em diversos segmentos impulsionadas por ferramentas de IA atuais.	(DEEPMIND, 2023), (MANKOWITZ et al., 2023)

Tabela 1 – Resumo das informações e referências utilizadas

4 DESENVOLVIMENTO DO *ASSEMBLY* OTIMIZADO

Em vista do referencial teórico levantado e tendo-se em vista o alinhamento com os objetivos apresentados, essa seção apresenta as etapas e os detalhes envolvendo o desenvolvimento do projeto. Na seção 3.5, é detalhado o processo de revisão bibliográfica que norteia esse trabalho. Na seção 4.1, é descrito o processo de delimitação do problema, ou seja, a determinação do *fuzzer* a ser trabalhado, seu corpo-de-testes mais adequado, dentre outras escolhas que se revelaram mais adequadas para a conclusão da proposta; Na seção 4.2.1, é executado o processo de execução do *profiling* do AFL, a fim de se identificar neste possíveis seções críticas que sejam mais suscetíveis à otimização e, por último, em 4.3, é detalhado o processo de projeto e desenvolvimento da versão *assembly* otimizada para *has_new_bits()*, produzida neste trabalho.

4.1 Delimitação do Problema

As subseções 4.1.1, 4.1.2 e 4.1.3, apresentadas a seguir, descrevem, respectivamente, o processo de escolha para o *fuzzer* utilizado nesse trabalho, o corpo de testes consumido por esse durante o processo de análise de gargalos e, por último, o *profiler* utilizado para análise do tempo de execução das funções envolvidas nesse processo.

4.1.1 Seleção de um *fuzzer* para análise

Esta consiste em, no primeiro lugar, na escolha de um *fuzzer* social, industrial e cientificamente relevante e que dispusesse de informações suficientes para alimentar e justificar a realização de um processo de otimização como o pretendido nesse trabalho. A partir da literatura previamente levantada, escolheu-se o famigerado *fuzzer* AFL, em função dos seguintes critérios:

- **Boa documentação e suporte à comunidade**: O AFL é um dos *fuzzers* mais conhecidos da atualidade (seção 2.7), sendo esta uma ferramenta de código-aberto bem-conhecida por profissionais e pesquisadores da área de segurança, contando com uma boa documentação e uma grande comunidade de usuários. Ademais, isso não apenas facilita a resolução de problemas, mas oferece uma ampla gama de extensões e melhorias desenvolvidas pela comunidade, que podem ajudar a adaptar o

AFL a diferentes cenários de *fuzzing*;

- **Escalabilidade**: O AFL é projetado para ser altamente eficiente em termos de recursos, o que o torna ideal para ser testado no laboratório de experimentos utilizado (vide seção 5.1), sem problemas de desempenho. Possui suporte nativo para execução em paralelo, permitindo que os usuários tirem proveito de múltiplos núcleos de CPU para aumentar a taxa de execução de testes. Dessa forma, o AFL é capaz de dividir o trabalho de *fuzzing* entre várias instâncias, cada uma explorando um subconjunto do espaço de entrada;

- **Eficiência de Detecção de *bugs***: O AFL utiliza um algoritmo de *fuzzing* evolucionário, ou seja, baseado em cobertura de código, sendo esse o tipo atualmente considerado como o mais eficaz na detecção de *bugs*. Com o auxílio de instrumentação em tempo-de-compilação, o AFL monitora quais partes do código estão sendo executadas e ajusta suas entradas de teste para explorar caminhos de código não visitados, aumentando as chances de descobrir falhas e vulnerabilidades ocultas;

- **Relevância de otimização no contexto do *fuzzing* moderno**: Uma vez que o AFL é considerado um grande avanço no contexto de *fuzzing* muito do seu fluxo de trabalho (vide seção 2.7.2) pode ser encontrado em diversas versões mais modernas desse. Nesse ponto, é natural que o leitor se questione o motivo da otimização não ser realizada sobre uma dessas versões mais modernas da ferramenta em questão. De fato, já existem diversas versões mais otimizadas do AFL em si, contudo, tais versões são otimizadas para contextos diferentes, com suas implementações possuindo diferentes variações do fluxo de execução do AFL padrão (vide seção 2.7.2), para diferentes casos de uso (vide trabalhos como (AFL++, 2023), (AHMED et al., 2021), (PANG et al., 2021), dentre outros). Dessa forma, espera-se que as otimizações realizadas na versão-base AFL possuam um potencial reaproveitamento ao longo de suas versões derivadas (vide seção 6.1).

4.1.2 Seleção de um corpo de testes

Como visto na seção 2.8, diversas métricas e ferramentas de *benchmarking* podem ser utilizadas quando na avaliação do *fuzzer* AFL. Segundo (METZMAN et al., 2021a),

a avaliação minuciosa de ferramentas de *fuzzing* é difícil e, muitas vezes, exige tempo e recursos dos quais a maioria dos pesquisadores não dispõem. Ainda, considerando-se os diversos modos de ferramenta e opções que um *fuzzer* moderno como o AFL dispõe, revelou-se desafiador a tarefa de escolher um corpo de testes que possa, sobretudo, fornecer uma visão mais geral do *overhead* das funções envolvidas em seu fluxo de trabalho mais comum (seção 2.7.2). Dessa maneira, considerando-se as opções levantadas na literatura, a escolha para essa trabalho foi o Fuzzgoat (seção 2.8), devido ao fato de que este não requer um procedimento de integração complexo como o Fuzzbench 2.8, ou um ajuste no código-fonte do SUT como o LAVA-M 2.8. Sua estrutura e funcionamento simplificado viabilizam, em tempo hábil, uma análise do mais geral do tempo de execução nas funções envolvidas no fluxo de trabalho mais comum esperado para o *fuzzer* AFL (seção 2.7.2), devido ao seu *design* especificamente projetado para testes do tipo de *fuzzer* utilizado.

4.1.3 Seleção de um *profiler* para análise

A ferramenta GProf (PROJECT, 2023a) foi utilizada com o intuito de determinar quais são os pontos de maior custo computacional (ou seja, que gera maior *overhead* de processamento) no AFL. Essa ferramenta é gratuita e de código aberto e muito conhecida por ser eficaz no monitoramento do desempenho do código. Destaca-se pela sua capacidade de auxiliar na identificação de gargalos, analisando tanto o tempo de CPU quanto as chamadas de função em programas. Como uma ferramenta de análise de desempenho, é fundamental na otimização do *software*, pois permite que os desenvolvedores identifiquem as partes do código que consomem mais recursos. Sua metodologia de análise envolve a inserção de código extra durante a compilação, que tem a função de rastrear a execução do tempo e as chamadas de função.

Além disso, o GProf emprega amostragem dinâmica do ponteiro de instrução da CPU, uma técnica que analisa o programa durante sua execução, coletando dados em intervalos regulares para fornecer um instantâneo do comportamento do programa ao longo do tempo. Isso permite que o GProf identifique os gargalos no código e forneça uma visão detalhada de como o tempo de CPU é gasto durante a execução do programa. Ainda, segundo (HPC.NRW, 2023), o GProf é ideal para localização de regiões adequadas para otimização, ajudando na mitigação do *overhead* para chamadas de funções custosas em termos de tempo de processamento. Além disso, o *overhead* induzido por sua instrumen-

tação é baixo, sendo também bem adequado para análise de aplicações C/C++ (dentre outras), assim como o AFL utilizado nesse trabalho.

4.2 *Profiling* do *fuzzer* AFL

Esta etapa consiste na análise do tempo de execução para o AFL em função do tamanho do seu *bitmap*, visto a importância crucial desse elemento no processo de funcionamento deste *fuzzer* (conforme visto na seção 2.7.2). O objetivo é determinar os trechos de código com maior *overhead* presentes na execução do AFL, que no caso, são as funções *run_target* e *has_new_bits()*(Tabela 2). Para tal, realizou-se a análise do tempo de execução três diferentes tamanhos de *bitmap*: 64KB, 128KB e 256KB. O resultado do *profiling* para essas variações de tamanho pode ser conferido na Tabela 2[30].

Tabela 2 – Dados de *profiling* do AFL para diferentes tamanhos de *bitmap*.

Função	**64KB**	**128KB**	**256KB**
run_target	44.97%	44.45%	46.40%
has_new_bits.constprop.0	39.94%	41.77%	46.51%
UR	6.51%	5.08%	1.73%
fuzz_one	3.40%	3.42%	1.73%
write_to_testcase	1.04%	1.11%	0.39%
common_fuzz_stuff	1.04%	0.92%	0.39%

Observa-se então o comportamento padrão esperado para o tempo de execução AFL, que aumenta de maneira diretamente proporcional com o aumento do tamanho do seu *bitmap*, comportamento que também pôde ser observado na Figura 14, para diferentes tipos de aplicações-alvo;

4.2.1 Análise dos gargalos encontrados

Dessa forma, após a análise de gargalos da aplicação, selecionaram-se duas rotinas que, nos cenários analisados, utilizados, apresentaram um elevado *overhead* em relação às demais. Sua listagem, descrição e análise de propensão à otimização desejada nesse trabalho, seguem abaixo:

[30]No caso, percebe-se que o nome da função *has_new_bits()* aparece sufixada com "constprop.0". Isso é o sufixo para "constant propagation"(ou "propagação de constantes", em português), uma técnica de otimização implementada nativamente pelo GCC, quando configurado em modo de otimização.

1. ***run_target***: Essa função executa o SUT em um espaço de endereçamento isolado, criado através da chamada de sistema *fork()*, com limites de memória e um tempo de execução definido. A função também configura e manipula a configuração do *timer* para acompanhar o tempo de execução do programa alvo e monitorar se o tempo limite foi atingido. Em relação ao seu elevado *overhead*, observa-se que isso decorre em função do uso da chamada de sistema *fork()* para criação dos processos do SUT que serão analisados pelo *fuzzer*. A criação de um processo filho com *fork()* é uma operação de sistema que costuma ser relativamente custosa, em decorrência do *overhead* envolvido na cópia da tabela de páginas do processo-pai para o filho. Além disso, o processo-pai também precisa monitorar pela sinalização da finalização do processo filho, mantendo e destruindo as estruturas de dados relacionadas, o que por sua vez também induz *overhead* no tempo de execução da função;

2. ***has_new_bits()***: Essa função tem como objetivo o de se verificar se a execução mais recente do SUT descobriu novos caminhos no código do corpo de testes, ou seja, se o seu mapa de cobertura de código foi atualizado. Seu algoritmo pode ser resumido da seguinte forma:

   (a) Verifica se o mapa de execução atual, representado por "*trace_bits*", atingiu novos caminhos no SUT, representado pelo mapa de virgindade "*virgin_map*";

   (b) Atualiza *virgin_map* para refletir as mudanças, caso ocorram;

   (c) Caso a única mudança em *virgin_map* seja uma contagem de ocorrências para uma tupla de *bits*, retorna 1; Retorna 2 caso novas tuplas tenham sido identificadas;

   (d) Atualiza *bitmap_changed* para refletir as mudanças, caso ocorram.

   Seu respectivo código-fonte e uma análise mais detalhada de seu funcionamento podem ser conferidos no Apêndice A.

Assim, concluí-se que, dentre as duas funções previamente analisadas, a função *has_new_bits()* é a mais viável de ser inicialmente trabalhada, por dois motivos:

- Tamanho: Enquanto *run_target* conta com 506 linhas de código, *has_new_bits()* conta com apenas 35, o que simplifica o processo de depuração e análise, em vista do tempo disponível para a tarefa;

- Sua execução é recorrente no código do AFL: Segundo a própria documentação no código do AFL original, *has_new_bits()* é chamado toda vez para cada teste de entradas realizado no SUT, sendo executada sobre estruturas de dados consideradas grandes pelo autor[31] e, dessa forma, essa função precisa ser o mais rápido possível. Pode-se corroborar essa afirmação diretamente no código-fonte da ferramenta: *"Essa função é chamada uma vez para cada chamada de exec() sobre um buffer muito grande e, dessa forma, precisa ser rápida."*[32]. Ainda, pode-se observar que, para um intervalo de tempo de 10 minutos de *profiling*, a função *has_new_bits()* é chamada mais de 400 mil vezes, como mostra a Figura 17, valor muito próximo ao total de chamadas de *run_target()*, comportamento esse que está de acordo com a documentação da ferramenta;

- Estrutura: Diferentemente de seu concorrente, *has_new_bits()* possui uma estrutura linear, sem desvios no fluxo de execução ou chamadas de sistema. Isso simplifica o projeto de um *assembly* otimizado, uma vez que um grau de acoplamento maior entre diferentes funções promove um maior dependência de seções específicas nesse *assembly* tais como a definição de *flags* específicas entre essas chamadas de função, por exemplo. Dessa forma, essa escolha também visa simplificar o processo de depuração e análise, em vista do tempo disponível para a tarefa;

Figura 17 – Análise de *profiling* realizado para o AFL, ao longo de 10 minutos, onde destaca-se o total de chamadas para *has_new_bits()* nesse período, com um valor de centenas de milhares de execuções.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 44.97      3.04     3.04   493001     0.01     0.01  run_target
 39.94      5.74     2.70   490805     0.01     0.01  has_new_bits.constprop.0
```

Fonte: Autor, 2023.

[31]Na prática, essas estruturas são os *arrays virgin_map* e *trace_bits*, vide Apêndice A.
[32]No inglês original: *"...This function is called after every exec() on a fairly large buffer, so it needs to be fast"*.

4.3 Projeto e implementação do *assembly* otimizado na CPU

Nessa etapa, considerou-se a ideia de que seria possível, com o uso de IA (vide seção 2.9), otimizar o gargalo da função *has_new_bits()* a nível *assembly*, ou seja, reescrevendo o *assembly* gerado durante a compilação do AFL original. Nessa seção são detalhados os passos realizados para realização dessa etapa.

4.3.1 Análise do *assembly* original de *has_new_bits()*

Essa etapa consiste em se determinar qual é o conjunto de instruções disponível na CPU-alvo (descrito em (INTEL, 2023)) e qual é o uso efetivo dessas instruções presentes no *assembly* original de *has_new_bits()*, gerado pelo compilador GCC no processo de compilação padrão do AFL. Primeiramente, fez-uso da ferramenta *Utilitário para identificação do processador Intel®* (INTEL, 2023a), obtendo-se a listagem do conjunto de instruções disponível para experimentos (vide Figura 15), constatando-se dessa maneira a disponibilidade (dentre outros) do conjunto de instruções vetorizadas AVX2 que, como discutido na seção 2.10, é um dos mais modernos e eficientes conjuntos de instruções SIMD atualmente disponíveis.

Após isso, realizou-se uma análise do *assembly* original de *has_new_bits()*, a fim de se determinar se algum dos conjuntos de instrução vetorizada disponíveis (listados na Figura 18) já estão sendo, de fato, utilizados no código padrão do AFL. Para realização dessa análise, utilizou-se a ferramenta *objdump* (GCC, 2023), para extração e análise do *assembly* relativo à função *has_new_bits()*, a partir do binário do AFL. Após a análise do *assembly* extraído (disponível no Apêndice A), constatou-se que, ao longo de suas 248 linhas de código *assembly*, não é feito uso de nenhuma das instruções vetorizadas mencionadas e, dessa forma, inferiu-se nesse ponto ser possível a otimização do código em questão, através do uso de instruções AVX2 (veja seção 2.10). Também, realizou-se uma análise estatística acerca da listagem de instruções, como pode ser visto na Tabela 3.

Figura 18 – Captura de tela do Utilitário para identificação do processador Intel, listando as tecnologias disponíveis na CPU utilizada, destacando a disponibilidade do conjunto de instruções AVX2.

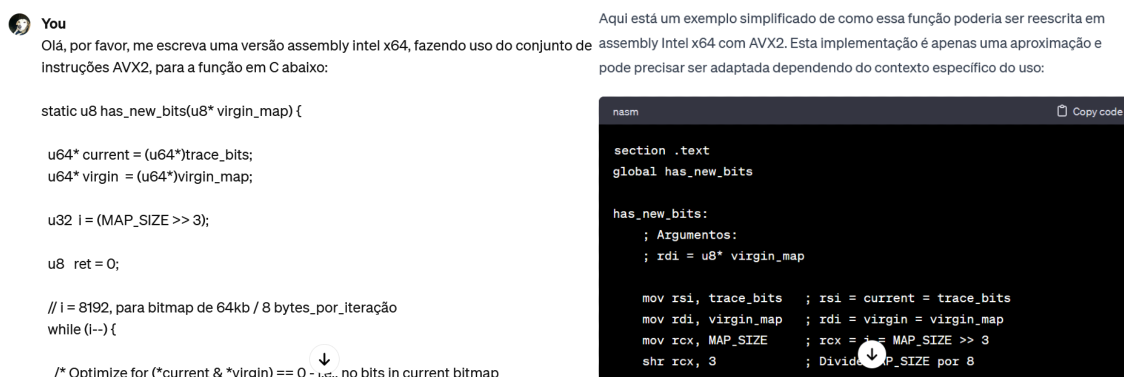
Fonte: (INTEL, 2023a)

Instrução	Categoria	Contagem Total (248)	Percentual (%)
cmp	Comparação	70	≈ 28.2
je	Salto	63	≈ 25.4
mov	Transferência de Dados	48	≈ 19.4
lea	Transferência de Dados	12	≈ 4.8
jne	Salto	12	≈ 4.8
test	Comparação	9	≈ 3.6
jmp	Salto	8	≈ 3.2
nop	(Outros / Sem Efeito)	5	≈ 2.0
sete	Configuração de Registro	4	≈ 1.6
add	Aritmética e Lógica	4	≈ 1.6
not	Aritmética e Lógica	4	≈ 1.6
and	Aritmética e Lógica	4	≈ 1.6
push	Stack	1	≈ 0.4
call	Controle	1	≈ 0.4
xor	Lógica	1	≈ 0.4
pop	Stack	1	≈ 0.4
ret	Controle	1	≈ 0.4

4.3.2 O uso da ferramenta ChatGPT no processo de desenvolvimento do código otimizado

A partir do que foi visto na seção 2.9, nessa etapa utilizou-se a ferramenta ChatGPT para auxílio na geração do código *assembly* otimizado para a função *has_new_bits()*. Na Figura 19, pode-se observar o uso da ferramenta de IA proposta para geração de uma versão inicial do *assembly* otimizado proposto, que após passar por uma série de ajustes manuais, considerou-se apto para se atingir o ganho de desempenho proposto.

Figura 19 – Captura de tela da ferramenta ChatGPT sendo utilizada para auxílio na geração da versão inicial do *assembly* otimizado proposto.



Fonte: Autor, 2023

4.3.3 Análise do *assembly* otimizado para *has_new_bits()*

Nessa etapa, considerando o que foi apresentado nas seções 2.9, 2.10 e na Tabela 3, decidiu-se que é viável de se desenvolver um *assembly* customizado para *has_new_bits()*, fazendo uso de recursos de *hardware* e técnicas de otimização que não foram identificadas no *assembly* original da função (disponível no Apêndice A), tais como:

- *Prefetching* dos *arrays* utilizados: Com o objetivo de se valer da localidade espacial do processador, é realizado o pré-carregamento de dados que são utilizados durante o fluxo de execução de *has_new_bits()*. A fim de se determinar a distância e o momento ideal para *prefetch* dos dados trabalhados, verificou-se que, segundo (BAILEY, 2023), é necessário se considerar, para cada caso, qual a necessidade da disponibilidade dos dados na cache, não trazendo esses dados para a cache muito cedo, pois isso arrisca a expulsão de dados mais necessários naquele momento e

também, nem muito tarde, deixando de valer-se da localidade temporal do processador. Ainda, para se determinar a distância de *prefetch* ideal, é necessário se considerar essa distância deve ser tal que o volume de dados trazidos para a cache seja tal, que esses dados sejam efetivamente trabalhados pelo processador durante o tempo que permanecem disponíveis na cache, evitando também que um possível excesso desse volume de dados trazidos para cache, force a expulsão de outros dados mais relevantes naquele momento. Ainda, segundo (JEFFERS; REINDERS, 2013), uma boa métrica para escola dessa distância de *prefetching* é a escolha de um fator (*n*=1,2,4,8) para o cálculo dessa distância em instruções vetorizadas (como o AVX2 utilizado no trabalho). Mais especificamente, recomenda-se o uso inicial de um fator de *n*=2 para laços de repetição que fazem uso de instruções vetorizadas. Dessa forma, determinou-se que o carregamento de 512 *bytes* para os *arrays* é um valor adequado para o perfil do algoritmo otimizado, devido a que 512 *bytes* é suficientemente à frente do fluxo de dados da função otimizada. Dessa forma, são disponibilizados na cache os dados trabalhados no laço interno de repetição (rotina *.iterate*, veja Apêndice B) *n* e da iteração *n+1*, considerando que a cada iteração do laço interno são tratados 256 *bytes*. Na Figura 20 pode-se verificar uma representação conceitual dessa estratégia utilizada;

Figura 20 – Representação conceitual da estratégia de *prefetching* utilizada, buscando disponibilizar na cache os dados dos *arrays* trabalhados, durante as iterações de *has_new_bits()*.

Fonte: Autor, 2023.

- Desenrolamento de laço em conjunto com instruções do AVX2: O novo código faz uso da técnica conhecida como desenrolamento de laço para otimização do tempo de execução do algoritmo. Para cada iteração do laço interno, são tratados 32 *bytes*, como indica as instruções *vmovdqu* que movem 32 *bytes* de dados para os registradores YMM (do conjunto de instruções AVX2) que, por sua vez, possuem uma

largura 256 *bits* (ou 32 *bytes*). O laço interno (*.iterate*), por sua vez, é repetido até que um total de 256 *bytes* sejam processados, o que acontece após 8 iterações, dessa forma: 8 iterações × 32 *bytes* por iteração = 256 *bytes* trabalhados em cada iteração do laço externo (*.loop*). Essa estratégia reduz o *overhead* associado ao controle de laço, aproveitando as operações SIMD para processamento de múltiplos dados em paralelo. Dessa maneira, vale-se efetivamente da largura de 256 *bits* dos registradores YMM providos pelo AVX2, maximizando a eficiência do processamento dos dados em blocos, cujo tamanho é perfeitamente alinhado com a largura dos registradores vetorizados AVX2 em uso. Na Figura 21, pode-se observar uma representação em alto-nível de quais trechos de *has_new_bits()* original farão uso das otimizações propostas: No trecho marcado em azul, observa-se a rotina equivalente no código *assembly* otimizado (rotina *.loop*) que se beneficiará da otimização através do desenrolamento de laço, reduzindo o total de iterações necessárias para percorrer os *arrays* relacionados; No trecho marcado em verde, observa-se a rotina equivalente no código *assembly* otimizado (rotina *.iterate*) que se beneficiará do uso de instruções vetorizadas AVX2, com o objetivo de tratar as operações sobre os *arrays* em blocos de 32 *bytes* ao invés de apenas 8 *bytes* na versão original do algoritmo.

Também, da mesma forma que fora feito anteriormente para a versão original de *has_new_bits()* (Tabela 3), realizou-se uma análise estatística para as instruções constituintes dessa nova versão (disponível no apêndice B), que pode ser conferida na Tabela 4. Ao se comparar as Tabelas 3 e 4, percebe-se claramente uma redução no número de instruções necessárias para implementação do mesmo algoritmo (uma redução total de mais de 1000%), na versão otimizada. Ainda, pode-se perceber um número muito menor (de 63 na versão original para apenas 1 na versão otimizada) de instruções de salto, que são bem conhecidas pelo seu alto custo de tempo de processamento associadas. Por fim, percebe-se na versão otimizada o uso mais eficiente de instruções de acesso à memória, fazendo uso de instruções SIMD para um tratamento em blocos dos *arrays* da função, reduzindo dessa maneira o tempo necessário de processamento.

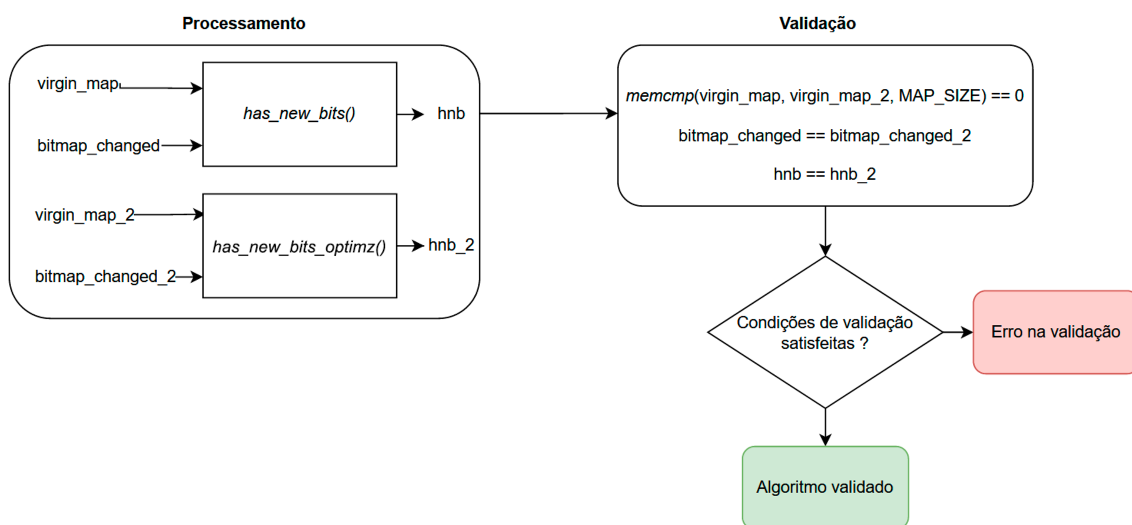
Por último, para validação das saídas desse *assembly* otimizado desenvolvido, utilizou-se o algoritmo de validação que pode ser visto na Figura 22. No caso, é realizado, para cada variável de entrada de *has_new_bits()* uma cópia destas e então, essas cópias são alimentadas para a versão customizada do algoritmo. Ao final do processamento, caso o novo algoritmo tenha um processamento fiel à sua versão original, é de esperar que suas

Instrução	Categoria	Contagem Total (29)	Percentual (%)
mov	Transferência de Dados	2	6.9
xor	Aritmética e Lógica	4	13.8
prefetchnta	Otimização de Cache	2	6.9
add	Aritmética e Lógica	3	10.3
cmp	Comparação	2	6.9
jnz	Salto	2	6.9
sub	Aritmética e Lógica	1	3.4
jz	Salto	1	3.4
vmovdqu	Transferência de Dados SIMD	4	13.8
vpand	Aritmética e Lógica SIMD	2	6.9
vptest	Aritmética e Lógica SIMD	1	3.4
vpandn	Aritmética e Lógica SIMD	1	3.4
vzeroupper	Otimização SIMD	2	6.9
ret	Controle	2	6.9

Tabela 4 – Análise das instruções do código *assembly* otimizado para a função *has_new_bits()*.

saídas produzidas sejam totalmente equivalentes (para os mesmos valores de entradas), validando-se dessa forma a proposta do algoritmo desenvolvido.

Figura 22 – Fluxograma para validação das saídas esperadas para a função *has_new_bits()* original e sua versão *assembly* otimizada.



Fonte: Autor, 2023.

Figura 21 – Algoritmo de *has_new_bits()* original, com destaque para as seções nas quais a versão otimizada faz uso das estratégias de otimização de laço (em azul), em conjunto com as instuções AVX2 (em verde).

```
1   static inline u8 has_new_bits(u8* virgin_map) {
2     u64* current = (u64*)trace_bits;
3     u64* virgin  = (u64*)virgin_map;
4     u32  i       = (MAP_SIZE >> 3);
5     u8   ret     = 0;
6
7     while (i--) {
8
9       if (unlikely(*current) && unlikely(*current & *virgin)) {
10
11        if (likely(ret < 2)) {
12
13          u8* cur = (u8*)current;
14          u8* vir = (u8*)virgin;
15
16          if ((cur[0] && vir[0] == 0xff) || (cur[1] && vir[1] == 0xff) ||
17              (cur[2] && vir[2] == 0xff) || (cur[3] && vir[3] == 0xff) ||
18              (cur[4] && vir[4] == 0xff) || (cur[5] && vir[5] == 0xff) ||
19              (cur[6] && vir[6] == 0xff) || (cur[7] && vir[7] == 0xff))
20            ret = 2;
21          else
22            ret = 1;
23        }
24
25        *virgin &= ~*current;
26      }
27
28      current++; virgin++;
29    }
30
31    if (ret && virgin_map == virgin_bits) bitmap_changed = 1;
32
33    return ret;
34  }
```

Fonte: Autor, 2023.

5 ANÁLISE DOS RESULTADOS

Nessa seção é feita a análise do *assembly* otimizado para *has_new_bits()* (desenvolvido na seção 4.3), a fim de comparar seu desempenho em vista do *assembly* padrão gerado pelo GCC. Em um primeiro momento, é realizada uma análise comparativa do tempo de execução para a função *has_new_bits()* original, isoladamente, em contaste com sua nova versão usando o *assembly* customizado desenvolvido. Após isso, é realizada uma análise de *profiling* para antes e depois da integração com a versão customizada desenvolvida nesse trabalho. O detalhamento da execução dessas etapas segue abaixo:

5.1 Laboratorio de Experimentos

Para execução da análise, foi necessário estabelecer um ambiente de operações adequado às necessidades de projeto, ou seja, um laboratório de experimentação que disponha da instalação e configuração devida das aplicações e recursos que serão utilizados nesse trabalho. A listagem desses elementos, assim como o seu respectivo papel no projeto, segue abaixo:

- CPU I7-10750H: Segundo (INTEL, 2023), o Intel Core i7-10750H é um processador de laptop de alto desempenho da 10ª geração da Intel, lançado em abril de 2020, sendo voltado para jogos e aplicativos de edição de vídeo e fotografia. É baseado na microarquitetura *Comet Lake* e possui 6 núcleos e 12 *threads*, com uma frequência base de 2.6 GHz e uma frequência turbo máxima de 5.0 GHz, além de possuir 12MB de cache L3 e suporte à tecnologia *Hyper-Threading*. Além disso, dispõe (dentre outros) do conjunto de instruções vetorizadas AVX2, (INTEL, 2023b), utilizado nesse trabalho para implementação do *assembly* otimizado em CPU proposto;

- Ubuntu Linux 22.04, 64 *bits* executando sobre *indows Subsystem for Linux* (WSL2): Sistema operacional responsável por fornecer os recursos necessários para execução do *fuzzer* AFL, tais como os compiladores GCC, Nasm e as demais ferramentas necessárias para sua análise e implementação do *assembly* otimizado proposto;

- Visual Studio Code: Segundo (CORP., 2023), o VS Code é uma IDE (*Integrated Development Environment*[33]) gratuita e de código aberto desenvolvida pela Microsoft. Oferece suporte a uma variedade de linguagens de programação, possui um

[33] Ambiente de Desenvolvimento Integrado, em Português.

rico ecossistema de extensões e é amplamente reconhecido por sua interface intuitiva, destacamento de sintaxe, terminal integrado e recursos poderosos de depuração. A flexibilidade e personalização do VSCode o tornaram uma escolha popular entre os desenvolvedores de software modernos;

- GNU *Compiler Collection*: Segundo (TEAM, 2023b), o GCC, ou GNU *Compiler Collection*, é uma coleção de compiladores de código aberto desenvolvida pelo projeto GNU. Originalmente criado para o C, o GCC agora suporta uma variedade de linguagens de programação. É amplamente reconhecido por seu desempenho, portabilidade e rica otimização de código, tornando-se uma escolha padrão em muitos sistemas operacionais;

- NASM (*Netwide Assembler*): Segundo (PROJECT, 2023b), o NASM, ou Netwide Assembler, é um *assembler* gratuito e de código aberto que transforma código *assembly* em diversos formatos binários, inclusive o formato de *assembly* Intel *x*86-64 utilizado no trabalho. Devido à sua compatibilidade com diversos conjuntos de instrução e comprovada eficiência, o NASM é uma ferramenta essencial para muitos desenvolvedores de software de baixo nível.

5.2 Análise de desempenho para as funções *has_new_bits()* original e otimizada, de maneira isolada

Para análise de desempenho dessa nova versão customizada de *has_new_bits()*, realizou-se a análise do tempo de execução para as duas versões. Essa análise inicial deu-se através de repetidas amostras de execução de ambas as funções (original e customizada), ao longo de milhares de execuções consecutivas, em diferentes configurações para o total de iterações, sobre diferentes tamanhos de *bitmap* (64KB, 128KB, 256KB), tendo em vista os seguintes fatores para a decisão desses parâmetros da análise:

- Segunda a própria documentação do AFL, a função *has_new_bits()* é chamada repetidamente ao longo da execução do programa, com seu total de execuções atingindo centenas de milhares de execução, para um intervalo curto de tempo, fato comprovado durante as experiências realizadas no trabalho (vide seção 4.2.1);

- Segunda a própria documentação do AFL, o maior tamanho viável para o *bitmap* (representado pela constante MAP_SIZE) é de 256KB, uma vez que, segundo o

autor, caso o SUT exija um *bitmap* maior do que isso, é preferível que o usuário da ferramenta ajuste os níveis de instrumentação diretamente no SUT, a fim de evitar o *overhead* de se trabalhar com um *bitmap* muito grande[34] [35].

Dessa maneira, buscou-se verificar se a relação de ganho de desempenho das funções analisadas se mantém estável ao longo de um determinado intervalo tempo (proporcional ao total de iterações). Ao se analisar os resultados encontrados, verificou-se um ganho de desempenho que é consistente através de centenas de milhares de chamadas, para os diferentes tamanhos de *bitmap* testados. Verificando-se os tempos de execução para a versão normal e otimizada da função desenvolvida (Tabelas 6, 7 e 8), calculou-se o ganho percentual médio para todas as iterações, ao longo dos diferentes tamanhos de *bitmap* testados. Dessa maneira, obtiveram-se os ganhos que podem ser conferidos na Tabela 5, para os diferentes tamanhos de *bitmap* testados (cálculos disponíveis no Apêndice C).

MAP_SIZE	**Ganho (%)**
64KB	60.92%
128KB	58.50%
256KB	54.92%

Tabela 5 – Ganho médio percentual para diferentes tamanhos de *bitmap*.

Ainda, ao analisar as Figuras 23, 24 e 25, percebe-se uma diminuição esperada para o tempo de execução das funções testadas, logo nas primeiras iterações, para os três tamanhos de *bitmap* analisados, devido à disponibilização desses dados na cache do processador na medida em que estes são requisitados no laço de repetição do teste, passando dessa maneira a valer-se da localidade espacial do processador. Por fim, nas Tabelas 6, 7 e 8 pode-se observar uma grande oscilação nos valores para a variância, em torno das 1000 iterações. Nesse ponto, encontram-se duas explicações possíveis para esse fenômeno:

- Para amostras menores, os *outliers*[36] podem ter um impacto maior na variância,

[34] No código-fonte do AFL, pode-se encontrar o comentário original em inglês: *"...you probably want to keep it under 18 or so for performance reasons (adjusting AFL_INST_RATIO when compiling is probably a better way to solve problems with complex programs)."*, onde o autor refere-se ao tamanho máximo recomendado para o *bitmap*, para o caso de programas complexos, como sendo igual a $2^{18} = 256KB$.

[35] É de se observar também que, essa alternativa de aumentar o grau de instrumentação do SUT, em detrimento do aumento do tamanho do *bitmap*, só é possível para o caso em que se tem disponível o código-fonte do SUT para instrumentação.

[36] No contexto de análise estatística, *outliers* são dados que se diferenciam drasticamente de todos os outros. Em outras palavras, um *outlier* é um valor que foge da normalidade e que pode (e provavelmente irá) causar anomalias nos resultados obtidos em cálculos de variância e desvio-padrão.

enquanto em amostras maiores, o impacto de um único *outlier* é diluído. No entanto, se os *outliers* se tornarem consistentes ou houver um aumento na frequência de *outliers* em torno de 1000 iterações, isso poderia explicar o aumento da variância nesse ponto;

- Algoritmos adaptativos ou *branch prediction*: Os processadores modernos e os sistemas operacionais podem ajustar seu comportamento com base no padrão de uso. Eles podem ter otimizações que se ajustam ao longo do tempo, e essas otimizações podem mudar a eficiência do código em execução, influenciando a variância.

Sendo assim, partiu-se para a implementação da função otimizada, já devidamente integrada no fluxo de execução do AFL, a fim de se verificar se há ganhos em uma análise de desempenho em um cenário de integração da função otimizada à aplicação completa, o que revelará qual é o ganho de desempenho da função otimizada, de fato, quando executada sobre um cenário de execução completa do AFL sobre o corpo de testes escolhido anteriormente (veja seção 4.1.2).

MAP_SIZE	**Método**	**Nº Iterações**	**Tempo (µs)**	**Variância (µs)**
64KB	ORIG	1	5.200	0.000
	OTIM		2.100	0.000
	ORIG	10	3.790	0.000
	OTIM		1.470	0.000
	ORIG	100	3.735	0.002
	OTIM		1.410	0.002
	ORIG	1000	3.777	0.046
	OTIM		1.405	0.042
	ORIG	10000	3.801	0.021
	OTIM		1.496	0.021
	ORIG	100000	3.916	0.001
	OTIM		1.604	0.001

Tabela 6 – Tempos médio de execução e variância para *has_new_bits* original e sua versão otimizada, em microssegundos, para um *bitmap* de 64KB.

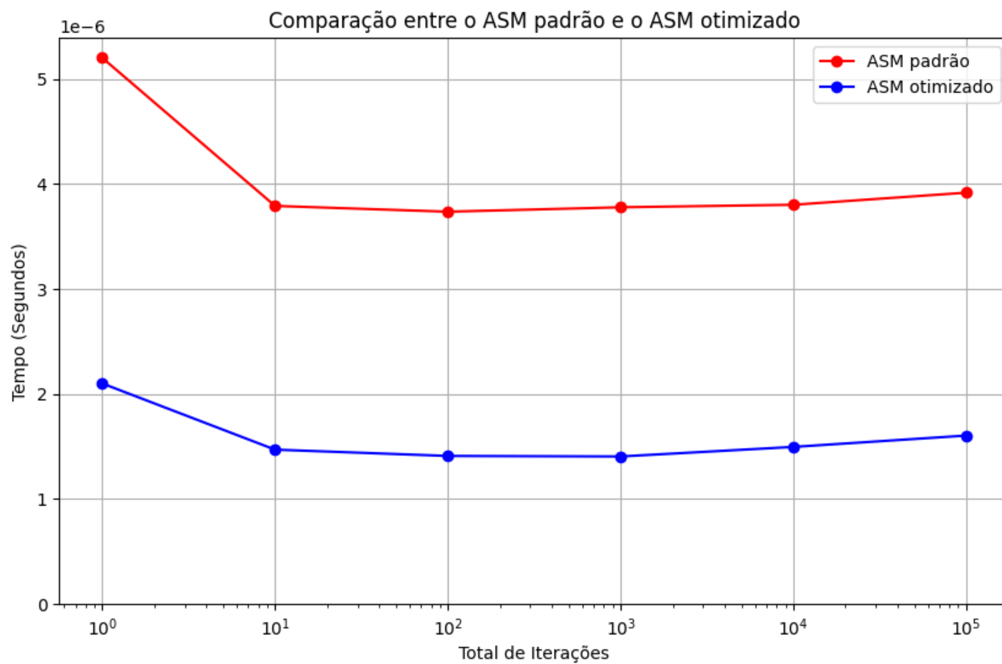
MAP_SIZE	Método	Nº Iterações	Tempo (µs)	Variância (µs)
128KB	ORIG	1	25.800	0.000
	OTIM		12.900	0.000
	ORIG	10	21.420	0.000
	OTIM		9.380	0.000
	ORIG	100	22.368	0.007
	OTIM		8.903	0.008
	ORIG	1000	12.117	0.082
	OTIM		3.812	0.092
	ORIG	10000	7.685	0.029
	OTIM		3.249	0.030
	ORIG	100000	7.807	0.002
	OTIM		3.253	0.002

Tabela 7 – Tempos médio de execução e variância para *has_new_bits* original e sua versão otimizada, em microssegundos, para um *bitmap* de 128KB.

MAP_SIZE	Método	Nº Iterações	Tempo (µs)	Variância (µs)
256KB	ORIG	1	18.000	0.000
	OTIM		8.400	0.000
	ORIG	10	16.070	0.000
	OTIM		7.170	0.000
	ORIG	100	16.122	0.003
	OTIM		6.945	0.004
	ORIG	1000	16.059	0.054
	OTIM		7.787	0.063
	ORIG	10000	15.945	0.023
	OTIM		7.010	0.025
	ORIG	100000	16.238	0.002
	OTIM		7.103	0.002

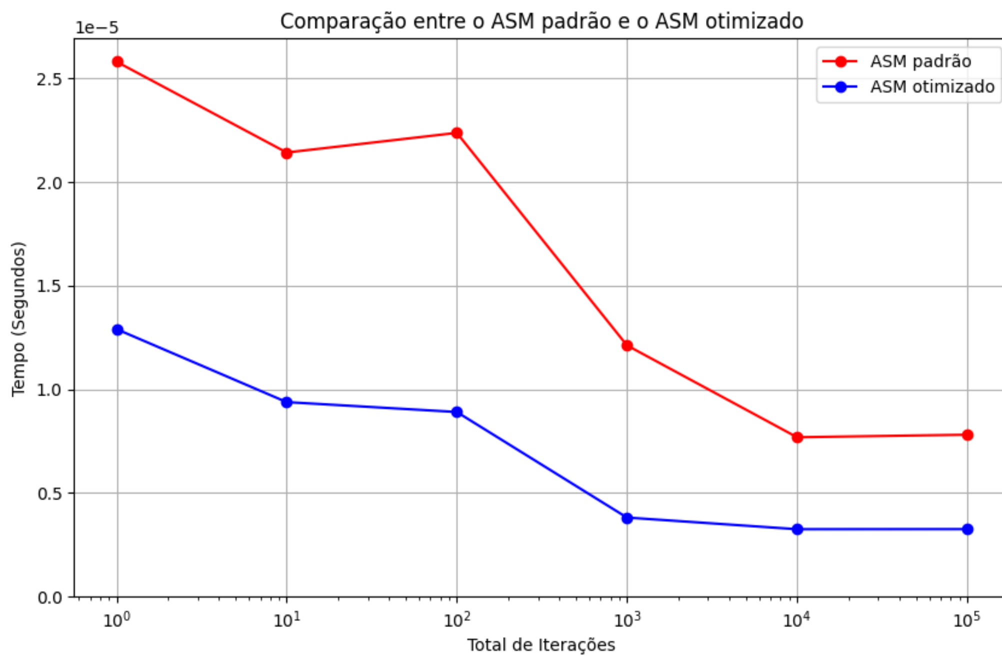
Tabela 8 – Tempos médio de execução e variância para *has_new_bits* original e sua versão otimizada, em microssegundos, para um *bitmap* de 256KB.

Figura 23 – Análise do desempenho para a função *has_new_bits()* otimizada com relação a sua versão original, para um *bitmap* de 64KB.



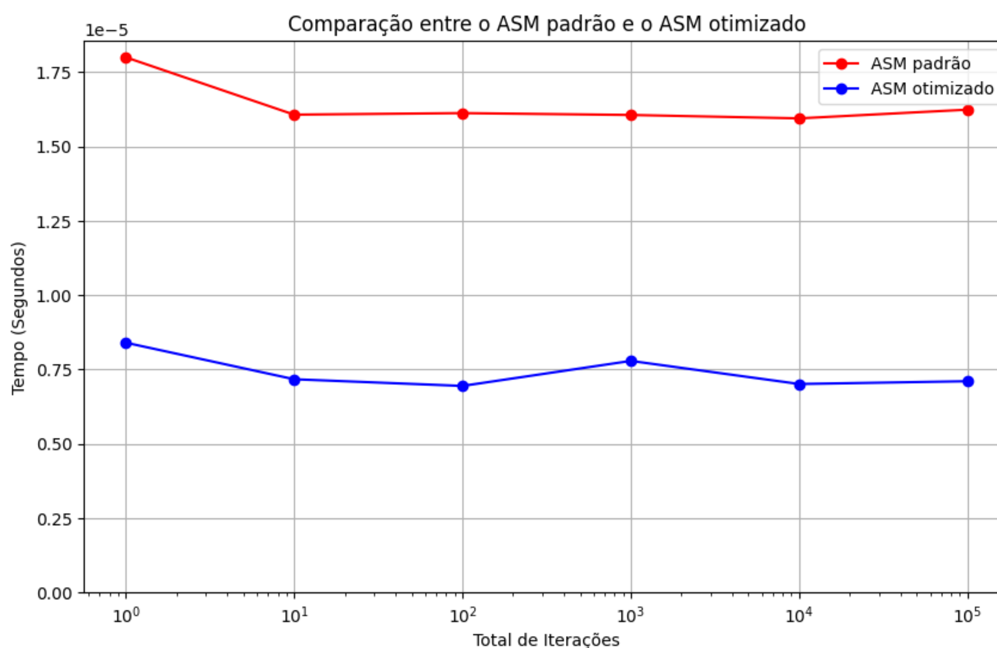
Fonte: Autor, 2023.

Figura 24 – Análise do desempenho para a função *has_new_bits()* otimizada com relação a sua versão original, para um *bitmap* de 128KB.



Fonte: Autor, 2023.

Figura 25 – Análise do desempenho para a função *has_new_bits()* otimizada com relação a sua versão original, para um *bitmap* de 256KB.



Fonte: Autor, 2023.

5.3 Análise de *profiling* do AFL após a integração com a versão otimizada de *has_new_bits()* desenvolvida

Após a análise preliminar da função otimizada de forma isolada na aplicação-alvo, integrou-se a versão otimizada de *has_new_bits()* ao fluxo de trabalho do AFL, realizando-se uma nova análise, na qual verifica-se, de fato, uma redução de *overhead* considerável da sua versão otimizada, para todos os tamanhos de *bitmap* testados, em um cenário realista de execução do AFL sobre o corpo de testes escolhido (veja 4.1.2). Na Tabela 9, observam-se os resultados comparativos do *profiling* realizado após a integração da versão otimizada, onde constata-se uma redução considerável do *overhead* do tempo de execução proporcional[37] para a função *has_new_bits()* que, em sua versão otimizada, apresenta um ganho de até 37.37% para o melhor caso (com 256KB de tamanho do *bitmap*) e de até 16.14% para o pior caso (com 64KB de tamanho do *bitmap*), com relação a sua versão original (Tabela 2).

Ainda, na Figura 26, pode-se observar a variação do tempo de execução para *has_-*

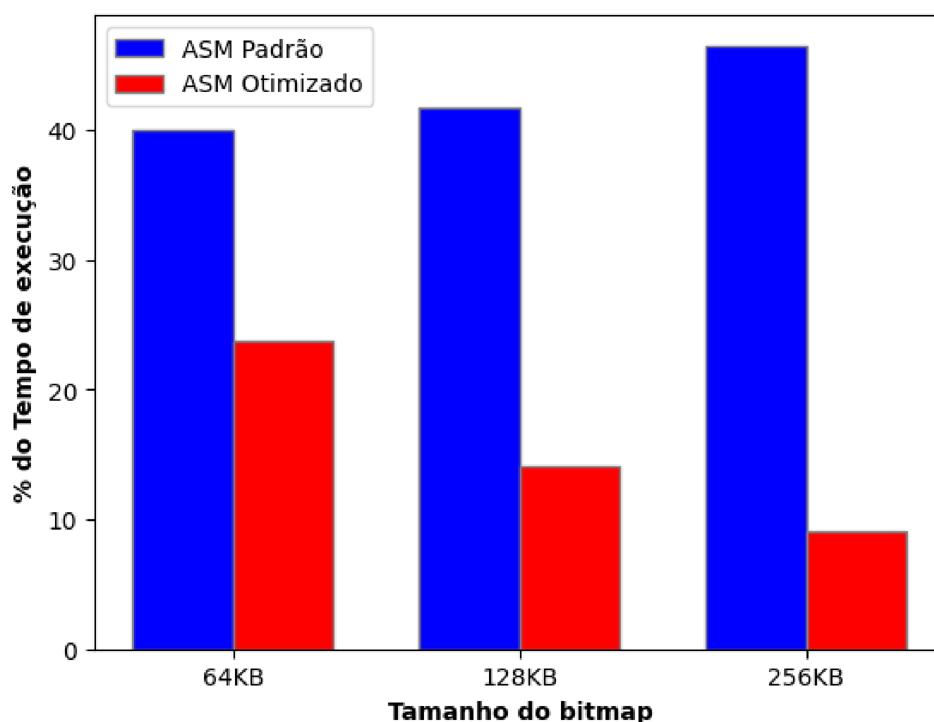
[37] No caso, proporcional ao tempo de execução de todas as outras funções analisadas durante o *profiling* da aplicação.

new_bits() em sua versão original (barra azul), em contraste com sua versão otimizada (barra vermelha). Nesse ponto, é importante observar que, ao se variar o tamanho do *bitmap*, o tempo de execução para todos trechos de código envolvendo o *bitmap* em todas as outras funções, também aumentará de acordo e, diferentemente da versão otimizada de *has_new_bits()*, esses trechos não foram igualmente otimizados, provocando uma diferença cada vez maior nos tempos de execução de todas as funções que lidam com o *bitmap*, com relação a *has_new_bits()*.

Tabela 9 – Dados comparativos de *profiling* do AFL para diferentes tamanhos de *bitmap*, já integrado com a versão otimizada de *has_new_bits()* desenvolvida.

Função	**64KB**	**128KB**	**256KB**
has_new_bits.constprop.0	39.94%	41.77%	46.51%
has_new_bits_asm_optimz	23.80%	14.10%	9.14%

Figura 26 – Tempos de execução percentuais sobre o tempo de execução total da aplicação, para as duas versões de *has_new_bits()*, normal e otimizada, para os diferentes tamanhos de *bitmap* testados.



Fonte: Autor, 2023.

Por fim, realizou-se uma análise do ganho proporcionado pelo *assembly* otimizado desenvolvido, em termos de velocidade de execução quando integrado de fato ao AFL. Nas Figuras 27 e 28, pode-se verificar, respectivamente, o total de execuções alcançadas

em um mesmo intervalo de tempo, antes e depois da integração com a versão *assembly* otimizada de *has_new_bits()* desenvolvida.

Figura 27 – Total de execuções do SUT alcançadas para um determinado intervalo de tempo, antes da integração com a versão *assembly* otimizada de *has_new_bits()*.

```
                  american fuzzy lop 2.57b (fuzzgoat)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 30 sec        │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec         │  total paths : 185   │
│ last uniq crash : 0 days, 0 hrs, 0 min, 17 sec        │ uniq crashes : 3     │
│  last uniq hang : none seen yet                       │   uniq hangs : 0     │
├─ cycle progress ────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 4 (2.16%)     │    map density : 0.11% / 0.00%         │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.00 bits/tuple       │
├─ stage progress ────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc             │ favored paths : 12 (6.49%)             │
│ stage execs : 12.0k/24.6k (48.78%) │  new edges on : 185 (100.00%)       │
│ total execs : 80.2k             │ total crashes : 17 (3 unique)          │
│  exec speed : 2721/sec          │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────┤
│   bit flips : 3/80, 0/77, 0/71                  │    levels : 3          │
│  byte flips : 0/10, 0/7, 0/2                    │   pending : 183        │
│ arithmetics : 0/556, 0/0, 0/0                   │  pend fav : 12         │
│  known ints : 0/64, 0/194, 0/88                 │ own finds : 184        │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a        │
│       havoc : 160/65.5k, 0/0                    │ stability : 100.00%    │
│        trim : 99.99%/31, 0.00%                  ├────────────────────────┘
^C────────────────────────────────────────────────┘          [cpu000: 27%]
```

Fonte: Autor, 2023.

Figura 28 – Total de execuções do SUT alcançadas para um determinado intervalo de tempo, após a integração com a versão *assembly* otimizada de *has_new_bits()*.

```
                  american fuzzy lop 2.57b (fuzzgoat)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 30 sec        │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec         │  total paths : 179   │
│ last uniq crash : 0 days, 0 hrs, 0 min, 13 sec        │ uniq crashes : 3     │
│  last uniq hang : none seen yet                       │   uniq hangs : 0     │
├─ cycle progress ────────────────┬─ map coverage ─┴──────────────────────┤
│  now processing : 16 (8.94%)    │    map density : 0.06% / 0.00%         │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.50 bits/tuple       │
├─ stage progress ────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc             │ favored paths : 10 (5.59%)             │
│ stage execs : 13.8k/32.8k (42.05%) │  new edges on : 179 (100.00%)       │
│ total execs : 81.9k             │ total crashes : 9 (3 unique)           │
│  exec speed : 2684/sec          │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────┤
│   bit flips : 4/72, 0/69, 0/63                  │    levels : 3          │
│  byte flips : 0/9, 0/6, 0/2                     │   pending : 177        │
│ arithmetics : 0/503, 0/140, 0/0                 │  pend fav : 10         │
│  known ints : 0/47, 1/160, 0/88                 │ own finds : 178        │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a        │
│       havoc : 145/65.5k, 0/0                    │ stability : 100.00%    │
│        trim : 99.99%/34, 0.00%                  ├────────────────────────┘
^C────────────────────────────────────────────────┘          [cpu000: 29%]
```

Fonte: Autor, 2023.

Dessa forma, concluem-se os seguintes pontos, em vista dos resultados alcançados:

- Conclui-se que o uso das estratégias de otimização propostas (veja seção 4.3) se revelam bem-sucedidas, com seu ganho de desempenho fornecido sendo observado em todos os tamanhos de *bitmap* utilizados nos testes;
- Embora o tempo de execução de cada função do AFL não seja constante ao longo de múltiplas análises, o ganho de desempenho proporcionado pela versão otimizada de *has_new_bits()* se revela estável ao longo de múltiplas execuções, como mostrado nas Figuras 23, 24 e 25;
- A escolha do conjunto de instruções AVX2 revelou-se, de fato, um excelente recurso de *hardware* para aceleração de aplicações que, como o AFL, fazem extenso uso de operações vetorizáveis através de instruções SIMD. Ainda, observa-se que uma das versões otimizadas mais famosas, o AFL++ (AFL++, 2023), também tem potencial de valer-se da mesma estratégia de otimização utilizada nesse trabalho, uma vez que este último também faz uso de uma versão de *has_new_bits()* que, por sua vez, possui uma assinatura muito semelhante à versão do seu antecessor AFL[38];
- O compilador GNU GCC, embora já bem estabelecido na comunidade de aplicações C/C++, ainda deixa bastante espaço para otimização do seu código *assembly* gerado, o que corrobora, empiricamente, o embasamento da literatura vigente (vide seção 2.9);
- Ferramentas de IA generativas como o ChatGPT já se revelam capazes, de fato, em auxiliar na produção de código *assembly* Intel *x*64 moderno, tendo em vista conjuntos de instrução da atualidade como o AVX2;
- Embora a solução proposta não resolva definitivamente o problema de colisão do AFL (veja seção 4), essa oferece uma mitigação no custo do *overhead* para um *bitmap* maior. De qualquer forma, a mesma estratégia de otimização que fora aplicada em *has_new_bits()* pode, a princípio, ser aplicada ao longo de diversas outras funções do AFL, bem como diversas outras aplicações que dispõem de conjuntos de instrução vetorizadas como o AVX2.

[38]Informação disponível no código-fonte do AFL++ em (CONTRIBUTORS, 2021).

6 CONSIDERAÇÕES FINAIS

Em vista dos resultados apresentados nas seções 5.2 e 5.3, concluí-se que o uso das estratégias de otimização propostas (veja seção 4.3 revelou-se, de fato, bem sucedido na otimização da principal função analisada neste trabalho. Além disso, evidencia-se neste trabalho a viabilidade de uso de instruções vetorizadas SIMD, para aceleração de trechos de código paralelizáveis, tal como a função *has_new_bits()*.

Por fim, demonstra-se o potencial de ferramentas de IA, como o ChatGPT (veja seção 4.3.2), para auxílio na geração de código *assembly* Intel otimizado, explorando dessa maneira o comprovado espaço de otimização (veja seção 2.9) que ainda é deixado por compiladores como o GCC, mesmo quando em modo de otimização máxima.

6.1 Trabalhos Futuros

Em vista de fato de que o AFL é o código-base para muitos outros *fuzzers* que o sucederam, é de esperar que a estratégia de otimização apresentada se apresente válida até para versões mais novas deste *fuzzer* tal como o AFL++ que, por sua vez, também padece do mesmo problema nativo de colisão de *hashes* durante o processamento do seu mapa de cobertura (TEAM, 2023a). Dessa forma, ainda que o AFL++ disponha de soluções de instrumentação que contornam esse problema, essas soluções ainda exigem acesso ao código-fonte do SUT, além de que uma instrumentação de menor granularidade também implica no *overhead* decorrente dessa instrumentação. Ainda, em vista das similaridades entre o AFL padrão e o AFL++, acredita-se que esse último seja igualmente passível de se beneficiar das abordagens de otimização descritas nesse trabalho.

Além disso, enfatiza-se o fato de que, embora o corpo de testes utilizado, no caso, o utilitário Fuzzgoat (seção 2.8) tenha se revelado útil para validar, em tempo hábil, o fluxo de execução padrão do AFL (seção 2.7.2), fornecendo uma visão geral dos tempos de execução das funções envolvidas, é de se esperar que o uso de uma ferramenta de *benchmarking* mais completa como o Google Fuzzbench (seção 2.8), forneça uma visão muito mais completa da abrangência da otimização alcançada nesse trabalho, a despeito das suas dificuldades de configuração e curva de aprendizado inerentes. Por fim, assume-se que a estratégia de otimização a nível-de-instrução com o uso do AVX2, não está restrita à função *has_new_bits()* analisada, podendo ser replicada a diversas outras funções do AFL, inclusive, a função que revelou o maior *overhead* durante o *profiling*

realizado (vide Tabela 2), no caso, a função *run_target()* e outras.

6.2 Contribuições desse trabalho

Nesse trabalho é explorado um mecanismo de mitigação para o *overhead* induzido quando no aumento do tamanho do seu mapa de cobertura, com o objetivo de se reduzir a ocorrência de possíveis colisões de *hashes*, no processo de cálculo do mapa de cobertura do SUT pelo *fuzzer* AFL. Mecanismos de mitigação e até de resolução completa desse problema de colisão já são abordados por diferentes autores ((GAN et al., 2018), (AHMED et al., 2021) e outros), embora nenhum autor de fato parece ter resolvido esse problema em termos do funcionamento AFL em si, sendo esse problema ainda presente até mesmo em versões mais recentes da ferramenta (TEAM, 2023a). Além disso, esse trabalho demonstra empiricamente o espaço de otimização que é deixado por compiladores como o GCC, que por sua vez, é bem-conhecido (JONES, 2005) por não ser tão inteligente ao gerar código otimizado para contextos muito específicos, tal como a micro-arquitetura utilizada nos experimentos (vide seção 5.1). Adicionalmente, esse trabalho corrobora com trabalhos recentes (vide seção 2.9) que evidenciam a capacidade de ferramentas de IA generativas em auxiliar na produção de código otimizado a nível *assembly*. Dessa forma, esse trabalho abre caminho para uma exploração mais profunda de exploração de paralelismo a nível de instrução, utilizando-se a família do conjunto de instruções AVX que, por sua vez, revelam-se ideais para aceleração de diversos tipos de aplicações, inclusive, *fuzzers* de aplicação como o AFL (SIMON; VERMA, 2020).

Por último, evidencia-se neste trabalho a importância de práticas de codificação segura, a abrangência de ferramentas de análise estática e dinâmica, o fluxo do processo de descobrimento de *bugs* em aplicações comerciais e qual o cenário da cybersegurança no contexto de técnicas de *fuzzing* da atualidade. Espera-se que, com isso, futuros estudiosos do tema possam ter um ponto do partida seguro, quando estes se debruçarem sobre as infinitas possibilidades de otimização de código *assembly* Intel *x*64 para aplicações da atualidade.

REFERÊNCIAS

ADOBE. **Adobe collaborates with Trend Micro and the Microsoft Active Protections Program**. 2021. Disponível em: https://blog.adobe.com/en/publish/2021/07/27/adobe-collaborates-trend-micro-microsoft-active-protections-program.html. Acesso em: 31 jul. 2021.

AFL++. 2023. Disponível em: https://github.com/AFLplusplus/AFLplusplus. Acesso em: 18 mar. 2023.

AHMED, A. et al. Bigmap: Future-proofing fuzzers with efficient large maps. In: IEEE. **2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. [S.l.], 2021. p. 531–542.

APPLE. **Another one bites the apple!** 2019. Disponível em: https://engineering.linecorp.com/en/blog/another-one-bites-the-apple/. Acesso em: 31 jul. 2021.

BAILEY, M. J. **Data Prefetch**. 2023. <https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/prefetch.2pp.pdf>. Acessado em 18 mar. 2021. Acesso em: 20 nov. 2023.

BBVA, O. **When Computer Bugs Were Actual Insects**. 2021. Disponível em: https://www.bbvaopenmind.com/en/technology/innovation/when-computer-bugs-where-real-insects/. Acesso em: 06 aug. 2021.

BISWAS, S. Role of chatgpt in computer programming.: Chatgpt in computer programming. **Mesopotamian Journal of Computer Science**, v. 2023, p. 8–16, 2023.

BOEHME, M.; CADAR, C.; ROYCHOUDHURY, A. Fuzzing: Challenges and reflections. **IEEE Softw.**, v. 38, n. 3, p. 79–86, 2021.

BRADEN, R. **RFC1122:Requirements for Internet hosts-communication layers**. [S.l.]: RFC Editor, 1989.

CHEN, C. et al. A systematic review of fuzzing techniques. **Computers & Security**, v. 75, p. 118–137, 2018. ISSN 0167-4048. Disponível em: https://www.sciencedirect.com/science/article/pii/S0167404818300658.

CHEN, P.; CHEN, H. Angora: Efficient fuzzing by principled search. In: IEEE. **2018 IEEE Symposium on Security and Privacy (SP)**. [S.l.], 2018. p. 711–725.

CHEN, T. M.; ABU-NIMEH, S. Lessons from stuxnet. **Computer**, v. 44, n. 4, p. 91–93, 2011.

CHEN, Y. et al. Taming compiler fuzzers. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 48, n. 6, p. 197–208, jun. 2013. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/2499370.2462173.

CISCO. **Threat Research - The Mutiny Fuzzing Framework and Decept Proxy**. 2017. Disponível em: https://blogs.cisco.com/security/talos/mutiny-decept. Acesso em: 20 jul. 2021.

COMMITTEE, I. S. et al. Ieee standard glossary of software engineering terminology. **IEEE Std**, v. 610, p. 12, 1990.

COMPANY bugsnag A. S. B. **The Worst Computer Bugs in History: Race conditions in Therac-25**. 2021. Disponível em: https://www.bugsnag.com/blog/bug-day-race-condition-therac-25. Acesso em: 17 jul. 2021.

COMPANY bugsnag A. S. B. **The Worst Computer Bugs in History: The Ariane 5 Disaster**. 2021. Disponível em: https://www.bugsnag.com/blog/bug-day-ariane-5-disaster. Acesso em: 17 jul. 2021.

CONTRIBUTORS, A. **afl-fuzz-bitmap.c in AFLplusplus**. 2021. <https://github.com/AFLplusplus/AFLplusplus/blob/stable/src/afl-fuzz-bitmap.c>. Acessado em 18 nov. 2023.

CORP., M. **Visual Studio Code - Code Editing. Redefined**. 2023. <https://code.visualstudio.com/>. Acesso em: 13 out. 2023.

Data Privacy Manager. **Your Simple Guide to Common Vulnerabilities and Exposures**. 2023. <https://dataprivacymanager.net/your-simple-guide-to-common-vulnerabilities-and-exposures/>. Acesso em: 13 out. 2023.

DEEPMIND. **AlphaDev Discovers Faster Sorting Algorithms**. 2023. <https://www.deepmind.com/blog/alphadev-discovers-faster-sorting-algorithms>. Acesso em: 10 out. 2023.

DING, R. et al. Hardware support to improve fuzzing performance and precision. In: **Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.: s.n.], 2021. p. 2214–2228.

DOLAN-GAVITT, B. et al. Lava: Large-scale automated vulnerability addition. In: IEEE. **2016 IEEE symposium on security and privacy (SP)**. [S.l.], 2016. p. 110–121.

ERNST, M. D. **Static and dynamic analysis: Synergy and duality**. [S.l.]: WODA, 2003.

FORBES.COM. **Cyber Crimes Costs Projected To Reach $2 Trillion by 2019**. 2016. Disponível em: http://www.forbes.com/sites/stevemorgan/2016/01/17/cyber-crime-costs-projected-to-reach-2-trillion-by-2019. Acesso em: 20 mar. 2021.

FU, Y. et al. **GPT4AIGChip: Towards Next-Generation AI Accelerator Design Automation via Large Language Models**. 2023.

FUZZSTATI0N. **Fuzzgoat**. 2021. Disponível em: https://github.com/fuzzstati0n/fuzzgoat. Acesso em: 20 mar. 2021.

GAN, S. et al. Collafl: Path sensitive fuzzing. In: IEEE. **2018 IEEE Symposium on Security and Privacy (SP)**. [S.l.], 2018. p. 679–696.

GCC, D. do G. **objdump(1) — Linux manual page**. 2023. <https://man7.org/linux/man-pages/man1/objdump.1.html>. Acesso em: 18 mai. 2023.

GLOBO, G. **Risco de vazamentos de dados faz aumentar procura por empresas de cibersegurança**. 2021. Disponível em: https://g1.globo.com/jornal-nacional/noticia/2021/02/15/risco-de-vazamentos-de-dados-faz-aumentar-procura-por-empresas-de-ciberseguranca.ghtml. Acesso em: 28 jul. 2021.

GODEFROID, P.; LEVIN, M. Y.; MOLNAR, D. Automated whitebox fuzz testing. In: . [s.n.], 2008. Disponível em: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

GODEFROID, P.; LEVIN, M. Y.; MOLNAR, D. Sage: whitebox fuzzing for security testing. **Communications of the ACM**, ACM New York, NY, USA, v. 55, n. 3, p. 40–44, 2012.

HASTINGS, R. Joyce. purify: Fast detection of memory leaks and access errors. In: **Proc. of the Winter USENIX Conference**. [S.l.: s.n.], 1991.

HPC.NRW. **HPC.NRW gprof Tutorial**. 2023. <https://hpc-wiki.info/mediawiki/hpc_images/8/83/HPC.NRW_gprof_Tutorial.pdf>. Acesso em: [data de acesso].

IBM. **Test Case Generation by Grammar-based Fuzzing for Model-driven Engineering**. 2021. Disponível em: https://www.research.ibm.com/haifa/conferences/hvc2012/papers/paper45.pdf. Acesso em: 31 jul. 2021.

INTEL. **Dynamic Analysis vs. Static Analysis**. 2021. Disponível em: https://software.intel.com/content/www/us/en/develop/documentation/inspector-user-guide-linux/top/getting-started/dynamic-analysis-vs-static-analysis.html. Acesso em: 22 aug. 2021.

INTEL. **Intel Processor Identification Utility - Windows Version**. 2023. <https://www.intel.com.br/content/www/br/pt/download/12136/intel-processor-identification-utility-windows-version.html>. Acesso em: 13 out. 2023.

INTEL. **Intel® Advanced Vector Extensions 2 (Intel® AVX2)**. 2023. Disponível em: https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/009/intel-advanced-vector-extensions-2-intel-avx2/. Acesso em: 11 mai. 2023.

INTEL. **Processador Intel® Core™ i7-10750H**. 2023. Disponível em: https://www.intel.com.br/content/www/br/pt/products/sku/201837/intel-core-i710750h-processor-12m-cache-up-to-5-00-ghz/specifications.html. Acesso em: 13 jan. 2023.

JANG, D. et al. Fuzzing@ home: Distributed fuzzing on untrusted heterogeneous clients. In: **Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses**. [S.l.: s.n.], 2022. p. 1–16.

JEFFERS, J.; REINDERS, J. **Intel Xeon Phi Coprocessor High Performance Programming**. Morgan Kaufmann, 2013. ISBN 9780124104143. Disponível em: https://www.sciencedirect.com/book/9780124104143/intel-xeon-phi-coprocessor-high-performance-programming.

JONES, M. T. Optimization in gcc. **Linux journal**, v. 2005, n. 131, p. 11, 2005.

JUUSO, A.-M.; TAKANEN. Building secure software using fuzzing and static code analysis. In: . [S.l.: s.n.], 2011.

KLEES, G. et al. Evaluating fuzz testing. In: **Proceedings of the 2018 ACM SIGSAC conference on computer and communications security**. [S.l.: s.n.], 2018. p. 2123–2138.

LAKSHMINARAYAN, S. B. **Fuzzing: A Comparison of Fuzzing Tools**. Dissertação (Mestrado) — University of Twente, 2023.

LCAMTUF.COREDUMP.CX. **American Fuzzy Lop**. 2021. Disponível em: https://afl-1.readthedocs.io. Acesso em: 18 mar. 2021.

LCAMTUF.COREDUMP.CX. **American Fuzzy Lop - Technical Details**. 2023. Disponível em: https://lcamtuf.coredump.cx/afl/technical_details.txt. Acesso em: 13 mai. 2023.

LI, J.; ZHAO, B.; ZHANG, C. Fuzzing: a survey. **Cybersecurity**, SpringerOpen, v. 1, n. 1, p. 1–13, 2018.

LI, Y.; FENG, C.; TANG, C. A large-scale parallel fuzzing system. In: **Proceedings of the 2nd International Conference on Advances in Image Processing**. [S.l.: s.n.], 2018. p. 194–197.

LIU, C. et al. Improving chatgpt prompt for code generation. **arXiv preprint arXiv:2305.08360**, 2023.

MANKOWITZ, D. J. et al. Faster sorting algorithms discovered using deep reinforcement learning. **Nature**, Nature Publishing Group UK London, v. 618, n. 7964, p. 257–263, 2023.

MCNALLY, R. et al. **Fuzzing: the state of the art**. [S.l.], 2012.

METZMAN, J. et al. Fuzzbench: an open fuzzer benchmarking platform and service. In: **Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering**. [S.l.: s.n.], 2021. p. 1393–1403.

METZMAN, J. et al. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In: **Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. (ESEC/FSE 2021), p. 1393–1403. ISBN 9781450385626. Disponível em: https://doi.org/10.1145/3468264.3473932.

MICROSOFT. **A brief introduction to fuzzing and why it's an important tool for developers**. 2020. Disponível em: https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers. Acesso em: 20 mar. 2021.

MICROSOFT. **Project OneFuzz**. 2020. Disponível em: https://www.microsoft.com/en-us/research/project/project-onefuzz. Acesso em: 20 jul. 2021.

MICROSOFT. **The Golden Hour remake - Defining metrics for a successful security operations**. 2023. Disponível em: https://techcommunity.microsoft.com/t5/microsoft-defender-for-endpoint/the-golden-hour-remake-defining-metrics-for-a-successful/ba-p/782014. Acesso em: 06 set. 2023.

MICROSOFT. **Relatório de andamento do SDL**. 2023. Disponível em: https://download.microsoft.com/download/E/3/5/E35F01FC-B38B-47FD-8598-70D53990B0CB/SDLProgressReportPortuguese.pdf. Acesso em: 14 mai. 2023.

MITCHELL, J. **Automated Tools for System and Application Security**. 2023. <https://crypto.stanford.edu/cs155old/cs155-spring18/lectures/16-tools-static-analysis-fuzzing.pdf>. Acesso em: 03 jun. 2023.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011.

NIST. **National Vulnerability Database**. 2021. Disponível em: http://nvd.nist.gov/. Acesso em: 24 aug. 2021.

PANG, H. et al. Spotfuzz: Fuzzing based on program hot-spots. **Electronics**, MDPI, v. 10, n. 24, p. 3142, 2021.

PRODANOV, C. C.; FREITAS, E. C. de. **Metodologia do trabalho científico: métodos e técnicas da pesquisa e do trabalho acadêmico-2ª Edição**. [S.l.]: Editora Feevale, 2013.

PROJECT, G. **GNU gprof - The GNU Profiler**. 2023. <https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_9.html>. Acesso em: [data de acesso].

PROJECT, T. N. A. **NASM Documentation**. 2023. <https://www.nasm.us/doc/>. Acesso em: 13 out. 2023.

RESEARCH, P. **Generative AI Market**. 2023. <https://www.precedenceresearch.com/generative-ai-market>. Acesso em: 19 mai. 2023.

S., B. **O que é metodologia Scrum? Gerenciamento de Projetos Scrum**. 2023. <https://www.nimblework.com/pt-br/agile/metodologia-scrum/>. Acessado em 10 dez. 2023. Acesso em: 10 dez. 2023.

SANTOS, C. A. S. **A primitiva fork()**. 2023. <https://www.dca.ufrn.br/~adelardo/cursos/DCA409/node34.html>. Acesso em: 17 mai. 2023.

SEAL, S. M. **Optimizing web application fuzzing with genetic algorithms and language Theory**. [S.l.]: Wake Forest University, 2016.

SHYPULA, A. et al. Learning to superoptimize real-world programs. **arXiv preprint arXiv:2109.13498**, 2021.

SIMON, L.; VERMA, A. Improving fuzzing through controlled compilation. In: IEEE. **2020 IEEE European Symposium on Security and Privacy (EuroS&P)**. [S.l.], 2020. p. 34–52.

SPARKS, S. et al. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In: IEEE. **Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)**. [S.l.], 2007. p. 477–486.

TAVARES, P. **Fuzzing introduction: Definition, types and tools for cybersecurity pros**. 2020. Disponível em: https://resources.infosecinstitute.com/topic/fuzzing-introduction-definition-types-and-tools-for-cybersecurity-pros/. Acesso em: 20 nov. 2022.

TEAM, A. D. **afl-clang-lto - collision free instrumentation at link time**. 2023. <https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md>. Acesso em: 15 out. 2023.

TEAM, G. D. **GNU Compiler Collection (GCC) Internals**. 2023. <https://gcc.gnu.org/onlinedocs/gcc.pdf>. Acesso em: 13 out. 2023.

WEBER. Vulnerabilidades e exploits: técnicas, detecção e prevenção. **FURG**, v. 1, p. 100, 06 2010.

WU, M. et al. One fuzzing strategy to rule them all. In: **Proceedings of the International Conference on Software Engineering**. [S.l.: s.n.], 2022.

XU, W. et al. Designing new operating primitives to improve fuzzing performance. In: **Proceedings of the 2017 ACM SIGSAC conference on computer and communications security**. [S.l.: s.n.], 2017. p. 2313–2328.

ZERO, G. P. **Project Zer0**. 2014. Disponível em: https://github.com/googleprojectzero. Acesso em: 24 jul. 2021.

ZHOU, X. et al. Ultrafuzz: Towards resource-saving in distributed fuzzing. **arXiv preprint arXiv:2009.06124**, 2020.

APÊNDICE A – CÓDIGO *ASSEMBLY* ORIGINAL PARA *HAS_NEW_BITS()*, SEM OTIMIZAÇÃO

O código-fonte para a função *has_new_bits()* original pode ser vista na Figura 29 e uma análise linha-a-linha de seu funcionamento, bem como a análise de complexidade da função seguem abaixo:

Figura 29 – Código da função *has_new_bits()*.

```
1   static inline u8 has_new_bits(u8* virgin_map) {
2     u64* current = (u64*)trace_bits;
3     u64* virgin  = (u64*)virgin_map;
4     u32  i       = (MAP_SIZE >> 3);
5     u8   ret     = 0;
6
7     while (i--) {
8
9       if (unlikely(*current) && unlikely(*current & *virgin)) {
10
11        if (likely(ret < 2)) {
12
13          u8* cur = (u8*)current;
14          u8* vir = (u8*)virgin;
15
16          if ((cur[0] && vir[0] == 0xff) || (cur[1] && vir[1] == 0xff) ||
17              (cur[2] && vir[2] == 0xff) || (cur[3] && vir[3] == 0xff) ||
18              (cur[4] && vir[4] == 0xff) || (cur[5] && vir[5] == 0xff) ||
19              (cur[6] && vir[6] == 0xff) || (cur[7] && vir[7] == 0xff))
20            ret = 2;
21          else
22            ret = 1;
23        }
24
25        *virgin &= ~*current;
26      }
27
28      current++; virgin++;
29    }
30
31    if (ret && virgin_map == virgin_bits) bitmap_changed = 1;
32
33    return ret;
34  }
```

Fonte: Autor, 2023.

- [Linha 1] Declaração da função. Observa-se o uso dos atributos *static* e *inline*,

com o objetivo de, respectivamente: Prover encapsulamento da função, ou seja, sugerir ao compilador que ela só poderá ser chamada de dentro da função onde é declarada, o que visa evitar conflitos de nomes de funções em outros arquivos; Sugere ao compilador que o código da função seja inserido diretamente no local onde a função seria chamada, evitando-se dessa forma o *overhead* de uma chamada de função convencional. Essa última estratégia pode vir a proporcionar ganhos de desempenho, em detrimento de um aumento no tamanho do binário final gerado;

Seu valor de retorno é do tipo inteiro não-sinalizado de 8 *bits* (u8) e seu único parâmetro, consiste em um ponteiro para *virgin_map* que é um mapa que o AFL usa para acompanhar quais ramos do código foram executados pelo menos uma vez;

- [Linhas 2-5] Declaração de variáveis. Nas linhas 2 e 3 observa-se que, embora as estruturas *virgin_map* e *trace_bits* sejam vetores em que cada elemento possui 8 *bits* de largura, os ponteiros **current* e **virgin* que operam sobre essas estruturas, trabalham em *offsets* [39] de 64 *bits*, por uma questão de eficiência. Quando o *laço* opera sobre as estruturas *trace_bits* e *virgin_map*, ao usar um ponteiro de 64 *bits*, pode-se ler 8 *bytes* (64 *bits*) de uma vez. Isso permite que seja feita a verificação e operação relacionados em blocos de 8 *bytes* em vez de apenas um *byte* por vez, melhorando assim a eficiência. Essa técnica, no entanto, pressupõe que os dados são alinhados de acordo com suas larguras naturais (ou seja, dados de 64 *bits* são alinhados em endereços que são múltiplos de 8). Isso é normalmente verdadeiro para a maioria dos sistemas e compiladores modernos, mas pode não ser o caso em alguns cenários;

  Na linha 4 observa-se a declaração da variável *i*, responsável pelo controle do *laço* e cujo valor inicial é igual a 8192; Por fim, na linha 5 inicializa-se a variável *ret*, responsável por armazenar o valor de retorno da função;

- [Linha 7] Início do *laço*;

- [Linha 9] Nessa linha estão sendo realizadas duas verificações usando a operação lógica AND (&&). A função *unlikely()* é uma macro que dá uma dica ao compilador

[39]Em ciência da computação, um *offset* dentro de uma matriz ou outra estrutura de dados é um inteiro indicando a distância entre o começo do objeto e um dado elemento ou ponto, presumivelmente dentro do mesmo objeto. O conceito de uma distância é válido apenas se todos os elementos do objeto forem do mesmo tamanho.

sobre a probabilidade de uma determinada condição ser verdadeira, a fim de otimizar o preditor de desvio da CPU. Neste caso, a macro está sugerindo ao compilador que a condição dentro dos parênteses é provavelmente falsa. Na primeira verificação, verifica-se se o valor apontado pelo ponteiro **current* é diferente de zero. Ou seja, se houve alguma execução na transição de ramo correspondente; Na segunda verificação, é realizada uma operação de AND *bit* a *bit* entre os valores apontados pelos ponteiros **current* e **virgin*. Essa operação retorna um valor diferente de zero (verdadeiro) somente se houverem *bits* correspondentes que são ambos iguais 1, no endereço apontado em **current* e **virgin*. Isso, por sua vez, significa que o ramo de código correspondente foi executado (porque há um bit correspondente setado em **current*), além de indicar esse ramo ainda não tinha sido atingido em nenhuma das execuções anteriores (pois o *bit* correspondente ainda está setado em **virgin*);

- [Linha 11] Nessa linha é verificado se já foram detectadas mudanças no mapa no mapa de cobertura do SUT (representado por *virgin_map*) A função *unlikely()* é uma macro que dá uma dica ao compilador sobre a probabilidade de uma determinada condição ser verdadeira, a fim de otimizar o preditor de desvio da CPU;
- [Linhas 13-14] Nessas linhas são declarados dois novos ponteiros de 1 *byte*, **vir* e **cur*, que apontam para o mesmo lugar de **virgin* e **current* respectivamente. A razão pela qual isso está sendo feito aqui é para se permitir a manipulação de **virgin* e **current* (64 *bits* cada) a um nível de granularidade menor;
- [Linhas 16-22] Nessas linhas são feitas comparações entre os valores apontados por **cur* e **vir*, 8 *bytes* por vez, a fim de se determinar se há algum *byte* em **cur* que é diferente de zero (ou seja, um ramo foi atingido) e o *byte* correspondente em **vir* é 0xff (ou seja, este é o primeiro hit para esse ramo). Se tal *byte* existir, *ret* é definido como 2, indicando que novos ramos foram atingidos. Caso contrário, *ret* é definido como 1, indicando que não foram encontrados novos ramos, mas sinaliza um aumento do número de vezes em que certos ramos já conhecidos foram atingidos;
- [Linha 25] Essa linha de código está limpando (setando para zero) todos os *bits* em **virgin* que correspondem a *bits* setados (iguais a 1) em **current*. Esta operação é realziada para se atualizar o *virgin_map* depois que uma nova execução de ramo é descoberta. Limpar o *bit* correspondente em **virgin* indica que a transição de ramo correspondente foi atingida pelo menos uma vez;

- [Linha 28] Nessa linha são incrementadas as posições apontadas por **current* e **virgin* para a próxima iteração;

- [Linha 31] Nessa linha são verificados, respectivamente, se houve atualização no valor de *ret* e também, se o endereço apontado por *virgin_map* corresponde ao endereço apontado pela variável global *virgin_bits*[40] e então, atualiza o valor de *bitmap_changed* de acordo;

- [Linha 33] Retorna o valor armazenado em *ret*: Igual a 2 caso novos ramos no mapa de execução tenham foram atingidos; Igual a 1 caso não tenham sido encontrados novos ramos, mas sinalizando um aumento do número de vezes em que certos ramos já conhecidos foram atingidos; E igual a 0, caso nenhuma mudança no mapa de execução tenha sido detectada.

A análise de complexidade temporal da função é a que segue:

1. **Inicialização de Variáveis**: A inicialização de variáveis é executada em um tempo constante, então:
$$T_{\text{init}} = O(1)$$

2. **Laço While**: A complexidade deste laço é determinada pelo número de iterações e pela complexidade do código dentro do laço:
$$T_{\text{laço}} = \frac{\text{MAP_SIZE}}{8} \times T_{\text{corpo da função}}$$
onde $T_{\text{corpo da função}}$ é o tempo para executar o corpo do laço.

3. **Corpo do Laço**: O corpo do laço consiste em operações *bitwise*, comparações e possivelmente outras operações simples. Todas estas operações são realizadas em tempo constante. Logo:
$$T_{\text{corpo da função}} = O(1)$$

4. **Instruções após o Laço**: Estas instruções também são executadas em tempo constante:
$$T_{\text{pós-laço}} = O(1)$$

[40]Embora isso não conste na documentação, considerando-se o fato de que a função *has_new_bits()* é chamada passando-se como parâmetro *virgin_map*, essa verificação na linha em questão é realizada com o intuito de se prevenir qualquer possível erro durante as operações de ponteiros realizadas.

Combinando todas as partes, a complexidade temporal total T da função é:

$$T = T_{\text{init}} + T_{\text{laço}} + T_{\text{pós-laço}}$$
$$T = O(1) + O(\text{MAP_SIZE}) + O(1)$$

Como a complexidade $O(\text{MAP_SIZE})$ domina os outros termos, podemos simplificar a expressão para:

$$T = O(\text{MAP_SIZE})$$

Resumindo, a complexidade temporal da função *has_new_bits()* é $O(\text{MAP_SIZE})$.

APÊNDICE B – CÓDIGO *ASSEMBLY* OTIMIZADO DESENVOLVIDO PARA *HAS_NEW_BITS()*

O código-fonte para a versão *assembly* otimizado para *has_new_bits()*pode ser visto na Figura 30, com uma análise linha-a-linha de seu funcionamento na sequência:

Figura 30 – Função *has_new_bits()* otimizada desenvolvida.

```
section .text
global has_new_bits_asm_optimz

; Argumentos:
; rdi = u8* trace_bits
; rsi = u8* virgin_bits
; rdx = u8* bitmap_changed

has_new_bits_asm_optimz:
    ; Inicializa o contador i
    mov ecx, 256      ; MAP_SIZE = 64KB

    ; Prepara a constante 0xFFFFFFFFFFFFFFFF
    mov rax, 0xFFFFFFFFFFFFFFFF
    movq xmm0, rax
    vpbroadcastq ymm3, xmm0  ; ymm3 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

    ; Inicializa ret = 0
    xor r8b, r8b

.loop:
    ; Prefetching dos dados da próxima e da subsequente iteração em cache
    prefetchnta [rdi + 512]
    prefetchnta [rsi + 512]

    xor r9b, r9b ; r9b = flag para bit encontrado
    xor r10, r10 ; r10 = offset total a partir da base

.iterate:
    vmovdqu ymm0, [rdi + r10] ; cur
    vmovdqu ymm1, [rsi + r10] ; vir

    ; Verifica se cur[i] não é zero
    vpcmpeqb ymm2, ymm0, ymm4  ; Compara cur com 0, resulta em 0xFF onde cur[i] != 0
    vpxor ymm2, ymm2, ymm3     ; Inverte o resultado, agora 0xFF onde cur[i] != 0

    ; Verifica se vir[i] é igual a 0xFF
    vpcmpeqb ymm1, ymm1, ymm3  ; Compara vir com 0xFF, resulta em 0xFF onde vir[i] == 0xFF

    ; Combinar as verificações
    vpand ymm0, ymm2, ymm1     ; 0xFF onde ambas as condições são verdadeiras
    vptest ymm0, ymm0
    jnz .set_found_bit         ; Pula se qualquer bit no ymm0 for 1

    add r10, 32
    cmp r10, 256
    jnz .iterate

.next_iteration:
    add rdi, 256
    add rsi, 256
    sub ecx, 64
    xor r10, r10 ; Zera o r10 para a próxima iteração
    xor r9b, r9b ; Zera o r9b para a próxima iteração
    jnz .loop

    ; Verifica se algum bit foi encontrado e .set_found_bit não foi chamado
    test r8b, r8b
    jnz .found_bit_set
    test r9b, r9b
    jz .end
    mov al, 1
    jmp .finalize

.found_bit_set:
    ; Atualiza bitmap_changed se necessário
    mov byte [rdx], 1
    mov al, 2

.finalize:
    vzeroall
    ret

.set_found_bit:
    ; Se um bit for encontrado, atualiza o mapa virgem e define a flag
    mov r8b, 2
    mov r9b, 1
    vmovdqu ymm1, [rdi + r10]
    vmovdqu ymm2, [rsi + r10]
    vpandn ymm2, ymm1, ymm2
    vmovdqu [rsi + r10], ymm2
    add r10, 32
    cmp r10, 256
    jnz .iterate
    jmp .next_iteration

.end:
    vzeroall
    ; Define o retorno da função
    mov al, r8b
    ret

```

Fonte: Autor, 2023

- [Linhas 1-2] Declaração e exposição global da função `has_new_bits_asm_-optimz`;
- [Linhas 4-6] Comentários explicativos dos argumentos da função, onde `rdi` é o mapa de *bits* de execução atual, `rsi` é o mapa de *bits* virgem (não testado) e `rdx` é um ponteiro para a variável que indica se o mapa foi alterado;
- [Linha 8] Definição do tamanho do mapa de *bits* através do registrador `ecx`, ajustado para 8192, que nesse exemplo, corresponde ao tamanho setado para o mapa de *bits* (64KB) dividido por 8, uma vez que 8 *bytes* (64 *bits*) são trabalhados a cada iteração do laço mais interno[41] (em *.iterate*);
- [Linha 11] Inicialização do registrador `r8b` para zero, usado para indicar se foi encontrada alguma alteração nos mapas de bits;
- [Linhas 13-20] Início do laço principal da função com o rótulo *.loop*. Dentro desse laço, os dados dos próximos blocos de memória são pré-carregados na cache para otimização de acesso à memória;
- [Linhas 22-31] Dentro do rótulo *.iterate*, realiza-se o desenrolamento do laço para processar 8 iterações de uma vez, utilizando operações SIMD para verificar se os *bits* do mapa de execução atual estão presentes no *virgin_map*;
- [Linhas 33-37] Após processar 256 *bits*, os ponteiros para os mapas de *bits* são incrementados e o contador do laço é decrementado;
- [Linhas 39-44] Teste e atualização da variável que indica a mudança do mapa de *bits*, caso a *flag* indicativa de alteração tenha sido definida;
- [Linhas 46-58] No rótulo *.set_found_bit*, o *virgin_map* é atualizado onde os *bits* correspondentes no mapa de execução atual estão definidos e a *flag* de alteração é ajustada;
- [Linhas 60-64] O rótulo *.end* contém instruções para limpar os registradores superiores YMM para evitar vazamentos de dados e preparar o valor de retorno da função;

[41] Para cada tamanho de *bitmap* testado, esse valor deve ser ajustado no algoritmo de acordo.

- [Linha 66] Final da função, onde o registrador `al` é definido com o valor do registrador `r8b` que contém a indicação se houve ou não alteração nos mapas de *bits* e é realizada a instrução de retorno da função.

APÊNDICE C – CÁLCULOS PARA DETERMINAÇÃO DO GANHO DE DESEMPENHO COMPARATIVO PARA AS VERSÕES NORMAL E OTIMIZADA DE *HAS_NEW_BITS()*.

Seguem os cálculos de melhoria percentual para cada tamanho de *bitmap* testado (valores em microssegundos):

- **64KB** (Tabela 6):

$$\begin{aligned}
\text{Iteração 1} &= \left(\frac{5.200 - 2.100}{5.200}\right) \times 100 = 59.62\% \\
\text{Iteração 10} &= \left(\frac{3.790 - 1.470}{3.790}\right) \times 100 = 61.21\% \\
\text{Iteração 100} &= \left(\frac{3.735 - 1.410}{3.735}\right) \times 100 = 62.25\% \\
\text{Iteração 1000} &= \left(\frac{3.777 - 1.405}{3.777}\right) \times 100 = 62.79\% \\
\text{Iteração 10000} &= \left(\frac{3.801 - 1.496}{3.801}\right) \times 100 = 60.62\% \\
\text{Iteração 100000} &= \left(\frac{3.916 - 1.604}{3.916}\right) \times 100 = 59.04\%
\end{aligned}$$

$$\begin{aligned}
\text{Média da Melhoria} &= \frac{59.62\% + 61.21\% + 62.25\% + 62.79\% + 60.62\% + 59.04\%}{6} \\
&= \frac{365.53\%}{6} \\
&\approx 60.92\%
\end{aligned}$$

- **128KB** (Tabela 7):

$$\text{Iteração 1} = \left(\frac{25.800 - 12.900}{25.800}\right) \times 100 = \frac{12.900}{25.800} \times 100 = 50.00\%$$
$$\text{Iteração 10} = \left(\frac{21.420 - 9.380}{21.420}\right) \times 100 = \frac{12.040}{21.420} \times 100 = 56.21\%$$
$$\text{Iteração 100} = \left(\frac{22.368 - 8.903}{22.368}\right) \times 100 = \frac{13.465}{22.368} \times 100 = 60.20\%$$
$$\text{Iteração 1000} = \left(\frac{12.117 - 3.812}{12.117}\right) \times 100 = \frac{8.305}{12.117} \times 100 = 68.52\%$$
$$\text{Iteração 10000} = \left(\frac{7.685 - 3.249}{7.685}\right) \times 100 = \frac{4.436}{7.685} \times 100 = 57.71\%$$
$$\text{Iteração 100000} = \left(\frac{7.807 - 3.253}{7.807}\right) \times 100 = \frac{4.554}{7.807} \times 100 = 58.33\%$$

$$\begin{aligned}\text{Média da Melhoria} &= \frac{50.00\% + 56.21\% + 60.20\% + 68.52\% + 57.71\% + 58.33\%}{6} \\ &= \frac{350.97\%}{6} \\ &\approx 58.50\%\end{aligned}$$

- **256KB** (Tabela 8):

$$\text{Iteração 1} = \left(\frac{18.000 - 8.400}{18.000}\right) \times 100 = \frac{9.600}{18.000} \times 100 = 53.33\%$$
$$\text{Iteração 10} = \left(\frac{16.070 - 7.170}{16.070}\right) \times 100 = \frac{8.900}{16.070} \times 100 = 55.42\%$$
$$\text{Iteração 100} = \left(\frac{16.122 - 6.945}{16.122}\right) \times 100 = \frac{9.177}{16.122} \times 100 = 56.94\%$$
$$\text{Iteração 1000} = \left(\frac{16.059 - 7.787}{16.059}\right) \times 100 = \frac{8.272}{16.059} \times 100 = 51.52\%$$
$$\text{Iteração 10000} = \left(\frac{15.945 - 7.010}{15.945}\right) \times 100 = \frac{8.935}{15.945} \times 100 = 56.03\%$$
$$\text{Iteração 100000} = \left(\frac{16.238 - 7.103}{16.238}\right) \times 100 = \frac{9.135}{16.238} \times 100 = 56.27\%$$

$$
\begin{aligned}
\text{Média da Melhoria} &= \frac{53.33\% + 55.42\% + 56.94\% + 51.52\% + 56.03\% + 56.27\%}{6} \\
&= \frac{329.51\%}{6} \\
&\approx 54.92\%
\end{aligned}
$$