Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Tese de Doutorado

Uma Técnica para Detectar Fraquezas de Código em Programas C

Raphael de Carvalho Muniz

Campina Grande, Paraíba, Brasil
03/2022

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Uma Técnica para Detectar Fraquezas de Código em Programas C

Raphael de Carvalho Muniz

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Wilkerson de Lucena Andrade e Patricia Duarte de Lima Machado
(Orientadores)

Campina Grande, Paraíba, Brasil

M966t Muniz, Raphael de Carvalho.
Uma técnica para detectar fraquezas de código em programas C / Raphael de Carvalho Muniz. – Campina Grande, 2022.
131 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2022.
"Orientação: Prof. Dr. Wilkerson de Lucena Andrade, Profa. Dra. Patricia Duarte de Lima Machado".
Referências.

1. Engenharia de Software. 2. Teste de Software. 3. Programas C. 4. Fraqueza. 5. Segurança – Sistemas de Software. I. Andrade, Wilkerson de Lucena. II. Machado, Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

RAPHAEL DE CARVALHO MUNIZ

UMA TÉCNICA PARA DETECTAR FRAQUEZAS DE CÓDIGO EM PROGRAMAS C

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do titulo de Doutor em Ciência da Computação.

Aprovada em: 04/03/2022

Prof. Dr. WILKERSON DE LUCENA ANDRADE, UFCG, Orientador

Profa. Dra. PATRICIA DUARTE DE LIMA MACHADO, UFCG, Orientadora

Prof. Dr. REINALDO CÉZAR DE MORAIS GOMES, UFCG, Examinador Interno

Profa. Dra. SABRINA DE FIGUEIRÊDO SOUTO, UEPB, Examinadora Interna

Prof. Dr. AVELINO FRANCISCO ZORZO, PUCRS, Examinador Externo

Prof. Dr. BALDOINO FONSECA DOS SANTOS NETO, UFAL, Examinador Externo

Documento assinado eletronicamente por **WILKERSON DE LUCENA ANDRADE**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 04/03/2022, às 18:17, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Sabrina de Figueiredo Souto**, **Usuário Externo**, em 05/03/2022, às 10:53, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Baldoino Fonseca dos Santos Neto**, **Usuário Externo**, em 05/03/2022, às 21:09, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **REINALDO CEZAR DE MORAIS GOMES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 07/03/2022, às 08:26, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **PATRICIA DUARTE DE LIMA MACHADO**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 07/03/2022, às 09:13, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **2146079** e o código CRC **285093C3**.

Referência: Processo nº 23096.006620/2022-50 SEI nº 2146079

Resumo

O investimento de empresas na implementação de meios para garantir a segurança em sistemas de software nos faz perceber tamanha importância deste tema. Todavia, garantir esta característica não é uma atividade trivial. Vários sistemas, como o Linux e OpenSSL, são implementados utilizando a linguagem C, e uma vulnerabilidade nesses sistemas pode impactar muitos usuários. No entanto, apesar dos esforços em aplicar técnicas e ferramentas com o objetivo de tornar sistemas de software mais seguros, esses sistemas ainda apresentam fraquezas de código, levando a códigos vulneráveis. O número de vulnerabilidades reportadas aumentou nos últimos anos, onde mais de 18 mil vulnerabilidades foram reportadas ao National Vulnerability Database (NVD) em 2020. Ferramentas de análise estática, como Flawfinder e Cppcheck, podem ajudar neste problema, dando indícios de alguns tipos de fraquezas. No entanto, este tipo de ferramenta apresenta uma alta taxa de falsos positivos, ou seja, um problema relatado em um programa quando nenhum problema realmente existe. Em síntese, neste trabalho apresentamos uma técnica que combina análise estática com teste de software para detectar fraquezas introduzidas no código-fonte durante os estágios iniciais de desenvolvimento de programas C. Consideramos que identificar problemas antecipadamente reduz o custo de correção. Nós implementamos esta técnica em um framework chamado Weaknesses Testing ou WTT. Por fim, nós realizamos dois estudos para avaliar a aplicação prática da técnica proposta. O primeiro estudo avaliou a técnica proposta com programas reais de código aberto, com o objetivo de detectar novas fraquezas. Nós avaliamos 103 warnings de 6 projetos diferentes e detectamos 22 fraquezas de três tipos: Buffer Overflow, Format String e Integer Overflow. Por sua vez, no segundo estudo avaliamos a técnica com um conjunto de exemplos de vulnerabilidades conhecidas. Nós avaliamos um total de 2.834 funções do conjunto de dados Juliet Test Suite com fraquezas CWE-190: Integer Overflow or Wraparound, CWE-191: Integer Underflow (Wrap or Wraparound) e CWE-369: Divide By Zero. Os resultados mostram evidências de que nossa técnica pode ajudar os desenvolvedores a antecipar a detecção de fraquezas em programas C, reduzindo a ocorrência de vulnerabilidade em versões operacionais.

Palavras-chave: Fraqueza, Teste de Software, Segurança, Programas C

Abstract

Companies' investment in implementing means to ensure security in software systems makes us realize the importance of this topic. However, ensuring this feature is not a trivial activity. Several critical systems, such as Linux and OpenSSL, are implemented using the C language, and a vulnerability in these systems may impact many users. However, despite the efforts to apply techniques and tools to make software systems more secure, these systems still have code weaknesses, leading to vulnerable code. The number of reported vulnerabilities has increased in the last years, where more than 18 thousand vulnerabilities were reported to the National Vulnerability Database (NVD) in 2020. Static analysis tools, such as Flawfinder and Cppcheck, may help in this problem, reporting some kinds of weaknesses. However, this kind of tool presents a high rate of false positives, i.e., an issue reported in a program when no problem actually exists. In summary, in this work we present a technique that combines static analysis with software testing to detect weaknesses introduced in the code during earlier development stages of C programs. We believe the earlier the weakness is detected, the lower is the cost to fix it. We implemented this technique in a framework named Weaknesses Testing or WTT. Finally, we carried out two studies to evaluate the practical application of the proposed technique. The first study evaluated the proposed technique with real open-source programs to detect new weaknesses. We evaluated 103 warnings from 6 different projects and detected 22 weaknesses of three kinds: Buffer Overflow, Format String, and Integer Overflow. On the other hand, in the second study, we evaluated the technique with a set of examples of known vulnerabilities. We evaluated a total of 2,834 functions from the Juliet Test Suite dataset with weaknesses CWE-190: Integer Overflow or Wraparound, CWE-191: Integer Underflow (Wrap or Wraparound), and CWE-369: Divide By Zero. The results show evidence that our technique can help developers anticipate the detection of weaknesses in C programs, reducing vulnerability in operational versions.

Keywords: Weakness, Software Testing, Security, C Programs.

Agradecimentos

Em minhas orações, sempre cito que sou uma pessoa muito abençoada por tudo que Deus me deu e, qualquer coisa que peça, estaria sendo egoísta. Por isso, sempre agradeço por minha saúde, pelo ar que respiro e por tudo aquilo que Ele me proporciona. Hoje, sou grato por mais uma conquista, a conclusão de mais uma fase da minha vida.

Agradeço também ao meu pai Zidailson Vieira Muniz, minha mãe Railma Lúcia de Carvalho Muniz e meu irmão Ramon de Carvalho Muniz por todo apoio e por estarem sempre ao meu lado. Todas as viagens à Campina Grande, quando eu não podia ir para casa (Mossoró) por causa das atividades do doutorado.

Agradeço à minha esposa Mariuchy Sammara de B. P. Franco por todo o companheirismo, dedicação e compreensão. Que possamos ser bons pais para o nosso filho que está por vir.

Agradeço principalmente aos meus orientadores Wilkerson de Lucena Andrade e Patrícia Duarte de Lima Machado pelo acolhimento quando decidi mudar de tema, motivação quando pensei em desistir e por todos os ensinamentos e paciência com os erros cometidos. Tenho plena convicção que sem ambos, não teria conseguido concluir esta fase da minha vida. Muito Obrigado.

Agradeço também a todos os amigos que fiz durante essa fase, tanto no SPLab quanto no SPG. Eles foram essenciais nos momentos de dúvidas, dicas e desabafo. Considero o convívio no laboratório muito importante para a construção do trabalho.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Segurança de sistemas é um tema importante e bastante discutido atualmente. No entanto, garantir essa característica não é uma tarefa trivial. Sem dúvida, estamos rodeados de programas que precisam apresentar um nível de segurança adequado para serem confiáveis para realizar transações bancárias ou mesmo uma simples conversa que terceiros não possam acessar. Grandes sistemas, como Linux e OpenSSL, são desenvolvidos usando a linguagem C, e uma vulnerabilidade nesses sistemas pode afetar muitos usuários.

É essencial evitar fraquezas em tais sistemas. Pfleeger et al. [94] definiram vulnerabilidade como sendo uma fraqueza no sistema que pode ser explorada por um atacante malicioso para causar perdas ou danos. Ao explorá-las, atacantes mal-intencionados podem, por exemplo, comprometer a disponibilidade do serviço ou expor informações confidenciais e pesquisar por exploração de dia zero. De acordo com Frei et al. [38], uma exploração de dia zero é uma exploração que tira proveito de uma vulnerabilidade no dia (ou antes) em que a vulnerabilidade é descoberta, ou seja, o fornecedor e o público têm zero dias para se prevenir da violação de segurança.

Nesse sentido, pesquisas têm sido realizadas para tratar de questões relacionadas à segurança. O *Common Vulnerabilities and Exposures (CVE)* é um projeto que compartilha informações sobre vulnerabilidades e como corrigi-las [1]. O *Common Weakness Enumeration (CWE)* é um projeto semelhante com foco em catalogar características de fraquezas de software [20]. O *Open Web Application Security Project (OWASP)* é uma fundação que tem o

[1] `http://cve.mitre.org/`

objetivo de melhorar a segurança de softwares [2]. Além disso, Howard et al. [53] propuseram os 24 pecados capitais relacionados à segurança de software e sugeriram como corrigi-los. Eles classificam os pecados em quatro categorias: aplicação da web, implementação, criptografia e rede.

Uma técnica utilizada com objetivo de detectar fraquezas é o teste de caixa branca. Godefroid et al. [45] propuseram uma ferramenta para aplicativos x86 do Windows. Por outro lado, Zhang et al. [117] apresentaram uma abordagem de teste baseada em rastreamento para a linguagem C. Eles reutilizaram casos de teste gerados a partir de métodos de teste anteriores para produzir rastreamentos de execução e detectar vulnerabilidades nos sistemas. Ambas as abordagens foram avaliadas em estudos empíricos. Como desvantagem, eles podem depender da arquitetura específica e apresentar limitações ao lidar com programas mais extensos. Estes trabalhos são apresentados em detalhes no Capítulo 6 desta tese.

Ferramentas de análise estática apresentam o potencial de apoiar desenvolvedores a encontrar e corrigir problemas no código [107; 102]. A análise estática pode ser útil na identificação de algum código suspeito que pode ser uma fraqueza do código. Existem trabalhos que propõem ferramentas que executam análise estática no contexto da linguagem C, como CPPCheck [3] e FlawFinder [4], que podem ajudar os desenvolvedores a detectar fraquezas. No entanto, as abordagens de análise estática geralmente executam análise léxica e procuram por correspondências em seu no banco de dados. Nesse sentido, as ferramentas de análise estática são muitas vezes imprecisas quando executadas sozinhas, causando falsos positivos excessivos, ou seja, um problema relatado em um programa quando não existe problema [16].

Considerando a utilização da análise estática, precisamos usar uma estratégia adicional para reduzir alarmes falsos, confirmando se um *warning* é uma fraqueza. O teste de software pode preencher essa lacuna fornecendo evidências das fraquezas. Ademais, pode ser executado desde os estágios iniciais do desenvolvimento do software até os estágios de evolução.

Fuzz testing se tornou uma das técnicas de teste mais eficazes para encontrar problemas de segurança em sistemas de software [112]. Esta técnica vem sendo aplicada por grandes empresas como Microsoft e Google em testes de segurança e garantia de qualidade [108;

[2] https://owasp.org/
[3] http://cppcheck.sourceforge.net/
[4] https://www.dwheeler.com/flawfinder/

30]. O método de *fuzzing* é baseado na técnica de injeção de falhas. Essa abordagem envia dados de entrada para o software em teste com o objetivo de detectar vulnerabilidades [58].

Fuzzing pode detectar muitas vulnerabilidades, como *buffer overflow*, *integer overflow*, formatação de *strings*, condição de corrida, injeção de SQL, *Cross-Site Scripting (XSS)*, execução remota de comando, ataques de sistema de arquivos e vulnerabilidades de vazamento de informações. Desta forma, trabalhos vêm sendo propostos utilizando *fuzz testing* no contexto de detecção de vulnerabilidades. Wen et al. [114] apresentaram uma técnica de *fuzz testing* guiada pelo uso de memória, que apresenta o objetivo de detectar *bugs* relacionado ao consumo de memória. Wang et al. [113] apresentaram um *fuzzer* guiado por para detectar vulnerabilidades que violam determinadas propriedades de estado de tipo.

Em uma pesquisa anterior [87], desenvolvedores C foram questionados sobre fraquezas de código em programas C e foram identificadas algumas observações interessantes. Foram identificadas observações, como por exemplo, (i) os desenvolvedores conhecem as vulnerabilidades que apresentam mais ocorrência; (ii) muitos desenvolvedores usam ferramentas de análise estática para encontrar pontos fracos; (iii) vários desenvolvedores seniores não conseguiram detectar estouros de *buffer* e inteiros. Muitos desenvolvedores questionados afirmam ter mais de cinco anos de experiência. As observações obtidas com o estudo nos levam a fazer alguns questionamentos sobre o tema. Por que vários programas C ainda são afetados por perdas causadas por vulnerabilidades no código? Existe uma maneira eficaz de evitar ou detectar pontos fracos em programas C antes que eles sejam explorados? Segundo o NVD, em 2020, mais de 18 mil vulnerabilidades foram confirmadas, aumentando nos últimos anos [5].

1.1 Exemplo Motivante

Nesta seção, descrevemos um exemplo de fraqueza relacionada a defeitos de formatação de *strings* de acordo com o projeto *Common Weaknesses Enumeration (CWE)* [6]. Além disso, apresentamos como esse tipo de fraqueza pode afetar um sistema de software. Ademais, apresentamos algumas abordagens que podem auxiliar na detecção deste tipo de problema

[5] `https://nvd.nist.gov`
[6] `https://cwe.mitre.org/`

relacionados à segurança.

O código fonte 1.1 apresenta um trecho de código simplificado do arquivo `htdbm.c` do projeto Apache (commit r814852) que foi reportado ao Bugzilla [7] (id 30586). O trecho de código citado apresenta a função `fprintf` (linha 8) que é utilizada para imprimir uma sequência de caracteres formatada. No entanto, a função não apresenta a formatação para sequência de caracteres `cmnt` a ser impressa. Dessa maneira, se um invasor puder manipular a entrada da função, poderá causar a execução de código arbitrário. Por exemplo, se o invasor passar como entrada a *string* "%x", o programa exibirá o endereço de memória. Este tipo de fraqueza é classificado pelo projeto CWE como *Use of Externally-Controlled Format String (CWE-134)*. A presença deste tipo de fraqueza pode permitir que um invasor explore o programa, causando danos, como disponibilização de informações que podem simplificar seriamente a exploração do programa ou resultar na execução de código arbitrário. Christey et al. [19] listaram os 25 erros de software considerados mais perigosos e o tipo de fraqueza apresentado no código fonte 1.1 corresponde ao 23º erro desta lista.

Código Fonte 1.1: Versão simplificada de `htdbm.c` arquivo do projeto Apache.

```
1  static apr_t htdbm_list(...){
2    char *cmnt;
3    int i = 0;
4    //...
5    while (key.dptr != NULL) {
6      //...
7      if (cmnt)
8      fprintf(stderr, cmnt + 1);9
       //...
10     if (rv != APR_SUCCESS)
11     fprintf(stderr, "Failed Next Key \n");12
       ++i;
13   }
14   return APR_SUCCESS;
15 }
```

Ferramentas de análise estática, como o Flawfinder, podem identificar a fraqueza apre-

[7] `https://bz.apache.org/bugzilla/show_bug.cgi?id=30586`

sentada no Código Fonte 1.1. No entanto, devido aos vários alarmes falsos apresentados com a utilização das ferramentas de análise estática, uma análise manual de todos os problemas detectados é impraticável sem uma estratégia bem definida.

Executamos análise estática no código original do trecho de código apresentado no Código Fonte 1.1 e o relatório da ferramenta apresentou muitos falsos positivos em comparação com os verdadeiros positivos. O código real apresenta 396 linhas e a ferramenta Flawfinder reportou 10 pontos fracos. A Figura 1.1 apresenta o relatório da execução de análise estática por meio da ferramenta Flawfinder. Neste sentido, realizamos uma análise manual e, com base em nossos conhecimentos, verificamos que, das fraquezas totais relatadas pelo Flawfinder, confirmamos apenas o exemplo apresentado no Código Fonte 1.1 como sendo verdadeiro positivo. As demais fraquezas reportadas foram classificadas como alarmes falsos.

A ferramenta de análise estática indica que o tipo da fraqueza está relacionado à formatação de cadeias de caracteres, a função `fprintf` como sendo a portadora da fraqueza e a linha de código (linha 237) que apresenta a fraqueza. Além disso, o Flawfinder classifica o nível de risco de cada possível fraqueza identificada no código e calcula o nível de risco avaliando a função e os valores dos parâmetros. Ela considera, por exemplo, que cadeias constantes são frequentemente menos arriscadas do que cadeias totalmente variáveis em muitos contextos. Neste sentido, Flawfinder classificou a fraqueza descrita no código fonte 1.1 como nível de risco quatro, em uma escala entre zero e cinco.

Como pudemos perceber, ferramentas de análise estática podem apontar algumas pistas de fraquezas no código. No entanto, os desenvolvedores precisam executar outras estratégias com o objetivo de confirmar a fraqueza ou tentar explorá-la no programa que está sendo testado. Como é possível verificar no relatório apresentado na Figura 1.1, a própria ferramenta de análise estática confirma essas informações, onde destaca *"Not every hit is necessarily a security vulnerability"*. Além disso, afirma: *"There may be other security vulnerabilities; review your code!"*. As ferramentas de análise estática reportam fraquezas como, definição de variáveis de tamanho estático ou utilização das funções *strcpy, memcpy e strlen*. No entanto, sabemos que nem sempre a presença dessas características significa a presença de vulnerabilidades. Os desenvolvedores podem garantir que, apesar do uso dessas funções, o código foi preparado para não apresentar vulnerabilidade.

Em geral, as abordagens de teste de software podem se beneficiar das informações sobre

```
Flawfinder version 1.31, (C) 2001-2014 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 169
Examining htdbm.c

FINAL RESULTS:

htdbm.c:237:  [4] (format) fprintf:
  If format strings can be influenced by an attacker, they can be exploited
  (CWE-134). Use a constant for the format specification.
htdbm.c:252:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
htdbm.c:318:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
htdbm.c:163:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
htdbm.c:167:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
htdbm.c:173:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
htdbm.c:183:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
htdbm.c:197:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
htdbm.c:268:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
htdbm.c:268:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).

ANALYSIS SUMMARY:

Hits = 10
Lines analyzed = 475 in approximately 0.01 seconds (35589 lines/second)
Physical Source Lines of Code (SLOC) = 396
Hits@level = [0]   0 [1]   7 [2]   2 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]  10 [1+]  10 [2+]   3 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 25.2525 [1+] 25.2525 [2+] 7.57576 [3+] 2.52525 [4+] 2.52525 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming for Linux and Unix HOWTO'
(http://www.dwheeler.com/secure-programs) for more information.
```

Figura 1.1: Relatório da ferramenta de análise estática Flawfinder aplicada ao arquivo `htdbm.c`.

quais partes do código testar, e as ferramentas de análise estática têm o potencial de ajudar os desenvolvedores fornecendo essas informações [47; 102; 107]. Portanto, acreditamos que a aplicação conjunta de análise estática e técnicas de teste de software pode ser uma estratégia promissora para abordar a detecção de fraquezas em código fonte C. Podem ser aplicados desde os estágios iniciais do processo de desenvolvimento, mesmo no nível de função. Na próxima seção, apresentamos o objetivo e as questões de pesquisa endereçadas que guiaram

o desenvolvimento deste trabalho.

1.2 Objetivo e Questões de Pesquisa

Este trabalho de doutorado tem o objetivo de propor uma técnica baseada em análise estática, automação de casos de teste e *fuzz testing* para detectar fraquezas de código durante os estágios iniciais do processo de desenvolvimento de software do ponto de vista dos desenvolvedores no contexto de programas desenvolvidos em linguagem C. Considerando o objetivo definido, foram endereçadas as seguintes questões de pesquisa:

Questão de pesquisa 1: *Quais os tipos de fraquezas podem ser detectados utilizando teste de software em programas C?*

Visando responder esta questão de pesquisa, realizamos uma revisão sistemática da literatura com o objetivo de identificar quais tipos de fraquezas poderiam ser detectados utilizando técnicas de teste de software. Além disso, foi possível identificar técnicas utilizadas na detecção de fraquezas de software no contexto de sistemas desenvolvidos utilizando a linguagem C (Seção 6.1). Ademais, apresentamos uma discussão detalhada dos trabalhos considerados mais relevantes (Seção 6.2).

Questão de pesquisa 2: *Teste de software combinado com a análise estática pode fornecer a base para uma técnica de detecção de fraquezas eficaz?*

Para responder essa questão de pesquisa, apresentamos uma técnica para detectar fraquezas de código fonte utilizando análise estática, automação de casos de teste e *fuzz testing*. Os detalhes da técnica proposta são apresentados no Capítulo 3. Ademais, avaliamos técnica proposta em dois estudos. No primeiro estudo (Capítulo 4), estávamos interessados em detectar novas fraquezas em sistemas reais de código aberto. No segundo estudo, avaliamos uma base de código com exemplos de fraquezas confirmadas (Capítulo 5).

1.3 Metodologia

Em geral, a metodologia utilizada para desenvolver este trabalho é descrita da seguinte forma:

i Com o objetivo de alcançar a resposta da primeira questão de pesquisa definida neste

trabalho de tese, realizamos uma revisão de trabalhos que apresentaram técnicas de teste para detectar tipos específicos de fraquezas no contexto de sistemas configuráveis. No entanto, não foi possível identificar técnicas de teste aplicadas neste contexto. Porém, identificamos trabalhos que apresentaram técnicas de teste no contexto da linguagem C que, potencialmente, poderiam ser discutidos. Desta forma, com a evolução da pesquisa foi decidido considerar todos os sistemas desenvolvidos em C. É importante mencionar que esta revisão foi constantemente atualizada durante o desenvolvimento deste trabalho.

Após a identificação dos tipos de fraquezas que poderiam ser detectadas utilizando teste de software, identificamos os passos necessários para responder a segunda questão de pesquisa definida neste trabalho de tese:

i Investigação de características comuns a fraquezas de determinados tipos;

ii Criação de guias para implementação de casos de teste abordando as principais características de cada fraqueza;

iii Seleção de uma ferramenta para execução de *fuzz testing*;

iv Ferramentas de análise estática foram estudadas e uma foi escolhida para execução do estudo;

v Definição de uma técnica de teste para detecção de fraquezas utilizando análise estática e *fuzz testing*;

vi Implementação de uma versão inicial da técnica em um *framework*, onde foram consideradas as fraquezas *Buffer Overflow*, *Integer Overflow* e *Format String*;

vii Execução de um estudo para avaliação do WTT com sistemas reais. Por meio da execução do estudo, foi possível responder questionamentos que nos permitiu avaliar se é possível detectar fraquezas de código em programas reais C, utilizando a técnica proposta;

viii Implementação de *templates* de casos de teste de acordo com os guias propostos;

ix Implementação de uma nova versão do WTT considerando geração automática de casos de teste com exemplos de código do conjunto de dados *Juliet Test Suite*;

x Execução de um estudo para avaliação da segunda versão do WTT com o conjunto de dados *Juliet Test Suite*. A execução deste estudo nos permitiu avaliar tanto a execução da técnica proposta com exemplos de vulnerabilidades conhecidas, como a geração automática de casos de teste.

1.4 Contribuições

Em resumo, neste trabalho de doutorado obtivemos as seguintes contribuições:

1. Uma revisão sistemática da literatura pela qual foi possível identificar as fraquezas mais recorrentes, as fraquezas que podem ser detectadas por técnicas específicas e as técnicas/ferramentas que vêm sendo aplicadas na detecção de fraquezas;

2. A definição de guias para implementação de casos de teste para cinco tipos de CWE, são eles: *CWE-120: Buffer Copy without Checking Size of Input, CWE-134: Use of Externally-Controlled Format String, CWE-190: Integer Overflow or Wraparound, CWE-191: Integer Underflow e CWE-369: Divide By Zero*;

3. Implementação de *templates* para geração automática de casos de teste utilizando os guias propostos;

4. Uma técnica baseada em análise estática, automação de casos de teste e *fuzz testing* para detectar fraquezas no contexto de programas C. Esta técnica foi implementada em um *framework* chamado *Weaknesses TesTing ou WTT*;

5. Um estudo sobre a utilização do WTT em programas reais. Este estudo resultou na publicação do artigo *Towards a Technique to Detect Weaknesses in C Programs* no *XXXV Simpósio Brasileiro de Engenharia de Software* [8];

6. Uma avaliação empírica com o conjunto de dados *Juliet Test Suite*, uma coleção de exemplos de código fonte que apresentam vulnerabilidades sintetizados em diferentes CWE [92].

[8] `https://dl.acm.org/doi/abs/10.1145/3474624.3474633`

Em resumo, avaliamos, da teoria à prática, a detecção de fraquezas de código no contexto de programas desenvolvidos em linguagem C. As evidências empíricas alcançadas sugerem que WTT pode apoiar os desenvolvedores na detecção de fraquezas durante os estágios iniciais de desenvolvimento de programas C.

1.5 Esboço da Tese

As demais partes deste documento estão estruturadas da seguinte forma:

Capítulo 2: Fundamentação Teórica. Este capítulo fornece a base teórica necessária para entendimento deste trabalho, incluindo conceitos sobre teste de software, segurança de sistemas, análise estática e *fuzz testing*.

Capítulo 3: Um Técnica para Detectar Fraquezas em Programas C. Este capítulo apresenta uma técnica para detecção de fraquezas no contexto de programas C, utilizando análise estática e *fuzz testing*.

Capítulo 4: Um Estudo Avaliativo com Sistemas Reais. Este capítulo apresenta um estudo avaliativo da técnica proposta nesta tese com um conjunto de programas reais.

Capítulo 5: Um Estudo Empírico com o Conjunto de Dados *Juliet Test Suite*. Este capítulo apresenta um estudo empírico da técnica proposta com um conjunto de arquivos que apresentam vulnerabilidades conhecidas.

Capítulo 6: Estado da Arte. Este capítulo apresenta uma revisão sistemática da literatura e uma revisão de trabalhos relevantes sobre análise estática, *fuzz testing*, white-box testing, execução simbólica e métricas de software.

Capítulo 7: Considerações Finais. Este capítulo final apresenta as conclusões e perspectivas para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo tem como objetivo principal fornecer uma fundamentação teórica relacionada aos conceitos essenciais discutidos nesta tese. Abordamos conceitos como teste de software (Seção 2.1), segurança de sistemas (Seção 2.2), enfatizando testes de segurança, e análise estática (Seção 2.3) e *fuzz testing* (Seção 2.4).

2.1 Teste de Software

Teste de software é uma das atividades de verificação e validação (V&V) que tem o objetivo de identificar evidências de defeitos inseridos durante qualquer fase de desenvolvimento ou manutenção de sistemas de software [98]. De acordo com o *SWEBOK 3.0 - IEEE Software Engineering Body of Knowledge* [12], o teste de software consiste na verificação dinâmica do comportamento de um programa em um conjunto finito de casos de teste, adequadamente selecionados do domínio de execuções geralmente infinito, em relação ao comportamento esperado.

No contexto de teste de software conceitos como **erro**, **falta**, **defeito** e **falha** são amplamente utilizados. De acordo com o padrão *1044-2009 - IEEE Standard Classification for Software Anomalies* [49], **erro** é uma atividade humana que produz um resultado incorreto. A manifestação deste erro corresponde a uma **falta**. **Defeito** é uma imperfeição ou deficiência em um produto de trabalho onde esse produto de trabalho não atende aos seus requisitos ou especificações e precisa ser reparado ou substituído. **Falha** corresponde ao término da capacidade de um produto de realizar uma função exigida ou de sua incapacidade de funcionar

dentro dos limites previamente especificados.

Jorgensen [57] afirma que determinar um conjunto de casos de teste para o item a ser testado é a essência do teste de software. Cada caso de teste deve ter pelo menos informações sobre **Entradas:** (i) Condições que devem ser satisfeitas antes da execução do teste; (ii) As entradas reais são escolhidas para testar o sistema; e **Saídas:** (i) Pós-condições que devem ser satisfeitas após a execução do teste; (ii) A saída real produzida pelo sistema em teste. No entanto, um caso de teste completo deve apresentar um identificador de caso de teste, uma breve declaração de propósito (uma regra de negócios), descrição das pré-condições, as entradas, as saídas esperadas, uma descrição das pós-condições esperadas e um histórico de execução.

Ainda de acordo com Utting e Legeard [105], existem muitos tipos de teste. Eles apresentaram uma forma, adaptada de Tretmans [104], de classificar vários tipos de teste entre três dimensões. A Figura 2.1 apresenta esta classificação. O eixo *z* apresenta a escala do software em teste (*Software Under Test ou SUT*), variando de pequenas unidades até todo o sistema. Ao longo deste eixo, temos: (i) teste de unidade, que envolve o teste de uma única unidade por vez, como um único procedimento, função ou classe. (ii) O teste de componentes, que tem o objetivo de testar cada componente/subsistema separadamente. (iii) O teste de integração, que apresenta o objetivo de garantir que vários componentes funcionem corretamente juntos. Finalmente, (iv) o teste do sistema, que tem o objetivo de testar o sistema como um todo. Já o eixo *x* apresenta diferentes características que podemos considerar testar, são elas (i) teste funcional ou teste comportamental, (ii) teste de robustez, (iii) Teste de performance e (iv) Teste de usabilidade. Finalmente, o eixo *y* apresenta as informações utilizadas para projetar os testes, temos: (i) o teste de caixa preta, que significa que não utilizamos informações sobre a estrutura interna (detalhes da implementação) do sistema. Neste caso, projetamos os casos de teste com informações sobre os requisitos. Por outro lado, (ii) o teste de caixa branca, onde projetamos os casos de teste utilizando detalhes de implementação do código.

Ainda sobre o tipo de informação utilizada na elaboração dos casos de teste, existe uma terceira possibilidade que seria uma abordagem híbrida entre o teste de caixa branca e de caixa preta, denominada teste de caixa cinza [69]. A ideia deste tipo de teste é acessar o sistema como um usuário (caixa preta), porém, com conhecimento da base de código e do

sistema (caixa branca). Utilizando esse conhecimento, o testador caixa cinza pode escrever testes caixa preta mais direcionados.

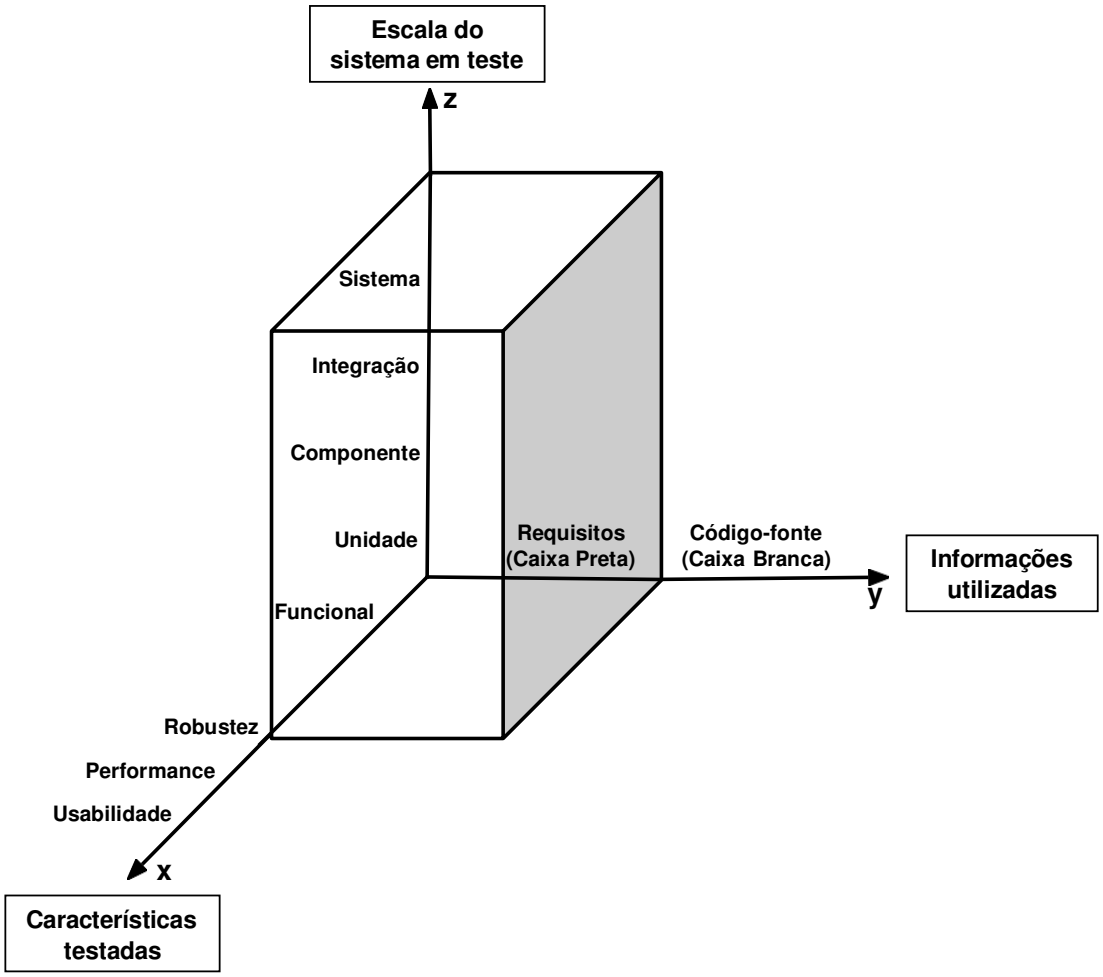

Figura 2.1: Diferentes tipos de teste [105].

2.1.1 Teste de Unidade

O teste de unidade especificamente, tem o objetivo de garantir que as menores "unidades" (métodos e funções individuais) operem de acordo com o esperado. É possível verificar se os métodos executam exatamente o que se espera deles, com uma entrada específica. Neste sentido, para se aplicar teste de unidade é necessário o conhecimento da estrutura interna da aplicação em teste [69].

De acordo com Ammann e Offutt [9], teste de unidade é projetado para avaliar as unidades produzidas durante a fase de implementação do processo de desenvolvimento e cor-

responde ao nível "mais baixo" de teste. Um software naturalmente apresenta comunicação entre seus elementos componentes, como funções, procedimentos ou classes. Estes elementos podem não apresentar um correto funcionamento em virtude de outras unidades de código. Neste sentido, o teste unitário deve verificar uma unidade localmente sem a preocupação com aspectos externos à unidade em teste. Desta forma, ao detectar uma falha no teste, é possível garantir que a falha corresponde ao código da unidade em teste.

Laboon [69] destaca alguns motivos para aplicar teste de unidade, são eles:

- **Detecção de problemas nos estágios iniciais do processo de desenvolvimento:** Testes unitários normalmente são escritos durante a fase de implementação, possibilitando a detecção antecipada de problemas. Vale a pena lembrar que quanto mais cedo um problema for encontrado em um processo, mais barato, mais rápido e mais fácil será de consertá-lo.

- **Rápida identificação e correção de problemas:** Além de se avaliar unidades isoladas do sistema, o desenvolvedor tem a possibilidade de poder corrigir um problema imediatamente após a sua detecção.

- **Rápida execução:** Apresenta um tempo de execução menor, em comparação a outros níveis mais altos de teste que consideram a integração de unidades ou o sistema completo.

- **Desenvolvedores entendem seu código:** Escrever casos de teste permite ao desenvolvedor entender melhor o comportamento esperado da função.

- **Documentação viva:** Possibilita a avaliação e identificação se requisitos permanecem válidos ou não.

- **Implementação alternativa:** A aplicação de testes de unidade permite que o desenvolvedor avalie seu código de outro ponto de vista.

- **Mudanças no código causaram problemas em outros lugares:** Um conjunto de teste relativamente completo, permite o desenvolvedor identificar uma alteração feita causou problemas em outras partes do código.

Muitas vezes, quando precisamos testar uma unidade de código, esta unidade apresenta dependências de elementos que foram definidos em outros blocos de comandos ou nem mesmo foram implementados ainda. Quando temos este tipo de cenário, precisamos substituir todas essas dependências por *test doubles* ou dublês de teste [61]. De acordo com Ammann e Offutt [9], dublês de teste correspondem a componentes de software que implementa funcionalidade parcial e é utilizado para substituir o componente de software original durante a execução dos testes. Os principais motivos para sua utilização são:

i Para casos em que as dependências ainda não foram implementadas, pode ocasionar problemas de testabilidade se as funcionalidades das dependências são mandatórias para testar outras partes do código.

ii Componentes que implementam ações irrecuperáveis, como em um sistema financeiro ou envio de um e-mail.

iii Casos em que componentes dependem de recursos não confiáveis ou imprevisíveis, como conexões de rede. Este tipo de caso pode tornar os testes não determinísticos.

iv Dublês de teste podem também ser utilizados em casos de dependências externas ao sistema, que torna o teste significativamente mais lento. Para estes casos, sua utilização é importante, principalmente, quando os testes são executados com frequência.

De acordo com Khorikov [61], existem alguns tipos de dependências falsas (*test doubles*) que podem ser utilizadas para executar testes. São utilizados, de forma geral, com o objetivo de facilitar os testes substituindo uma dependência real, que pode ser de difícil utilização. Khorikov [61] agrupa os dublês de teste em dois tipos: *mocks*, utilizados para emular e verificar interações futuras, que correspondem a chamadas que o código em teste realiza às suas dependências para alterar seu estado; e *stubs*, utilizados para emular interações recebidas, correspondentes às chamadas que o código em teste realiza às suas dependências para obter dados de entrada.

2.1.2 Seleção de Teste

Teste de conformidade é um tipo de teste em que o comportamento de um sistema é testado sistematicamente em relação à especificação de comportamento funcional do sistema [103].

Consiste em verificar se o comportamento de uma implementação real de um sistema (IUT para *Implementation Under Test*) está correta em relação a uma especificação [28].

O teste de conformidade relaciona uma IUT e uma especificação por meio da relação **conforms-to** $\subseteq IMPS \times SPECS$, onde $IMPS$ representa o universo de implementações e $SPECS$ especificações. Neste sentido, IUT **conforms-to** *s*, se e somente se, IUT é uma implementação correta das especificações *s*.

A relação **conforms-to** é difícil de ser verificada por meio de testes e as implementações geralmente são inadequadas para o raciocínio formal. Portanto, é assumida uma hipótese de teste em que qualquer IUT pode ser modelada por um objeto formal $i_{IUT} \in MODS$, sendo $MODS$ o universo de modelos [10]. Neste sentido, uma relação de implementação **imp** $\subseteq MODS \times SPECS$ é definida de modo que uma IUT **conforms-to** *s* se i_{IUT} **imp** s.

Considerando $TESTS$ o domínio de casos de testes $t \in TESTS$ é um caso de teste. Então $EXEC(t, IUT)$ representa o processo de execução do caso de teste t na implementação IUT, tendo como resultado um conjunto de observações do domínio OBS. Em virtude do não determinismo, podemos notar que a execução do teste pode envolver várias execuções de $EXEC(t, IUT)$. Supondo que função de observação que modela formalmente $EXEC(t, IUT)$ seja definida como $obs : TESTS \times MODS \rightarrow P(OBS)$. Então, $\forall IUT \in IMPS \; \exists i_{IUT} \in MODS \; \forall t \in TESTS.EXEC(t.IUT) = obs(t, IUT)$, de acordo com a hipótese de teste.

Sendo uma família de funções de veredicto $v_t : P(OBS) \rightarrow \{$**passa**, **falha**$\}$ que pode ser abreviado para IUT **passa** $t \Leftrightarrow_{def} v_t(EXEC(t, IUT)) =$ **passa**. Assim, para qualquer suíte de teste $T \subseteq TESTS$, IUT **passa** $T \Leftrightarrow \forall t \in T$. IUT **passa** t e IUT **falha** $T \Leftrightarrow \neg(IUT$ **passa** $T)$.

De acordo com Tretmans [103], um conjunto de testes é chamado de completo quando este pode distinguir exatamente entre todas as implementações conformes e não conformes. Considerando testes práticos, um conjunto de testes completo é um requisito impraticável. Neste sentido, são necessários requisitos de teste mais fracos. Uma suíte de teste é dita *sound* se todas as implementações corretas e, possivelmente, algumas implementações incorretas passam, ou seja, qualquer implementação defeituosa detectada não está em conformidade, mas não o contrário.

Fernandez et al. [33] citaram duas limitações importantes ao teste de conformidade: (i)

Necessita de uma especificação formal exaustiva, considerando que a conformidade é definida pela relação entre especificação e qualquer IUT que, exibindo comportamentos inesperados, seria rejeitado. (ii) Não é considerada uma relação flexível, considerando que não é possível adaptá-la para testar uma funcionalidade particular.

Vries e Tretmans [28] propuseram um framework formal para propósito de teste. Diferente do teste de conformidade que visa aceitar ou rejeitar uma implementação, o framework proposto visa observar o comportamento desejado que não está necessariamente relacionado a um comportamento exigido ou corretude do sistema. A confiança na corretude é aumentada se um comportamento é observado. Por outro lado, não é possível chegar a nenhuma conclusão definitiva.

Vries e Tretmans [28] utilizaram o termo "observation objective" para se referir a propósito de teste. No entanto, o termo propósito de teste é mantido nesta tese. Propósitos de teste descrevem observações que desejamos visualizar testando uma implementação. Estão relacionados a implementações que são capazes de exibir estas observações por um conjunto bem escolhido de experimentos. Isso é definido pela relação **exhibits** $\subseteq IMPS \times TOBS$, onde $TOBS$ é o universo de propósitos de teste.

Para descrever a relação **exhibits**, também consideramos a hipótese de teste apresentada anteriormente, definindo a relação de revelação **rev** $\subseteq MODS \; x \; TOBS$, de modo que, para $e \in TOBS$, IUT **exhibits** e se e somente se i_{IUT} **rev** e, com $i_{IUT} \in MODS$ de IUT.

Sendo uma função veredicto $H_e : P(OBS) \Leftrightarrow \{\textbf{hit}, \textbf{miss}\}$ define se um propósito de teste é exibido por uma implementação. Então, IUT **hits** e por $t_e =_{def} H_e(EXEC(t_e, IUT)) = \textbf{hit}$. Isso é estendido a um conjunto de testes T_e como IUT **hits** e por $T_e =_{def} H_e(\cup\{EXEC(t_e, IUT) \mid t \in T_e\}) = \textbf{hit}$.

Um conjunto de teste que pode distinguir entre todas as implementações apresentam e não apresentam um propósito de teste é dito *e-complete*, de modo que, IUT **exhibits** e se e somente se IUT **hits** e por T_e.

Um conjunto de teste é considerado *e-exhaustive* quando é possível detectar apenas implementações não exibidoras (IUT **exhibits** e implica em IUT **hits** e por T_e). Finalmente, um conjunto de teste é considerado *e-sound* quando é possível detectar apenas implementações exibidoras (IUT **exhibits** e de IUT **hits** e por T_e.

É possível notar semelhanças entre a finalidade das suítes de teste *sound* e das suítes de

teste de *e-sound*, embora as implicações sejam relativamente invertidas. As suítes de teste consideradas *sound* podem revelar a presença de implementações não conformes, enquanto suíte de teste *e-sound* pode revelar o comportamento pretendido.

Utting e Legeard [105] resumem teste de software como sendo atividade de executar um sistema para detectar defeitos. É importante lembrar que existem outras técnicas, diferentes e complementares, que visam melhoria de qualidade de software, como análise estática, inspeções e revisões, além de depuração e correção de erros que são executados após a detecção de um defeito. De forma geral, teste de software tem como foco principal a qualidade de produtos de software. Neste sentido, diversas estratégias de teste vêm sendo aplicadas com sucesso na detecção de problemas relacionados à segurança de sistemas. As próximas seções abordarão com mais detalhes este tema.

2.2 Segurança de Sistemas

Atualmente, a segurança de sistemas é um tema bastante discutido não apenas na comunidade da computação, mas por todos que possuem dados confiados a algum sistema de software. Nessas discussões a respeito da segurança dos sistemas, emergem diversos conceitos que consideramos importantes definir a fim de melhor compreendermos este assunto.

Chess e West [16] definem a segurança do software como a prática de construir software para ser seguro e funcionar corretamente contra ataques maliciosos. Alguns fatores podem facilitar este tipo de ataque, como a presença de defeitos inseridos no código por um erro humano e não detectados durante a execução do processo de teste do software. Pfleeger et al. [94] afirmaram que um erro humano na execução de alguma atividade de software pode levar a uma falha ou a um passo incorreto, comando, processo ou definição de dados incorreta em um programa de computador, design ou documentação.

Erros humanos podem levar os desenvolvedores a inserirem vulnerabilidades no programa, facilitando atividades maliciosas. Krsul [66] definiu uma vulnerabilidade de software como uma instância de um erro na especificação, desenvolvimento ou configuração de software de forma que sua execução pode violar a política de segurança. Pfleeger et al. [94] definiram vulnerabilidade como uma fraqueza no sistema que pode ser explorada para causar perdas ou danos. Com base nessas definições, concluímos que *vulnerabilidade* é uma

fraqueza que pode ser explorada por um agente malicioso. Nesse sentido, nem todas as fraquezas são vulnerabilidades, mas todas as vulnerabilidades são fraquezas. Por exemplo, um sistema pode conter uma fraqueza que pode não ser explorada. Nesse caso, essa fraqueza não é uma vulnerabilidade. Usamos o termo *fraqueza* para nos referir a um defeito no código que um agente malicioso pode explorar.

Não há senso comum entre os pesquisadores de segurança sobre a abordagem adequada em relação à detecção de vulnerabilidades. Cada vulnerabilidade tem sua preocupação dependendo de seu tipo, e não é possível adotar uma estratégia única para detectá-las. No entanto, de acordo com Sutton et al. [101], existem três abordagens para detectar vulnerabilidades de segurança em alto nível: caixa branca, caixa preta e caixa cinza. Essas abordagens são definidas de acordo com o nível de acesso aos recursos que o testador possui.

Dowd et al. [29] agruparam vulnerabilidades em três classes de vulnerabilidade principais, com base nas fases de desenvolvimento de software: vulnerabilidades de projeto, vulnerabilidades de implementação e vulnerabilidades operacionais. Na fase de design do software, os requisitos do usuário são selecionados e traduzidos para uma especificação do sistema que, por sua vez, é traduzida em um design de alto nível. Vulnerabilidades de projeto, podem ocorrer quando os requisitos de segurança não são coletados ou traduzidos corretamente na especificação ou quando as ameaças não são identificadas corretamente. Uma vez executado, o projeto proposto é implementado em código na fase de implementação do software. De acordo com Dowd et al. [29], vulnerabilidades de implementação podem ocorrer se a implementação se desviar do design para resolver discrepâncias técnicas. No entanto, situações exploráveis ocorrem por artefatos técnicos, nuances da plataforma e do ambiente de linguagem usado para desenvolvimento. Finalmente, as vulnerabilidades operacionais não são executadas por erros de codificação na fase de implementação, mas ocorrem como resultado da implantação do software em um ambiente específico. Alguns fatores podem incluir uma vulnerabilidade operacional, como a configuração do software ou software e hardware com o qual ele interage, treinamento e conscientização do usuário, ambiente operacional físico.

De acordo com Clarke [21], é impossível desenvolver um sistema de software complexo com mil linhas de código livre de defeitos. Nesse sentido, Hoglund e McGraw [52] fizeram uma análise com o sistema operacional Windows XP quanto ao número de defeitos por mil

linhas de código. Eles afirmam que um produto de sistema de software pode conter até cinco defeitos a cada mil linhas de código, mesmo quando os desenvolvedores incluem estratégias de teste rigorosas. Além disso, levando em consideração o sistema operacional Windows XP, usando esta regra geral, ele pode conter potencialmente 40.000 defeitos de software em aproximadamente 40 milhões de linhas de código.

Prover segurança de software é uma tarefa difícil e necessária, e a complexidade de um produto de software e o número de linhas de código podem dificultar as atividades de teste. Em 2020, foram reportados ao *National Vulnerability Database (NVD)* mais de 18 mil vulnerabilidades. A Figura 2.2 apresenta o número de vulnerabilidades reportadas no NVD nos últimos seis anos. É possível perceber um crescimento considerável nas vulnerabilidades relatadas ao NVD nos últimos anos. Nesse sentido, muitos esforços têm sido aplicados à segurança de software. Com o objetivo de catalogar e classificar fragilidades e vulnerabilidades, a *MITER Corporation* [1] idealizou dois projetos: o *Common Vulnerabilities and Exposures (CVE)* e *Common Weakness Enumeration (CWE)*. O projeto CVE foi idealizado para compartilhar informações sobre as vulnerabilidades usando uma enumeração comum [4]. O projeto CWE corresponde a uma lista de fraquezas comuns de segurança de software. Este projeto é usado como uma linguagem padrão, uma medida para ferramentas de segurança de software e uma linha de base para a identificação de fraquezas, mitigação e esforços de prevenção [20].

Outro trabalho idealizado com o objetivo de evitar vulnerabilidades é o *Security Development Lifecycle (SDL)* proposto por Howard e Lipner [54]. O SDL corresponde a um processo de desenvolvimento de software, segundo eles, comprovadamente mais seguro. Eles acreditam que a análise do código fonte antes da compilação fornece um método altamente escalonável de revisão do código de segurança e garante que as políticas de codificação seguras sejam seguidas. No entanto, algumas fraquezas de codificação podem ser detectadas no ambiente do desenvolvedor, como a existência de funções inseguras ou outras funções proibidas e sua substituição por alternativas mais seguras. Além disso, eles sugerem a execução de algumas políticas de codificação seguras. Uma dessas políticas é: não usar funções proibidas. Nesse sentido, o SDL construiu uma lista com funções classificadas como banidas, de acordo com a experiência com *bugs* de segurança em programas reais, e foca quase

[1] https://www.mitre.org/

Figura 2.2: Número de Vulnerabilidades Confirmadas pelo NVD por Ano [5].

que exclusivamente em funções que podem levar a problemas relacionados a *buffer*. Cada função nesta lista deve ser substituída por uma versão mais segura [54].

Howard et al. [53] apresentaram os 24 pecados capitais relacionados à segurança de software e sugeriram como corrigi-los. Eles classificam os pecados em quatro categorias: aplicações web, implementação, criptografia e rede. Além disso, eles fornecem a descrição, as línguas afetadas, e os CWE que estão relacionados a cada pecado. A Tabela 2.1 apresenta os 24 pecados capitais e suas categorias.

2.2.1 Testes de Segurança

Teste de segurança é um tipo de teste peculiar em comparação com outros tipos de teste de software, pois há um adversário inteligente que também procura por fraquezas no código. Os agentes maliciosos estão constantemente testando fraquezas nos sistemas de software que podem ser usadas para explorá-los [69].

Pfleeger et al. [94] apontam que o trabalho do testador é explorar a interação de muitos fatores, como dependências, usuários imprevisíveis, efeitos colaterais e bases de implementação (linguagens, compiladores, infraestrutura). Todos esses fatores contribuem para a di-

Tabela 2.1: Os 24 pecados capitais apresentados por Howard et al. [53].

Categorias	Pecados
Aplicações Web	SQL Injection
	Web Server–Related Vulnerabilities
	Web Client–Related Vulnerabilities
	Use of Magic URLs, Predictable Cookies, and Hidden Form Fields
Implementação	Buffer Overruns
	Format String Problems
	Integer Overflows
	C++ Catastrophes
	Catching Exceptions
	Command Injection
	Failure to Handle Errors Correctly
	Information Leakage
	Race Conditions
	Poor Usability
	Not Updating Easily
	Executing Code with Too Much Privilege
	Failure to Protect Stored Data
	The Sins of Mobile Code
Criptografia	Use of Weak Password-Based Systems
	Weak Random Numbers
	Using Cryptography Incorrectly
Rede	Failing to Protect Network Traffic
	Improper Use of PKI, Especially SSL
	Trusting Network Name Resolution

ficuldade na tarefa de teste de software. Além disso, para execução de teste de segurança, precisamos nos preocupar com centenas de maneiras pelas quais o software pode apresentar comportamento vulnerável.

Graham et al. [48] definiram alguns princípios de teste de software, dos quais sete podem ser aplicados a testes de segurança de software. Clarke [21] destacou que os três primeiros princípios são de particular relevância:

i *Teste apresenta a presença de defeitos*. O teste pode mostrar que há defeitos, mas não pode provar que não há defeitos. Nesse sentido, o teste de segurança nunca pode provar a ausência de vulnerabilidades de segurança e pode apenas reduzir o número de defeitos não descobertos.

ii *Teste exaustivo é inviável*. Precisamos usar a análise de risco e as prioridades para nos concentrar nos esforços de teste.

iii *Teste o quanto antes possível*. Iniciar as atividades de teste no início do ciclo de vida de desenvolvimento do software.

Além dos princípios citados anteriormente, Graham et al. [48] continuam:

iv Geralmente, um pequeno número de módulos apresenta a maiorias dos defeitos descobertos.

v Muitas execuções do mesmo conjunto de teste não encontrarão novos defeitos. Os testes e dados de teste existentes podem precisar de alteração para detectar novos defeitos, e novos testes podem ser criados.

vi O teste é dependente do contexto, ou seja, o teste é feito de forma diferente em contextos diferentes. Por exemplo, um software que necessita de alto nível de segurança é testado de forma diferente de um aplicativo móvel de comércio eletrônico.

vii A ausência de defeitos é uma crença equivocada. Algumas organizações esperam que os testadores executem todos os testes possíveis e possam encontrar todos os defeitos possíveis. No entanto, como relatado nos princípios 1 e 2, essa é uma atividade impossível.

O teste funcional envolve determinar se o aplicativo satisfaz os requisitos contidos na especificação do software. Nos testes de segurança de software, os testadores estão preocupados em determinar a funcionalidade completa de uma aplicação, incluindo qualquer funcionalidade que pode ser executada por um ataque malicioso que causa algum comportamento inesperado e indesejado. Além disso, a funcionalidade completa de uma aplicação de software pode ser maior do que a especificada nos requisitos [21].

Não é possível determinar o teste de segurança mais adequado para todos os casos. De acordo com Laboon [69], o teste de segurança mais eficaz é aquele que é feito. Ele acredita que precisamos defender os testes de segurança que podem ser realizados e continuar sendo realizados. No entanto, algumas técnicas se destacam com relação ao suporte na detecção de fraquezas, as próximas seções destacam as técnicas de análise estática e *fuzz testing*.

2.3 Análise Estática

Como visto anteriormente, o teste de caixa branca é uma abordagem realizada quando o testador tem o conhecimento dos detalhes internos do sistema em teste, ou seja, normalmente significa que o testador tem acesso ao código fonte. Podemos dividir o teste da caixa branca em análise estrutural (estática) e teste estrutural (dinâmico) [21]. No que diz respeito à identificação de problemas de segurança, é possível utilizar uma variedade de técnicas de prevenção e detecção de vulnerabilidades ao longo do ciclo de desenvolvimento do software. Uma dessas técnicas, é a análise estática do código fonte, uma maneira escalonável para a revisão do código, que pode ser usada no início do ciclo de vida, não requer que o sistema seja executado e pode ser usada em partes do código [47].

Com relação ao teste de segurança, a análise estática significa identificar fraquezas, não do aplicativo executado, mas por meio da análise de código. Isso geralmente é obtido por correspondência de padrões. De forma geral, análise estática pode ser executada tanto no código fonte quanto no arquivo binário. Já no teste estrutural, o sistema em teste é testado em execução. O teste estrutural envolve ter conhecimentos dos detalhes de implementação, com o objetivo de detectar vulnerabilidades, no caso de testes de segurança [21].

Conforme a complexidade e tamanho dos softwares aumentaram, as ferramentas que executam análise estática automaticamente substituíram a audição manual do software. Neste

sentido, ferramentas como ITS4 [2] e RATS [3], surgiram com o objetivo de automatizar esta tarefa. De acordo com Chess e West [16], qualquer ferramenta que analisa o código sem executá-lo está realizando análise estática.

Problemas de segurança podem surgir por erros simples que um desenvolvedor pode cometer ou falta de compreensão sobre o que a programação segura envolve. É comum que os programadores não conheçam maneiras utilizadas por atacantes para tirar proveito de um trecho de código [16]. Neste sentido, Chess e West [16] listaram alguns motivos pelos quais a análise estática é adequada para identificar problemas de segurança. São eles:

- Ferramentas de análise estática verificam o código por completo de forma imparcial, sem dar mais atenção a partes de código consideradas mais "interessantes" ou fáceis, como um programador pode fazer.
- Ferramentas de análise estática podem apontar para a causa raiz de um problema, não apenas para um dos seus sintomas. Essa característica é importante para garantir que vulnerabilidades sejam corrigidas de forma adequada.
- Por meio da análise estática é possível detectar problemas durante as primeiras fases do processo de desenvolvimento, antes mesmo do programa ser executado pela primeira vez. Identificar problemas antecipadamente além de reduzir o custo de correção, ajuda a nortear o programador a corrigir problemas pelo rápido feedback.
- A análise estática nos permite reavaliar um grande corpo de código em busca de novas formas de ataques, não consideradas anteriormente.

O SANS Institute corresponde a uma fonte de treinamento em segurança da informação e certificação de segurança [4]. Eles mantêm uma lista dos 25 erros de software mais perigosos. Thien [68], do SANS Institute, concluiu que a ferramenta de análise estática *Flawfinder*, é uma excelente ferramenta de varredura de código fonte que os desenvolvedores podem executar para encontrar os problemas de segurança mais comuns com programas C. Além disso, é oficialmente compatível com CWE e recebeu o selo de "aprovação" no *CII Best Practices*.

[2] https://seclab.cs.ucdavis.edu/projects/testing/tools/its4.html
[3] https://security.web.cern.ch/recommendations/en/codetools/rats.shtml
[4] https://www.sans.org/

Flawfinder funciona fazendo tokenização léxica simples e procurando correspondências de tokens em sua base de dados. Além disso, examina os parâmetros das funções com o objetivo de estimar o risco. *Flawfinder* funciona comparando o código fonte com um banco de dados embutido de funções com problemas conhecidos, como estouro de buffer, problemas com formato de *strings* e condições de corrida. Em seguida, ele produz uma lista de *"hits"* (possíveis vulnerabilidades), classificados por nível de risco. Os *hits* considerados mais perigosos são apresentados primeiro. Para classificar o nível de risco, *Flawfinder* considera a função analisada e seus parâmetros. Por exemplo, *strings* constantes são consideradas menos arriscadas do que *strings* totalmente variáveis em muitos contextos [35].

Flawfinder pode ser uma ajuda bastante útil para indicar pistas de vulnerabilidades. No entanto, ele faz basicamente correspondência de padrões de texto simples, não entende a semântica do código. Neste sentido, nem todo acerto é na verdade uma vulnerabilidade de segurança, e nem toda vulnerabilidade de segurança é necessariamente encontrada [35]. De maneira geral, os problemas da análise estática são considerados indecidíveis [16], ou seja, não é possível construir um algoritmo que sempre apresente uma resposta correta para todos os casos. Neste sentido, utilizando ferramentas de análise estática, não é possível detectar todas as vulnerabilidades no código (falsos negativos). Além disso, elas são propensas a reportar casos que não correspondem a vulnerabilidades de segurança (falsos positivos) [47].

De acordo com Wheeler [35], a ferramenta de análise estática Flawfinder é oficialmente compatível com o CWE. As descrições de ocorrências com `MS-banned` indicam funções que estão na lista de funções proibidas lançada pela Microsoft. Mesmo assim, de acordo com Godefroid et al. [45], na Microsoft, *fuzzing* é obrigatório para todas as interfaces não confiáveis de todos os produtos, conforme apresentado no *Security Development Lifecycle*.

2.4 Fuzz Testing

Fuzzing ou *fuzz testing* se tornou uma das técnicas de teste mais eficazes em relação à identificação de problemas de segurança em sistemas de software. O *fuzz testing* visa gerar entradas inválidas, aleatórias e inesperadas, especialmente entradas válidas, que podem desencadear comportamentos errôneos ou de falha, incluindo a exploração de vulnerabilidades [112].

Segundo Juranic´ [58], *fuzzing* é baseado na técnica de injeção de falhas. Essa abordagem

envia muitos dados de entrada para o software em teste com o objetivo de detectar vulnerabilidades. *Fuzz testing* pode detectar muitos tipos de vulnerabilidades, como:

- Buffer overflows
- Integer overflows
- Format string vulnerabilities
- Race condition vulnerabilities
- SQL injection
- Cross Site Scripting (XSS)
- Remote command execution
- Filesystem attacks (reverse traversal, etc.)
- Information leaking vulnerabilities.

Em geral, o *fuzzing* é executado gerando entradas aleatórias e executando o programa com as entradas geradas com o objetivo de executar uma ramificação específica do programa tanto quanto possível, na esperança de travar o programa [112]. Sutton et al. [101] definiram o *fuzzing* como um método para descobrir defeitos no software, fornecendo entradas inesperadas e monitorando exceções. Geralmente, é um processo automatizado ou semiautomático que envolve a manipulação repetida e o fornecimento de dados ao software de destino para processamento. *Fuzzers* são classificados em duas categorias: baseado em mutação ou baseado em geração. O *fuzzer* baseado em mutação executa mutações em amostras de dados existentes para criar dados de teste, e o baseado em geração cria dados de teste modelando o protocolo de destino do formato de arquivo.

Além disso, Clarke [21] afirma que o termo *fuzzer* é usado para rotular muitas ferramentas, *scripts*, aplicativos e frameworks. No entanto, *fuzzers* apresentam um conjunto de recursos padrão, onde os dois últimos recursos não são obrigatórios ou podem ser implementados externamente ao fuzzer. São eles:

- **Geração:** Geração de dados a serem enviados ao programa em teste;

- **Transmissão:** Enviando os dados gerados ao programa;
- **Monitoramento e registro:** Observar e registrar a reação da execução do programa com os dados enviados;
- **Automação:** Reduzir o número de interações diretas do usuário para realizar o teste.

De acordo com Sutton et al. [101], *fuzzing* apresenta seis fases básicas, são elas:

- **Identificação do alvo:** nesta fase, identificamos o alvo para execução do *fuzzing*. Esta fase é considerada crítica para a definição das ferramentas ou técnicas que serão utilizadas.
- **Identificação de pontos de entrada de dados:** identificação dos vetores de entrada com o objetivo de expor a vulnerabilidade. Identificar fontes potenciais de entrada ou as entradas esperadas é muito importante para o teste.
- **Geração de dados**: uma vez que os vetores de entrada tenham sido identificados, os dados *fuzzing* devem ser gerados. A geração de dados pode ser usada com valores predeterminados, alterar os dados existentes ou gerar dados dinamicamente. Essa decisão depende do destino e do formato dos dados.
- **Execução**: nesta fase, o programa é executado com os dados gerados nas fases anteriores. A execução consiste em enviar um pacote de dados ao destino.
- **Monitoramento**: nesta fase, são identificados os resultados da execução. Quando enviamos muitos pacotes de dados, fazendo com que o alvo trave, é um esforço inútil se não pudermos identificar o pacote responsável pelo travamento. O processo de monitoramento de exceção ou falha é uma etapa vital, porém muitas vezes esquecida.
- **Determinar a explorabilidade**: é a fase em que é determinado se o defeito descoberto pode ser explorado. É um processo tipicamente manual e pode ser executado por outra pessoa que não seja a pessoa que está realizando *fuzzing*.

Fuzzing é uma das estratégias mais poderosas na identificação problemas de segurança em software do mundo real. De acordo com Clark [21], a grande maioria dos defeitos que

são revelados por *fuzzing* são atribuíveis à fase de implementação. No entanto, os erros que ocorrem na fase de projeto ou operação também podem ser detectados utilizando esta técnica.

Segundo Li e Paxson [71], os desenvolvedores podem melhorar seu processo de teste utilizando ferramentas desenvolvidas com esse objetivo de forma mais extensa, como o *fuzzer American fuzzy lop (AFL)* [2], que trata de fornecer suporte na identificação de entradas que desencadeiam problemas potencialmente exploráveis. Apesar da existência de diversas ferramentas capazes de executar *fuzzing* que permitem ao usuário verificar a segurança do software e o alto número de vulnerabilidades de segurança detectadas utilizando esta técnica, não corresponde a uma abordagem universal para realizar a detecção de vulnerabilidades de segurança [58].

Com o objetivo de melhorar a cobertura funcional do código em teste, AFL é uma ferramenta de força bruta e orientado para a segurança que une instrumentação de tempo de compilação e algoritmos genéticos, com a intenção de descobrir automaticamente casos de teste que acionam novos estados internos no alvo binário [2]. Foi utilizado na detecção de vulnerabilidades e bugs em grandes projetos, como Mozilla Firefox, OpenSSL e Linux. De acordo com Wang e Kang [112], AFL é uma das ferramentas de teste *fuzzing* mais avançadas. A maior parte de sua popularidade pode estar relacionada à facilidade de uso e configuração.

2.5 Considerações Finais

Neste capítulo, destacamos os conceitos essenciais considerados relevantes para o melhor entendimento deste trabalho de tese. Inicialmente apresentamos os principais conceitos relacionados a teste de software, segurança de sistemas, análise estática e dinâmica, teste de segurança e *fuzz testing*. O próximo capítulo apresenta uma técnica para detecção de fraquezas no contexto de programas C, utilizando análise estática e *fuzz testing*.

Capítulo 3

Uma Técnica para Detectar Fraquezas em Programas C

Este capítulo tem como objetivo descrever uma técnica para detecção de fraquezas no contexto de programas C, utilizando análise estática e *fuzz testing*. Nossa técnica apresenta uma abordagem a nível de unidade para detecção de fraquezas. Estamos interessados em fornecer suporte aos desenvolvedores na detecção de fraquezas introduzidas durante os estágios iniciais de desenvolvimento de programas C para que os desenvolvedores possam verificar a presença de fraquezas em uma função ou procedimento individual que está sendo desenvolvido. A identificação e correção de problemas de segurança é mais barata quando executada durante os estágios iniciais do processo de desenvolvimento de software [25].

Neste capítulo, apresentamos uma visão geral sobre a técnica para detecção de fraquezas em código C (Seção 3.1), onde descrevemos a técnica de forma conceitual. Em seguida, detalhamos a implementação de cada fase da técnica proposta em um *framework*, incluindo suas respectivas entradas e saídas. Finalmente, apresentamos as considerações finais do capítulo (Seção 3.3).

3.1 Visão Geral

Durante o desenvolvimento desta pesquisa, estudamos como detectar fraquezas em código fonte de sistemas de softwares desenvolvidos em linguagem C, por meio de estratégias de verificação e validação. Neste sentido, elaboramos uma técnica onde avaliamos a presença

de fraquezas no código fonte de programas C, utilizando análise estática e *fuzz testing*. A Figura 3.1 apresenta um fluxograma com as etapas da técnica proposta.

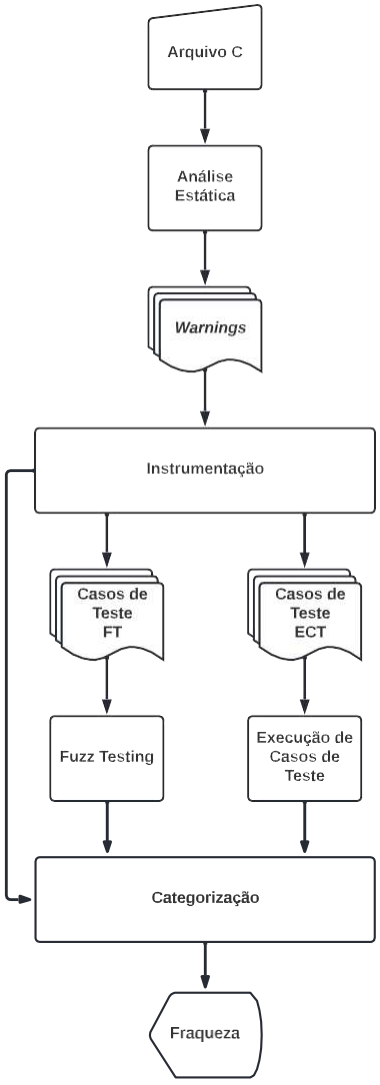

Figura 3.1: Modelo conceitual de uma técnica para detectar fraquezas em programas C.

A técnica recebe como entrada um arquivo único de código fonte C. Antes de avaliar se um determinado trecho de código corresponde ou não a uma fraqueza, consideramos importante a utilização de uma estratégia automática, capaz de analisar muitas linhas de código e indicar trechos de código suspeitos, direcionando a técnica para partes específicas do programa. Neste sentido, executamos `análise estática` no arquivo selecionado com o objetivo de identificar partes do programa consideradas suspeitas. Nesta etapa, é gerada uma lista de avisos *(warnings)* que serão avaliados nas etapas posteriores.

Em seguida, selecionamos um dos *warnings* da lista gerada e damos início a etapa de `Instrumentação`. Nesta etapa, isolamos as funções que apresentam trechos de código considerados suspeitos e geramos dois tipos de casos de teste, para serem executados nas etapas de *fuzz testing* (FT) e Execução de casos de teste (ECT). Nesta etapa, comentamos as demais partes do código com o objetivo de isolar a função em teste. Os casos de teste são gerados de acordo com o tipo de fraqueza encontrada no código. Neste sentido, foram elaborados guias com o objetivo de auxiliar o processo de criação dos casos de teste indicando os elementos necessários aos casos de teste. Os guias são discutidos em detalhes na Seção 3.2.2.

Ainda na etapa de `Instrumentação`, avaliamos se é possível elaborar os casos de teste, considerando problemas relacionados a testabilidade do código em teste. De acordo com Ammann e Offutt [9], testabilidade corresponde ao nível em que um sistema facilita o estabelecimento de critérios de teste e o desempenho de testes para determinar se esses critérios foram atendidos. Desta forma, caso não seja possível gerar os casos de tese, o processo segue para a etapa de categorização, onde classificamos o *warning* como não testado.

Caso seja possível criar os casos de teste, executamos `fuzz testing` com o objetivo de verificar se a fraqueza analisada é capaz de causar *crash* ao código testado com as entradas geradas pelo *fuzzer*. Além disso, o caso de teste utilizado nesta etapa possui a responsabilidade de armazenar a lista de entradas geradas para ser utilizada na etapa de execução de casos de teste. Em seguida, na etapa `Execução de caos de teste`, executamos os casos de teste com as entradas geradas pelo *fuzzer* e com valores definidos manualmente pelos desenvolvedores para verificar comportamentos inesperados causados pela fraqueza. Finalmente, na etapa de `Categorização`, categorizamos o conjunto de fraquezas identificadas e geramos um relatório com uma lista de entradas e os resultados de suas execuções: *fail*, *pass* ou *crash*. Além disso, como resultado, apresentamos se um código suspeito pode ser classificado como uma fraqueza com base nos resultados obtidos nas etapas *fuzz testing* e Execução de Casos de Teste onde, *fail* e *crash*, indica que uma fraqueza foi detectada e *pass* indica que uma fraqueza não foi detectada.

3.2 Implementação

A técnica apresentada na seção anterior foi implementada em um *framework* chamado *Weaknesses TesTing* (WTT), que corresponde a um conjunto específico de ferramentas, scripts, guias para construção de casos de teste e diretrizes de execução. Nesta seção apresentamos os detalhes da implementação do *framework*. A Figura 3.2 apresenta uma visão geral das principais fases da nossa técnica. Apresentamos os detalhes de cada fase nas subseções a seguir.

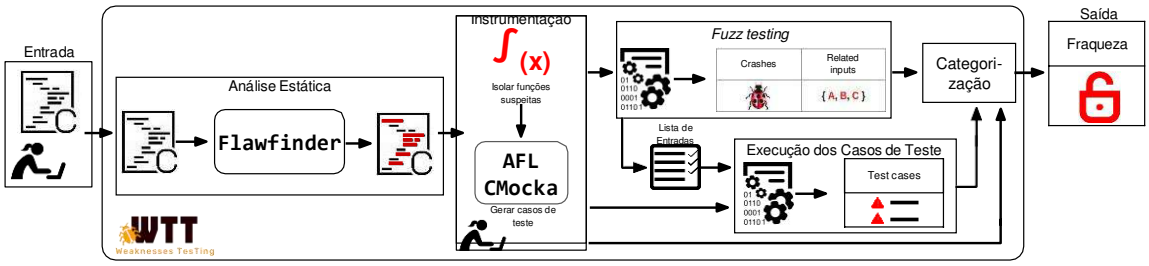

Figura 3.2: Uma Técnica para Detectar Fraquezas em Código C.

Para melhor entendimento da técnica proposta, utilizaremos o Código Fonte 3.1, selecionado do conjunto de dados *Juliet Test Suite* [92], para apresentar em detalhes cada fase da técnica. As linhas 6 a 11 apresentam uma verificação da leitura dos dados de entrada pela função *fgets()*. Na linha 13 é armazenado o resultado da soma de *data*, recebido como dado de entrada, com o valor 1 na variável *result*. Em seguida, na linha 14 é impresso o resultado da operação de adição.

Código Fonte 3.1: Exemplo de Código para Execução da Técnica Proposta.

```
1  void Integer_Overflow(){
2      int data;
3      data = 0;
4      char inputBuffer[CHAR_ARRAY_SIZE] = "";
5      /* POTENTIAL FLAW: Read data from the console using fgets() */
6      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL){
7          /* Convert to int */
8          data = atoi(inputBuffer);
9      }else{
10         printf("fgets() failed.");
```

```
11      }
12      /* POTENTIAL FLAW: Adding 1 to data could cause an overflow */
13      int result = data + 1;
14  printf(result);15  }
```

3.2.1 Análise Estática

A análise estática é um processo fundamental na avaliação da qualidade do código. Ferramentas de análise estática apresentam o potencial de apoiar desenvolvedores a encontrar e corrigir vulnerabilidades durante os primeiros estágios do processo de desenvolvimento de software [107; 102].

O objetivo principal desta fase é identificar trechos de código suspeitos em arquivos C por meio de ferramentas de análise estática, como `Flawfinder`[1]. Nesta fase, WTT recebe como entrada um arquivo de código fonte C e apresenta como resultado uma lista de possíveis fraquezas (*warnings*) no código fonte analisado.

Para execução da fase de análise estática WTT utiliza `Flawfinder`, uma excelente ferramenta de revisão de código, oficialmente compatível com o projeto CWE, que recebeu o selo *"passing"* das Melhores Práticas CII [35]. A ferramenta produz um relatório que indica todos os códigos suspeitos detectados apontando para sua localização no código, os classifica por tipo de fraqueza e atribui um nível de risco. `Flawfinder` rotula os pontos fracos identificados com o nível de risco que varia de 0, para baixo risco, até 5, para maior risco.

A Figura 3.3 apresenta o relatório simplificado da execução do `Flawfinder` no Código Fonte 3.1. Como é possível verificar, a ferramenta reportou dois *warnings* de dois tipos diferentes. Além disso, indicou o nome arquivo avaliado, as linhas de cada fraqueza (linhas 4 e 8), o nível de risco (nível de risco 2), uma classificação relacionada ao tipo da fraqueza e uma breve descrição.

Como a verificação de todos os *warnings* da lista gerada pela ferramenta pode não ser viável para programas reais com muitas linhas de código, recomendamos que os desenvolvedores selecionem os *warnings* que apresentem maior risco de acordo com os recursos

[1] https://www.dwheeler.com/flawfinder/

```
CWE190.c:4:  [2] (buffer) char: Statically-sized arrays can be improperly
        restricted, leading to potential overflows or other issues (CWE-119!/
        CWE-120). Perform bounds checking, use functions that limit length, or
    ensure that the size is larger than the maximum possible length.
CWE190.c:8:  [2] (integer) atoi: Unless checked, the resulting number can
    exceed the expected range (CWE-190). If source untrusted, check both
    minimum and maximum, even if the input had no minus sign (large
    numbers can roll over into negative number; consider saving to an
    unsigned value if that is intended).
```

Figura 3.3: Relatório da Execução do Flawfinder no Código Fonte 3.1

disponíveis para a execução das próximas fases do WTT.

3.2.2 Instrumentação

A segunda fase da técnica proposta corresponde à instrumentação do código. O objetivo principal desta fase, é criar casos de teste adequados para avaliar a presença da fraqueza no código selecionado. Para este fim, esta fase recebe como entrada a lista de *warnings* gerados na fase anterior. Primeiramente, selecionamos um dos *warnings* listados baseado em critérios que podem ser nível de risco, tipo de fraqueza ou probabilidade de exploração. Em seguida, isolamos a função que contém o código suspeito. Finalmente, criamos casos de testes adequados para detecção do tipo de fraqueza selecionada de acordo com guias de construção de casos de teste. Atualmente, o processo de criação dos casos de teste é feito de forma manual. No entanto, temos indicativos de automatização por meio de *templates*, onde seria possível criar casos de teste substituindo algumas linhas de código, como será apresentado no Capítulo 5. Nesta fase, utilizamos o framework CMocka na criação dos casos de teste. CMocka é um framework de teste de unidade para código fonte C que apresenta suporte a objetos *mock* [3].

Os projetos CWE e OWASP apresentam a descrição de cada fraqueza abordada neste trabalho, bem como modos de introdução, plataformas aplicáveis, consequências comuns, probabilidade de exploração e alguns exemplos. Neste sentido, de acordo com as características de cada fraqueza apresentadas pelos projetos citados, foram criados guias para construção

de casos de teste, onde são definidos os elementos que cada caso de teste deve apresentar para expor as fraquezas objeto deste estudo. Nos guias propostos são definidos elementos, como: resumo do caso de teste, pré-condições, entradas, ações, resultados esperados e pós-condições.

Executamos um estudo inicial onde consideramos alguns critérios com o objetivo de definir quais tipos de fraquezas seriam abordadas pelo WTT. Os seguintes critérios foram considerados neste estudo inicial:

- Selecionamos tipos de fraquezas inseridas durante a fase de implementação, já que estamos considerando o escopo de unidade;
- Fraquezas que apresentaram média ou alta probabilidade de exploração, de acordo com o projeto CWE; e
- Fraquezas que as ferramentas de análise estática fossem capazes de identificar.

Além disso, quando este estudo foi executado, buffer overflow, integer overflow e format string estavam na lista das 25 fraquezas mais perigosas [19]. Neste estudo, de acordo com este estudo inicial, identificamos que seria possível detectar os seguintes tipos de fraquezas: 1) Buffer Copy without Checking Size of Input - *Buffer Overflow* (CWE-120[2]); 2) Use of Externally-Controlled Format String - *Format String* (CWE-134[3]); 3) *Integer Overflow* or Wraparound (CWE-190[4]);

A Tabela 3.1 apresenta o guia proposto para elaboração de casos de teste com o objetivo de detectar fraquezas do tipo *buffer overflow*. De acordo com Ammann e Offutt [9], entradas inválidas geralmente fazem com que o software se comporte de maneira inesperada, que atacantes mal-intencionados podem utilizar a seu favor. É assim que funciona o clássico *"buffer overflow attack"*. A etapa chave em um ataque de *buffer overflow* é fornecer uma entrada que seja muito longa para caber no *buffer* disponível. Desta forma, o objetivo principal dos casos de teste gerados seguindo o guia proposto, é verificar se o trecho de código que está sendo avaliado permite alocação além da capacidade de armazenamento da variável destino. Como pré-condições para elaboração dos casos de teste, observamos se o código suspeito

[2]https://cwe.mitre.org/data/definitions/120.html
[3]https://cwe.mitre.org/data/definitions/134.html
[4]https://cwe.mitre.org/data/definitions/190.html

apresenta funções que são propícias a ocasionar a fraqueza, como *strcpy, strcat, strcmp* etc. Como entrada, temos: instância de variável maior que a capacidade do buffer alocado como destino. Finalmente, como resultados esperados temos *crash*, ocasionado pelo programa tentas acessar um endereço de memória ocupado ou inexistente, ou o programa não permite a cópia para o *buffer* destino.

Tabela 3.1: Guia para criação de casos de teste com o objetivo de detectar *buffer overflow*.

Seção	Descrição
Resumo	- Verificar se a implementação, ao receber uma variável maior que a capacidade do buffer alocado como destino, permite a realização da cópia.
Pré-condições	- Código suspeito apresenta uma das funções banidas que podem ocasionar o estouro do buffer, como strcpy. - Definir variável que supere o tamanho da variável para a qual seu conteúdo será copiado.
Entradas	- Instância de variável maior que a capacidade do buffer alocado como destino.
Ações	- Implementar funções *mock* necessárias à execução. - Instanciar variável. - Enviar a variável instanciada para a função. - assert_true(1).
Resultados esperados	- Crash. - A implementação permite a cópia além do limite alocado. O teste falha. - A implementação não permite a cópia. O teste passa.
Pós-condições	- Crash no sistema. - Cópia realizada com sucesso.

O Código Fonte 3.2 apresenta uma versão simplificada de um exemplo de fraqueza do tipo *buffer overflow*. As linhas 6 a 10 apresentam uma verificação da leitura dos dados de

entrada pela função `fgets()`. O dado recebido como entrada, é definido como índice da variável *buffer*, definida na linha 12. O *overflow* ocorre na linha 15, ao tenta gravar em um índice do *array* que está acima do limite superior.

Código Fonte 3.2: Versão simplificada de código fonte com fraqueza do tipo *buffer overflow*.

```
1  CWE121_Stack_Based_Buffer_Overflow(){
2      int data;
3      data = -1;
4      char inputBuffer[CHAR_ARRAY_SIZE] = "";
5      /* Read data from the console using fgets() */
6      if(fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL){
7          data = atoi(inputBuffer);
8      }else{
9      printf("fgets()failed.");10}
11     int i;
12     int buffer[10] = { 0 };
13     /* Write to an index of the array that is above the upper bound */
14     if(data >= 0){
15         buffer[data] = 1;
16         for(i = 0; i < 10; i++){
17             printf(buffer[i]);
18         }
19     }else{
20     printf("ERROR: Array index is negative.");21   }
22 }
```

O Código Fonte 3.3 apresenta uma versão simplificada do caso de teste criado com o objetivo de detectar *buffer overflow*, baseado no guia apresentado anteriormente. As linhas 1 a 5 apresentam funções e estruturas que são utilizadas na função que está sendo testada, porém definidas fora do escopo da função. As linhas 6 a 15 apresentam a função de teste, onde declaramos uma variável `data` e atribuímos um valor recebido pela função de entrada de dados *scanf* (linha 7). Em seguida, enviamos a variável para a função em teste, onde a cópia ao *buffer* destino é realizada.

Código Fonte 3.3: Versão simplificada do caso de teste criado para testar *buffer overflow*.

```
1  char inputBuffer[CHAR_ARRAY_SIZE] = "";
2  char __wrap_fgets(){
3      strcpy(__s, inputBuffer);
4  return__s;5 }
6  static void test_juliet_ft(){
7      (void)state; //unused variable
8      int data;
9      data = 0;
10     scanf("%d\n", &data);
11     sprintf(inputBuffer, "%d", data);
12     // Lines of code
13     CWE121_Stack_Based_Buffer_Overflow();
14 assert_true(1);15 }
```

Na Tabela 3.2 são apresentados os elementos que compõem o guia proposto para a criação de casos de teste com o objetivo de detectar código fonte que podem causar *integer overflow*. Os casos de teste criados seguindo este guia tem como objetivo verificar se o código, ao receber um valor inesperado, atingindo os limites da capacidade de alocação do resultado do cálculo, retorna resultados inconsistentes. Pré-condições para este tipo de casos de teste é o código suspeito efetuar cálculo e armazenar o resultado. O problema acontece quando o valor numérico resultante estiver fora do intervalo que pode ser representado com um determinado número de dígitos. Entradas para este tipo de caso de teste podem ser valores aleatórios ou valores que atinjam os limites de alocação. Os resultados esperados são valores que condizem com o resultado real do cálculo.

Tabela 3.2: Guia para criação de casos de teste com o objetivo de detectar *integer overflow*.

Seção	Descrição
Resumo	- Verificar se a implementação, ao receber um valor inesperado, atingindo os limites da capacidade de alocação do resultado do cálculo, retorna resultados inconsistentes. Retornar um valor negativo ou zero são possíveis indícios da ocorrência de um integer overflow.
Pré-condições	- Código suspeito efetua cálculos e armazena o resultado.
Entradas	- Valores aleatórios; - Valor que verifique os limites da capacidade de alocação do resultado do cálculo;
Ações	- Implementar funções *mock* necessárias à execução. - Instanciar variável. - Enviar a variável instanciada para a função. - assert_int_equal(valor esperado, valor retornado).
Resultados esperados	- Valor diferente do resultado calculado na função testada. Teste Falha. - Valor igual ao resultado calculado na função testada. Teste Passa.
Pós-condições	- Comportamento inesperado na execução. - Operação efetuada com sucesso.

O Código Fonte 3.4 apresenta uma versão simplificada do caso de teste baseado no guia apresentado na Tabela 3.2, que apresenta o objetivo de detectar o *integer overflow* apontado no código fonte 3.1. A linha 1 apresenta uma função *mock* da função `fgets`, utilizada na função que está sendo testada. As linhas 2 a 10 apresentam a função de teste, onde declaramos uma variável `data` e atribuímos um valor recebido pela função de entrada de dados *scanf* (linhas 3, 4 e 5). A função em teste recebe o valor enviado, efetua a operação de soma com o valor "1" e retorna o resultado. Desta forma, efetuamos a comparação da

variável `result` (data + 1), definida na linha 6, com o resultado retornado da função em teste, armazenado na variável `buf`.

Código Fonte 3.4: Versão simplificada do caso de teste criado para testar *integer overflow*

```
1 char __wrap_fgets(...){return __s;}
2 static void test_juliet_rtc(...){
3     int data;
4     data = 0;
5     scanf("%d\n", &data);
6     int result = data + 1;
7     /*Lines of code here*/
8     Integer_Overflow();
9 assert_true(atoi(buf)== result);10
  }
```

A Tabela 3.3 apresenta o guia proposto para elaboração de casos de teste com o objetivo de detectar fraquezas classificadas como *format string*. Este tipo de caso de teste tem como objetivo verificar se a formatação da string pode ser influenciada por um agente externo. A pré-condição definida para este tipo de caso de teste é o programa receber a formatação da string de uma fonte externa. Como resultados esperados, temos a comparação da string enviada para a função para ser impressa com a string que foi impressa. Se as strings forem iguais o teste passa, caso contrário, o teste falha.

Durante a fase de análise estática, foi indicado que o Código Fonte 1.1 apresenta uma fraqueza do tipo CWE-134: Use of Externally-Controlled Format String. Para este caso, criamos um caso de teste com o objetivo de detectar a fraqueza, baseado no guia apresentado na Tabela 3.3. O Código Fonte 3.5 apresenta uma versão simplificada do caso de teste criado com o objetivo de detectar a fraqueza no Código Fonte 1.1.

Tabela 3.3: Guia para criação de casos de teste com o objetivo de detectar *format string*.

Seção	**Descrição**
Resumo	- Verificar se a implementação, ao receber uma string com a formatação de uma fonte externa, permite que a string seja impressa.
Pré-condições	- Função que recebe a formatação de string de fonte externa.
Entradas	- String com formatação: %xexemplo.
Ações	- Implementar funções *mock* necessárias à execução. - Instanciar string com a formatação. - Enviar string para a função testada. - assert_string_equal(string esperada, string impressa).
Resultados esperados	- String impressa diferente da string enviada à função testada. Teste Falha. - String impressa igual a string enviada à função testada. Teste Passa.
Pós-condições	- Acesso indevido a valor hexadecimal referente a um espaço de memória. - Impressão da string enviada a função.

Código Fonte 3.5: Versão simplificada do arquivo `Tester_htdbm.c` criado para testar o arquivo `htdbm.c` apresentado em sua versão simplificada no Código Fonte 1.1.

```
1  //Mocked functions
2  int __wrap_apr_dbm_firstkey(...){}
3  int __wrap_apr_dbm_fetch (...){}
4  int __wrap_apr_dbm_nextkey (...){}
5
6  typedef struct{...} htdbm_t_teste;
7  // Test functions
8  static void test_htdbm_list(void** state) {
9    char *expStr = "teste:%xex";
10   char retStr[BUFSIZ];
11   char buf[BUFSIZ];
12   freopen("/dev/null", "a", stderr);
13   setbuf(stderr, buf);
14   htdbm_list(expStr);
15   freopen ("/dev/tty", "a", stderr);
16   assert_string_equal(expStr, retStr);
17 }
18 int setup (void ** state){}
19 int teardown (void ** state){}
20
21 int main(int argc, char** argv) {
22     const struct CMUnitTest tests[] = {
23         cmocka_unit_test(test_htdbm_list)
24     };
25     int count_fail_tests =
26      cmocka_run_group_tests (tests, setup, teardown);
27     return count_fail_tests;
28 }
```

As linhas 1 a 6 apresentam funções e estruturas que são chamadas na função que está sendo testada, porém definidas fora do escopo da função. As linhas 8 a 17 apresentam a função de teste, onde comparamos a cadeia de caracteres passada como entrada com a cadeia de caracteres que foi impressa pela função `htdbm_list`. Se as cadeias de caracteres comparadas são iguais, o teste passa. Neste exemplo, passamos como entrada `teste:%xex`, já

que a função `htdbm_list` imprime o conteúdo à direita do caractere ":" (dois pontos) e, sabíamos que se a função imprimisse o valor "%x", seria impresso um hexadecimal que corresponde ao endereço de memória, por exemplo, em vez de imprimir "%xex", a função iria imprimir "fbad3484ex". As linhas 18 a 28 apresentam *setup*, *teardown* e a função principal com parâmetros da estrutura do framework CMocka.

Além dos guias apresentados anteriormente, elaboramos guias para construção de casos de teste para os tipos *CWE-191: Integer Underflow (Wrap or Wraparound)* e *CWE-369: Divide By Zero*. Estes tipos de fraquezas foram abordados no estudo apresentado por Mouzarani e Sadeghiyan [82].

Na Tabela 3.4 são apresentados os elementos que compõem o guia proposto para a criação de casos de teste com o objetivo de detectar código fonte que podem causar *integer underflow*. Casos de teste criados seguindo este guia tem como objetivo verificar se o código efetua cálculos de modo que o resultado seja menor que o valor mínimo permitido, ocasionando um valor diferente do resultado esperado. Pré-condições para este tipo de casos de teste é o código suspeito efetuar cálculo e armazenar o resultado. O problema acontece quando o valor numérico resultante de uma operação estiver fora do limite inferior que pode ser representado. Possíveis entradas para este tipo de caso de teste podem ser valores aleatórios ou que atinjam o limite inferior de alocação. Os resultados esperados são valores que condizem com o resultado real do cálculo.

O Código Fonte 3.6 apresenta um exemplo de fraqueza do tipo *integer underflow*. As linhas 5 a 9 apresentam uma verificação da leitura dos dados de entrada pela função `fgets()`. Neste caso, a função em teste efetua uma multiplicação do valor recebido por 2 e retorna o resultado da multiplicação. Desta forma, se a função receber um valor negativo, que o resultado de sua multiplicação por 2 for menor que o valor mínimo possível de ser armazenado, causará o *underflow*.

Tabela 3.4: Guia para criação de casos de teste com o objetivo de detectar *integer underflow*.

Seção	**Descrição**
Resumo	- Verificar se a implementação, ao receber um valor inesperado, atingindo o limite inferior da capacidade de alocação do resultado do cálculo, retorna resultados inconsistentes.
Pré-condições	- Código suspeito efetua cálculos e armazena o resultado.
Entradas	- Valores aleatórios; - Valor que verifique o limite inferior da capacidade de alocação do resultado do cálculo;
Ações	- Implementar funções *mock* necessárias à execução. - Instanciar variável. - Enviar a variável instanciada para a função. - assert_equal(valor esperado, valor retornado).
Resultados esperados	- Valor diferente do resultado calculado na função testada. Teste Falha. - Valor igual ao resultado calculado na função testada. Teste Passa.
Pós-condições	- Comportamento inesperado na execução. - Operação efetuada com sucesso.

Código Fonte 3.6: Exemplo de código que apresenta fraqueza do tipo *integer underflow*

```
1  void CWE191_Integer_Underflow__int_fgets(){
2      int data;
3      data = 0;
4      char inputBuffer[CHAR_ARRAY_SIZE] = "";
5      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL){
6          data = atoi(inputBuffer);
7      }else{
8      printf("fgets() failed.");9}
10     if(data < 0){ /* ensure we won't have an overflow */
11         /* if (data * 2) < INT_MIN, this will underflow */
12         int result = data * 2;
13     printf(result);14
       }
15 }
```

O Código Fonte 3.7 apresenta uma versão simplificada do caso de teste criado com o objetivo de detectar *integer underflow*, baseado no guia apresentado na Tabela 3.4. A linha 1 apresenta uma função *mock* da função `fgets`. As linhas 2 a 10 apresentam a função de teste, onde declaramos uma variável `data` e atribuímos um valor recebido pela função de entrada de dados *scanf* (linhas 3, 4 e 5). Para verificar a ocorrência do buffer underflow, efetuamos a comparação da variável `result` (data * 2), definida na linha 6, com o resultado retornado da função em teste, armazenado na variável `buf`.

Código Fonte 3.7: Versão simplificada do caso de teste criado para testar *integer underflow*

```
1  char __wrap_fgets(...){return __s;}
2  static void test_juliet_rtc(...){
3      int data;
4      data = 0;
5      scanf("%d\n", &data);
6      int result = data * 2;
7      /*Lines of code here*/
8      CWE191_Integer_Underflow__int_fgets();
9  assert_true(atoi(buf)== result);10
   }
```

A Tabela 3.5 apresenta o guia proposto para elaboração de casos de teste com o objetivo de detectar fraquezas do tipo *CWE-369: Divide By Zero*. Casos de teste elaboradores seguindo este guia tem como objetivo verificar se a implementação, ao receber um valor inesperado, efetua uma divisão por zero. Pré-condições para este tipo de casos de teste é a implementação efetuar cálculos que podem ser influenciados por entradas fornecidas. As entradas para este tipo de caso de teste podem ser valores aleatórios ou direcionados à divisão por zero. Finalmente, os resultados esperados seriam: (i) O código não permite efetuar a divisão por zero. O teste passa. (ii) O código permite a divisão por zero. *Crash* no sistema.

Tabela 3.5: Guia para criação de casos de teste com o objetivo de detectar *divide by zero*.

Seção	**Descrição**
Resumo	- Verificar se a implementação, ao receber um valor inesperado, efetua uma divisão por zero.
Pré-condições	- O código efetua cálculos que podem ser influenciados por entradas fornecidas.
Entradas	- Valores aleatórios ou que que direcione o código à divisão por zero.
Ações	- Implementar funções *mock* necessárias à execução. - Instanciar variável. - Enviar a variável instanciada para a função. - assert_equal(valor esperado, valor retornado).
Resultados esperados	- O código não permite efetuar a divisão por zero. Teste passa. - *Crash* decorrente de uma divisão por zero.
Pós-condições	- *Crash* no sistema. - O sistema apresenta mensagem de alerta.

O Código Fonte 3.8 apresenta um exemplo de fraqueza do tipo *divide by zero*. As linhas 6 a 10 apresentam uma verificação da leitura dos dados de entrada pela função `fgets()`. Neste exemplo, a função em teste efetua uma divisão do valor “100” pelo valor recebido

como entrada e retorna o resultado da divisão (linha 12). Desta forma, se a função receber o valor zero e efetuar a divisão, causará o *crash* no sistema.

Código Fonte 3.8: Exemplo de código que apresenta fraqueza do tipo *divide by zero*

```
1  void  CWE369_Divide_by_Zero__int_fgets (){
2      int data;
3      data = -1;
4      char inputBuffer[CHAR_ARRAY_SIZE] = "";
5      /* Read data from the console using fgets() */
6      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL){
7          data = atoi(inputBuffer);
8      }else{
9      printf("fgets()failed.");10}
11     /* Possibly divide by zero */
12 printf(100 / data);13
   }
```

O Código Fonte 3.9 apresenta uma versão simplificada do caso de teste criado com o objetivo de detectar *divide by zero*, baseado no guia apresentado na Tabela 3.5. A linha 1 apresenta uma função *mock* da função `fgets`. As linhas 2 a 8 apresentam a função de teste, onde declaramos uma variável `data` e atribuímos o valor zero (linhas 3 e 4).

Código Fonte 3.9: Versão simplificada do caso de teste criado para testar *divide by zero*

```
1  char __wrap_fgets(...){return __s;}
2  static void test_juliet_rtc(...){
3      int data;
4      data = 0;
5      /*Lines of code here*/
6      CWE369_Divide_by_Zero__int_fgets();
7  assert_true(1);8   }
```

Por meio dos guias elaborados para criação de casos de teste apresentados anteriormente é possível criar *templates* de casos de teste. O Código Fonte 3.10 apresenta um exemplo de template criado baseado no guia apresentado na Tabela 3.2 para testar a função apresentada no Código Fonte 3.1 do tipo *integer overflow*. Este template foi elaborado utilizando o

Apache FreeMarker, uma biblioteca Java para gerar saída de texto com base em modelos e dados variáveis [37]. A Figura 3.4 representa o processo de criação dos arquivos de teste.

Código Fonte 3.10: Versão simplificada de template de casos de teste para testar a função apresentada no Código Fonte 3.1

```
1  #include <...>
2  #include "${pathDataSet}${fileName}"
3  ${externVar}
4  ${mockedFunctions}
5  char __wrap_fgets(...){return __s;}
6  static void test_juliet_rtc(...){
7      int data;
8      data = INT_MAX;
9      int result = data + 1;
10     /*Lines of code here*/
11     char buf[BUFSIZ];
12     freopen(...);
13     setbuf(stdout, buf);
14     ${testedFunction}
15     freopen("/dev/tty", "a", stdout);
16 assert_true(atoi(buf) == result);17
   }
18 int setup(...){...}
19 intteardown(...){...}
20 int main(...){...}
21 }
```

Pela utilização do *Apache FreeMarker* é possível definir varáveis em tempo de execução. As linhas 2, 3, 4 e 14 apresentam variáveis que são definidas durante a execução do WTT. Na linha 2 é definido o caminho para o arquivo em teste. As linhas 3 e 4 definem variáveis externas e funções *mocks*. Nas linhas 7-9 definimos a variável `data`, atribuímos um valor com `INT_MAX` e efetuamos a mesma operação apresentada na função do Código Fonte 3.1. Nas linhas 10-15 enviamos o dado de teste para a função em teste (linha 14) e preparamos o código para capturar a saída gerada pela função em teste. A linha 16 apresenta a comparação entre o resultado esperado com o resultado apresentado pela função em teste.

O resultado desta fase corresponde a dois arquivos que serão utilizados como entrada

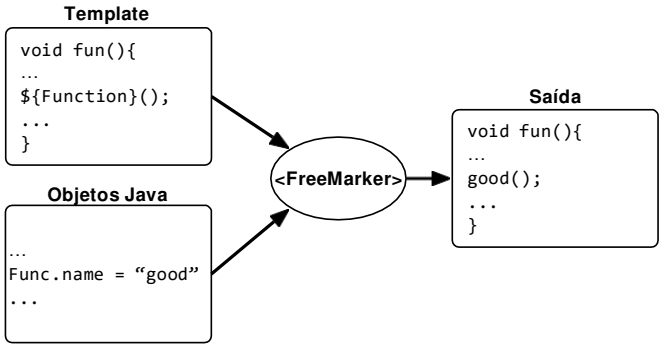

Figura 3.4: Representação do processo de criação dos arquivos de teste baseado em templates

para as próximas fases. O primeiro arquivo apresenta os casos de teste simples com dados de teste predefinidos estaticamente pelo desenvolvedor e o segundo executa o teste com dados de teste gerados pelo AFL.

Como visto na Subseção 2.1.2, o teste de conformidade apresenta o objetivo de aceitar ou rejeitar uma implementação avaliando se a implementação está em conformidade ou não com a sua especificação. Por outro lado, o propósito de teste apresenta o objetivo de observar se uma implementação apresenta um comportamento desejado, ou seja, avaliar se uma propriedade está presente ou não. Além disso, é destacado que uma suíte de teste é considerada *e-sound* quando esta pode revelar um comportamento pretendido.

No caso específico do nosso trabalho, nossas evidências foram baseadas em comportamentos que deveriam ser apresentados por implementações que apresentam fraquezas. Neste sentido, os guias para implementação dos casos de teste apresentados anteriormente, foram baseados em caraterísticas sobre fraquezas de código apresentadas pelos projetos CWE e OWASP.

A técnica proposta neste estudo foi elaborada visando testar uma propriedade. Além disso, tomamos o cuidado de observar os guias de implementação com o objetivo de evitar falsos positivos. Neste sentido, considerando o conjunto de características adotadas na elaboração dos guias de casos de testes, podemos dizer que casos de teste implementados seguindo os guias propostos irão falhar para uma implementação que apresenta características de uma fraqueza de código. Desta forma, consideramos os casos de teste elaborados seguindo os guias propostos *e-sound*, ou seja, toda implementação detectada pelos casos de teste apresenta características de uma fraqueza no código.

3.2.3 Fuzz Testing

Utilizamos *fuzz testing* em nosso *framework* com objetivo de gerar entradas aleatórias para verificar se alguma entrada causa *crash* ao programa. Para esta fase da nossa técnica utilizamos o American fuzzy lop (AFL), um *fuzzer* de força bruta e orientado a segurança que foi utilizado para detectar vulnerabilidades em grandes repositórios, como Mozilla Firefox, OpenSSL e Linux [2].

Para execução desta fase, preparamos um dos arquivos com os casos de teste gerados na fase anterior para receber as entradas fornecidas pelo AFL por meio de funções de entrada de dados, como `gets()`. O AFL gera entradas repetidamente e lista as entradas que causam *crash*. Suponha que identificamos alguma entrada que causa *crash* no programa testado. Nesse caso, confirmamos a fraqueza do código, ou seja, se um invasor enviar essa entrada específica para a função, ele pode obter vantagens ou causar algum dano ao programa.

Nesta fase, além de identificar todas as entradas que podem causar *crash*, o arquivo de teste utilizado possui a responsabilidade de salvar todas as entradas geradas pelo AFL. As entradas salvas serão utilizadas na execução dos casos de teste na próxima fase (Seção 3.2.4). Neste sentido, esta fase apresenta como resultado uma lista com todas as entradas que podem causar *crash* ao programa e uma lista com todas as entradas geradas pelo AFL.

O Código Fonte 3.11 apresenta uma versão simplificada da função utilizada na fase *fuzz testing* para testar a fraqueza apresentada no Código Fonte 3.1. A função recebe os dados de entrada do AFL (linha 4), salva a entrada recebida em uma lista (linhas 6-11), invoca a função em teste com o dado recebido (linha 15) e efetua a comparação entre o resultado gerado pela função em teste com o resultado esperado (linha 17).

Código Fonte 3.11: Versão simplificada da função para testar o Código Fonte 3.1

```
1  static void test_juliet_ft(...){
2      int data;
3      data = 0;
4      scanf("%d\n", &data);
5      int result = data + 1;
6      FILE *fileAddress;
7      fileAddress = fopen("log_afl_bad.txt", "a");
8      if (fileAddress != NULL){
9          fprintf(fileAddress, "%d\n", data);
10     fclose(fileAddress);11
       }
12     char buf[BUFSIZ];
13     freopen(...);
14     setbuf(...);
15     Integer_Overflow();
16     freopen(...);
17 assert_true(atoi(buf) == result);18
   }
```

O Código Fonte 3.12 apresenta a função de teste `test_htdbm_list` utilizada na fase *fuzz testing* para testar o Código Fonte 1.1. Neste exemplo, utilizamos a função `gets()` (linha 4) para receber as entradas do AFL. Além disso, ao mesmo tempo que o AFL identifica entradas que podem causar *crash* à função em teste, o caso de teste utilizado nesta fase tem a responsabilidade de salvar a lista de todas as entradas geradas pelo AFL (linhas 7-12).

Código Fonte 3.12: Versão simplificada da função para testar o Código Fonte 1.1

```
1  // Test function
2  static void test_htdbm_list(void** state) {
3    char *expStr;
4    gets(expStr);
5    char retStr[BUFSIZ];
6    char buf[BUFSIZ];
7    FILE *fileAddress;
8    fileAddress = fopen("log_afl.txt", "a");
9    if (fileAddress != NULL){
10     fprintf(fileAddress, "%s\n", expStr);
11   fclose(fileAddress);12
     }
13   freopen("/dev/null", "a", stderr);
14   setbuf(stderr, buf);
15   htdbm_list(expStr);
16   freopen ("/dev/tty", "a", stderr);
17 assert_string_equal(expStr,retStr);18}
```

3.2.4 Execução de Casos de Teste

Se o programa em teste não apresentar *crash* na fase anterior, não significa que o código fonte esteja livre de fraquezas. Alguns tipos, como *format string*, podem não apresentar *crash*, mesmo com a presença de fraquezas no código. Por esse motivo, precisamos executar os casos de teste para verificar o comportamento do programa. Para o exemplo apresentado no Código Fonte 1.1, o AFL não identificou nenhuma entrada que pudesse causar *crash* ao programa. Nesse sentido, executamos nossos casos de teste fornecendo entradas que podem fazer com que eles falhem.

Nesta fase, realizamos duas execuções de teste. A primeira é utilizando as entradas coletadas durante a fase de *fuzz testing*. Na fase anterior, estávamos interessados em identificar entradas que pudessem causar *crash*. O objetivo, neste momento, é verificar se alguma entrada gerada pelo AFL pode fazer com que nossos casos de teste falhem. A segunda execução é usando os dados fornecidos pelo desenvolvedor. Estes dados podem ser obtidos, por exem-

plo, a partir de *code review* e conhecimento sobre a fraqueza analisada. O Código Fonte 3.13 apresenta uma versão da função de teste utilizando dados fornecidos manualmente pelo desenvolvedor.

Código Fonte 3.13: Versão simplificada da função para testar o Código Fonte 3.1

```
1  static void test_juliet_rtc(...){
2      int data;
3      data = INT_MAX;
4      int result = data + 1;
5      char buf[BUFSIZ];
6      freopen(...);
7      setbuf(...);
8      Integer_Overflow();
9      freopen(...);
10 assert_true(atoi(buf) == result);11
   }
```

Sintetizamos todos os resultados da execução do teste em um arquivo JSON com cada entrada e seu respectivo resultado (PASSED, FAILED ou CRASH). A Tabela 3.6 apresenta uma lista de alguns resultados da execução do caso de teste do Código Fonte 3.12 com entradas do AFL.

Tabela 3.6: Exemplo simplificado do resultado da execução dos casos de teste para cada entrada gerada pelo AFL.

Id	Entrada	Resultado
1	teste	PASSED
2	%xteste	FAILED
3	test	PASSED
4	est	PASSED
5	4est	PASSED
6	Test	PASSED
7	dest	PASSED
8	lest	PASSED
9	pest	PASSED
10	vest	PASSED

Nós poderíamos utilizar outras abordagens para gerar dados de teste, como partição de classes de equivalência ou análise de valor limite, entre outros. Porém, para testes de segurança, devemos identificar dados adequados para explorar os pontos fracos. Assim, usamos a experiência do testador e o *fuzz testing*.

3.2.5 Categorização

Finalmente, a última fase do WTT corresponde a categorização dos resultados das fases *Fuzz Testing* e Execução dos Casos de Teste. A Figura 3.5 apresenta uma representação do processo de categorização.

A técnica proposta apresenta três possíveis saídas, são elas: `Não Testado (NT)`, `Detectado (D)` e `Não Detectado (ND)`. Na versão atual, WTT executa automaticamente o processo de categorização sintetizando os resultados da execução em um relatório com informações consideradas pertinentes, como tipo de fraqueza, entradas para expô-la e o resultado de cada fase.

Durante a fase de instrumentação, identificamos se é possível construir os casos de teste. Caso não seja possível por qualquer motivo, como complexidade do código, classificamos

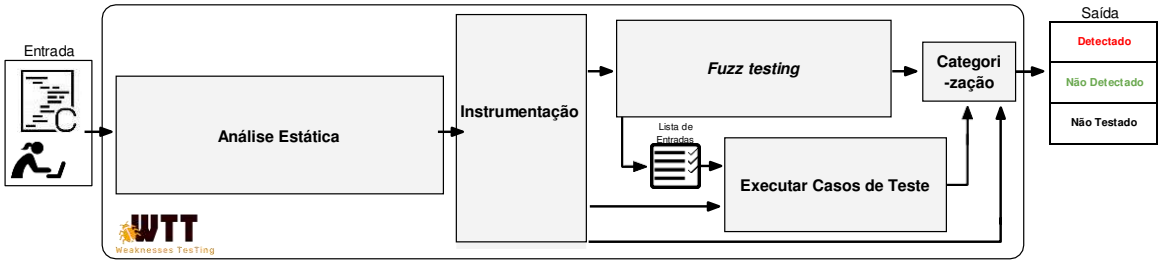

Figura 3.5: Processo de categorização do WTT.

o código como `Não Testado`. Neste caso, o processo segue diretamente para a fase de categorização. Caso seja possível construir os casos de teste, seguimos para as fases de execução dos casos de teste. Se algum caso de teste falhar ou for detectado *crash* nas fases *Fuzz Testing* e Execução dos Casos de Teste, concluímos que o código apresenta uma fraqueza e o classificamos como `Detectado`. Caso contrário, se todos os casos de teste passarem, concluímos que não foi possível confirmar a fraqueza e classificamos como `Não Detectado`.

A Tabela 3.7 apresenta um exemplo do relatório gerado pelo WTT na fase de categorização. Na coluna *FT* (*fuzz testing*) identificamos a quantidade de entradas que causaram *crashes* ao programa durante a execução da fase *fuzz testing*. Na coluna ECT (Execução dos Casos de Teste), identificamos se o caso de teste com a entrada definida pelo testador. Finalmente, na coluna AFL ECT, que corresponde a execução dos casos de teste com as entradas geradas pelo AFL, identificamos quantas entradas fizeram os casos de teste passar, falhar ou causaram *crash* ao programa.

Tabela 3.7: Exemplo da categorização resultante da técnica proposta. FT = *fuzz testing*; ECT = Execução de casos de teste; AFL ECT = Execução de casos de teste com entradas do AFL; Cat. = Categorização;

	FT	**ECT**		**AFL ECT**			
CWE	**Crashes**	**Entrada**	**Resultado**	**Pass**	**Fail**	**Crash**	**Cat.**
134	0	teste:%xex	Falhou	36.800	3.980	0	Detectado

3.3 Considerações Finais

Este capítulo apresentou inicialmente a visão geral da técnica proposta. Em seguida, foi apresentada uma descrição detalhada de cada etapa: Primeiramente, a etapa de análise estática, com o objetivo de identificar códigos suspeitos. Em seguida, apresentamos a instrumentação do código referente à criação dos casos de teste e preparação do código para execução da técnica. *fuzz testing* com o objetivo de gerar entradas aleatórias para identificar entradas que podem causar *crash* no código. Execução de casos de teste, com o objetivo de identificar algum comportamento inesperado no código em teste. Por fim, a fase de categorização apresentando a síntese do resultado gerado pela técnica. Na versão atual, WTT apresenta a fase de Instrumentação de forma manual. As fases de análise estática, *fuzz testing*, Execução de casos de teste e categorização são executadas automaticamente.

Capítulo 4

Um Estudo Avaliativo com Sistemas Reais

Neste capítulo, apresentamos um estudo empírico com o objetivo de avaliar o WTT com um conjunto de programas reais. Este estudo foi publicado na trilha de pesquisa do *XXXV Simpósio Brasileiro de Engenharia de Software* [1]. Primeiramente, apresentamos a definição do estudo (Seção 4.1) e o planejamento (Seção 4.2). A Seção 4.3 apresenta os resultados do estudo. Em seguida, apresentamos a discussão (Seção 4.4) e ameaças à validade (Seção 4.5). Finalmente, apresentamos nossas considerações finais (Seção 4.6).

4.1 Definição

O objetivo do nosso estudo consiste em analisar a técnica proposta para avaliar a dimensão de segurança na detecção de fraquezas do ponto de vista dos desenvolvedores no contexto de programas C. Este estudo nos ajudou a responder a segunda questão de pesquisa definida neste trabalho de tese, avaliando se a técnica proposta é eficaz na detecção de fraquezas. Endereçamos as seguintes questões de pesquisa:

- QP_1**:** Qual a taxa de detecção de cada tipo de fraqueza?
  Identificamos a taxa de detecção de cada tipo de fraqueza a partir do total de fraquezas testadas pelo WTT.

[1]https://dl.acm.org/doi/abs/10.1145/3474624.3474633

- **QP$_2$:** Qual a taxa de precisão *(precision)*, recuperação *(recall)* e pontuação f1 *(f1 score)*?

  Para cada *warning* testado pelo WTT, inspecionamos manualmente os resultados para identificar verdadeiro positivo, falso positivo, verdadeiro negativo e falso negativo por revisão de código nos casos de teste e código em teste. A partir desses dados, calculamos a precisão, a recuperação e a pontuação f1.

4.2 Planejamento

Nesta seção descrevemos os projetos selecionados para execução do estudo, a metodologia adotada e o ambiente de execução.

4.2.1 Seleção dos Projetos

Avaliamos nossa técnica com os projetos listados na Tabela 4.1. Consideramos alguns critérios para selecionar os alvos para realizar nossos experimentos. Todos os projetos selecionados são projetos de código aberto disponíveis online, implementados principalmente em C e atualmente ativos.

Tabela 4.1: Visão geral dos projetos selecionados para execução do estudo.

Projeto	Descrição	Versão
Genymobile	Controle de dispositivos android	v1.6
Lede	Sistema operacional	v19.07.4
Libexpat	Analisador XML	v2.2.1
Libomemo	Biblioteca de criptografia	v0.7.1
Libxml2	Biblioteca/kit de ferramentas XML	v2.9.10
Socket_wrapper	Teste cliente-servidor	v1.3.0

Descrevendo os projetos selecionados, temos o Genymobile [2], um aplicativo para exibir e controlar dispositivos android conectados via USB. O projeto Lede [3] corresponde a um

[2]https://github.com/Genymobile/scrcpy

[3]https://github.com/lede-project/source

sistema operacional Linux direcionado a dispositivos embarcados para fornecer um sistema de arquivos gravável com gerenciamento de pacotes e permite personalizar o dispositivo por meio de pacotes. Libexpat [4] corresponde a um analisador XML orientado a fluxo de biblioteca C. Libomemo [5] é um *XMPP Extension Protocol (XEP)* para criptografia multicliente de ponta a ponta. Libxml2 [6], conhecido por ser portátil e funcionar sem problemas em vários sistemas, é um analisador XML C e conjunto de ferramentas desenvolvido para o projeto Gnome. Socket_wrapper [7] é um conjunto de ferramentas para teste cliente-servidor, que permite que os desenvolvedores testem softwares em máquinas UNIX com acesso limitado à rede.

4.2.2 Metodologia

Este estudo propõe uma avaliação de uma técnica de teste para ser executada durante os estágios iniciais do processo de desenvolvimento de programas C.

Primeiramente, executamos análise estática nos projetos selecionados. Devido ao grande número de *warnings* recuperados pela ferramenta de análise estática, filtramos os *warnings* por seu nível de risco (médio ou alto) e as fraquezas que WTT pode detectar. Neste sentido, selecionamos os arquivos C que apresentam algum dos *warning* selecionados, de acordo com os critérios citados anteriormente, e executamos as demais fases da nossa técnica. Posteriormente, isolamos a função que será testada e elaboramos os casos de teste. Nós rotulamos como `NÃO TESTADO (NT)` os *warnings* para os quais o código tem problemas, como dependências que não podemos tratar. Rotulamos como `DETECTADO (D)` os *warnings* selecionados na fase de análise estática que WTT classificou como uma fraqueza, ou seja, algum caso de teste falhou. Por outro lado, marcamos como `NÃO DETECTADO (ND)` os *warnings* avaliados pelo WTT e não classificados como uma fraqueza.

Considerando que os casos de teste podem apresentar problemas de implementação que podem influenciar o resultado da execução, efetuamos uma revisão de código manual nos casos de teste utilizados neste estudo para tentar identificar problemas de implementação. Os casos de teste podem fornecer falsos positivos ou falsos negativos devido a defeitos em

[4]https://github.com/libexpat/libexpat
[5]https://github.com/gkdr/libomemo
[6]https://github.com/GNOME/libxml2
[7]https://git.samba.org/?p=socket_wrapper.git

seu código, como configuração ou asserções ausentes e configuração ou asserções incorretas que tornam incerto sobre a classificação de uma aprovação ou falha como verdadeira. Além disso, mesmo se o caso de teste for bem formado, a execução pode perder dados que desencadeariam a falha.

A Figura 4.1 apresenta nosso processo de classificação de *warnings*. Nós inspecionamos manualmente as fraquezas rotuladas como `detectado` e `não detectado` por revisão de código, classificando-os como verdadeiro positivo, falso positivo, verdadeiro negativo e falso negativo, com o objetivo de avaliar a precisão de nossa técnica. Para os casos rotulados como `detectado`, se identificarmos algum defeito na construção dos casos de teste que possa dar um falso alarme, classificamos como *falso positivo*. Caso contrário, se não conseguirmos identificar nenhum defeito, classificamos como *verdadeiro positivo*. Para os casos rotulados como `não detectado`, inspecionamos os casos de teste e, se identificarmos algum defeito na construção dos casos de teste que possa prejudicar a detecção, classificamos como *falso negativo*. Caso contrário, classificamos como *verdadeiro negativo*.

Figura 4.1: Processo de classificação de *warnings*.

4.2.3 Ambiente de Execução

Executamos o estudo no sistema operacional Ubuntu 16.04 LTS (2.4 GHz, i7 e 16GB RAM). Utilizamos o Flawfinder 1.31 como ferramenta de análise estática. Para instrumentar o código, gerar casos de teste e executar *fuzz testing*, utilizamos o GCC 5.4.0 (Ubuntu 5.4.0-6ubuntu1 16.04.10), cmocka 1.1.3 e o American Fuzzy Lop (2.52b), respectivamente. Definimos o número de ciclos do AFL em 30, gerando uma média de 27,4 mil entradas por

execução.

4.3 Resultados

A seguir, descrevemos os principais resultados que embasam as respostas às questões de pesquisa definidas neste estudo. O código do WTT e alguns exemplos de execução estão disponíveis online [8].

Nossa avaliação retornou um total de 1.729 *warnings* de 6 projetos. Nós filtramos, de forma manual, os *warnings* retornados pelos tipos de fraquezas abordadas neste trabalho e seu nível de risco classificado por análise estática. Para execução deste estudo, foram considerados os níveis de risco médio ou alto, de acordo com a ferramenta de análise estática. Desta forma, foram selecionados 103 *warnings* localizados em 35 arquivos C. Então, executamos WTT em 103 *warnings* e rotulamos 24 deles como detectados (fraquezas confirmadas), 37 deles como não detectados e 42 deles como não testados. Além disso, categorizamos os *warnings* em três tipos de fraquezas: *Buffer Overflow*, *Integer Overflow* e *Format String*. A Tabela 4.2 apresenta o total de *warnings* analisados para cada tipo, indicando quantos foram testados ou não.

Tabela 4.2: *Warnings* analisados classificados por tipo.

	Testado		**Não Testado**
	Detectado	Não Detectado	
Buffer Overflow	1	27	18
Format String	22	8	24
Integer Overflow	1	2	0

Do total de *warnings* rotulados como detectados, classificamos 22 deles como *verdadeiro positivo*, ou seja, *warnings* classificados como detectados pelo WTT e confirmados por revisão de código. Além disso, classificamos um total de 2 *warnings* como *falso positivo*, ou seja, identificamos, por revisão de código, algum defeito na construção dos casos de teste que o tornam incerto sobre a classificação do seu resultado de execução. Além disso, do total de *warnings* rotulados como não detectados, classificamos 36 deles como *verdadeiro*

[8]https://github.com/WeaknessesTesting/wtt

negativo, ou seja, *warnings* classificados como não detectados pelo WTT e confirmados por revisão de código, e um total de 1 aviso como *falso negativo*.

Com o objetivo de avaliar a precisão de nossa técnica, calculamos métricas de avaliação, como *precision*, *recall* e *f1 score*. A métrica *precision* descreve a proporção das fraquezas detectadas e confirmadas em relação ao total de fraquezas detectadas, como pode ser visto em (4.1). O valor de precisão varia de 0, a pontuação mais baixa disponível, a 1, a pontuação mais alta disponível. Uma pontuação de precisão igual a 1 significa que todas as fraquezas reais foram detectadas. Nossos resultados apresentam uma pontuação de precisão de aproximadamente 0.92.

$$Precision = \frac{verdadeiropositivo}{verdadeiropositivo + falsopositivo} \tag{4.1}$$

Por outro lado, a métrica *recall* descreve quantos do total de verdadeiro positivos (verdadeiro positivo + falso negativo) nossa técnica detectou e rotulou como verdadeiro positivo, como pode ser visto em (4.2). Assim como a métrica *precision*, *recall* varia entre 0 e 1. Nossos resultados apresentam *recall* de 0.96.

$$Recall = \frac{verdadeiropositivo}{verdadeiropositivo + falsonegativo} \tag{4.2}$$

Por fim, definimos a pontuação da F1 com base nas métricas anteriores, conforme mostrado em (4.3). Essa métrica geralmente é usada para descrever a precisão e o recall de uma só vez. Assim como nas métricas anteriores, ela retorna um valor entre 0 e 1. Nossos resultados apresentam uma pontuação f1 de aproximadamente 0.94.

$$f1 = 2.\frac{precision.recall}{precision + recall} \tag{4.3}$$

A Tabela 4.3 resume os resultados das métricas citadas para cada tipo de fraqueza.

A Tabela 4.4 apresenta as principais características de cada fraqueza classificada como detectada pelo WTT. Apresenta a linha da fraqueza no arquivo (WL), o tipo de fraqueza relacionado ao CWE, o resultado da revisão do código (CR) como verdadeiro positivo (VP) ou falso positivo (FP). Além disso, apresentamos o resultado de cada fraqueza na fase de execução do caso de teste com os valores especiais (ECT), apresentamos o número de entradas (NI) geradas e travamentos *(Crashes)* identificados na fase de *fuzz testing*. Por fim,

Tabela 4.3: Métricas para cada tipo de fraqueza: Verdadeiro Positivo (VP); Falso Positivo (FP); Verdadeiro Negativo (VN); Falso Negativo (FN); *Precision* (P), *Recall* (R) e *F1 score*.

Fraqueza	VP	FP	VN	FN	P	R	F1
Buffer Overflow	1	0	26	1	1.00	0.50	0.67
Format String	20	2	8	0	0.91	1.00	0.95
Integer Overflow	1	0	2	0	1.00	1.00	1.00
Todos	22	2	36	1	0.92	0.96	0.94

apresentamos a fase em que usamos as entradas geradas na fase de fuzzing para exercitar nossos casos de teste (ECT AFL). Nesse sentido, apresentamos o número de entradas que fizeram os casos de teste falhar (IFTC), o número de entradas que fizeram nossos casos de teste passarem (IPTC) e o número de entradas que causaram travamentos (IC).

Após a apresentação dos dados relativos às fraquezas de código analisadas, é possível responder as questões de pesquisa da seguinte forma.

- QP_1**:** Qual a taxa de detecção de cada tipo de fraqueza?

  Do total de *warnings* testados, 36% foram detectados como fraquezas pelo WTT (verdadeiros positivos), onde 1,6% são *Buffer Overflow*, 32,8% são *Format String* e 1,6% são *Integer Overflow.* Um total de 59% dos *warnings* testados foram descartados (verdadeiro negativo), onde 42,6% são *Buffer Overflow*, 13,1% são *Format String* e 3,3% são *Integer Overflow*. Além disso, 3.4% foram classificados como falsos positivos e 1.6% como falsos negativos.

- QP_2**:** Qual a taxa de precisão *(precision)*, recuperação *(recall)* e pontuação f1 *(f1 score)*?

  Conforme apresentado na Tabela 4.3, do total de casos detectados pelo WTT, classificamos 91.67% deles como *verdadeiro positivo* e 8.33% deles como *falso positivo*. Além disso, nossos resultados apresentaram 0.92 de *precision*, 0.96 de *recall*, resultando em um *f1* de 0.94.

Tabela 4.4: Resultados da execução do WTT em projetos C para detectar fraquezas; LOC = Linhas de código; WL = Linha da fraqueza; CR = revisão de código; ECT = Execução de casos de teste com valor especial; FT = *fuzz testing*; NI = Número de entradas; AFL ECT = Execução de Casos de Teste com entradas geradas pelo AFL; IFTC = Entradas que fizeram os casos de teste falhar; IPTC = Entradas que fizeram os casos de teste passarem; IC = entradas que causaram *crashes*;

								FT	**AFL ECT**		
Projeto	**Arquivo**	**LOC**	**WL**	**CWE**	**CR**	**ECT**	**NI**	**Crashes**	**IFTC**	**IPTC**	**IC**
Genymobile	command.c	175	107	134	FP	PASSED	17,473	1	1,387	16,075	11
Genymobile	command.c	175	126	134	FP	PASSED	17,258	1	1,367	15,880	11
Lede	confdata.c	1,006	40	134	VP	FAILED	17,454	1	1,329	16,109	16
Lede	confdata.c	1,006	49	134	VP	FAILED	17,229	0	17,186	0	43
Lede	confdata.c	1,006	104	120	VP	PASSED	118,968	0	0	118,926	42
Lede	menu.c	578	26	134	VP	FAILED	39,567	3	4,635	34,932	0
Lede	util.c	542	120	120	VP	FAILED	116,130	1	58,792	56,606	732
Lede	mconf.c	965	777	134	VP	PASSED	15,325	0	15,282	0	43
Libexpat	outline.c	66	64	120	VP	PASSED	16,898	0	0	16,897	1
Libomemo	libomemo.c	1,197	948	134	VP	PASSED	4,305	0	4,305	0	0
Libomemo	libomemo.c	1,197	956	134	VP	PASSED	4,275	0	536	3,739	0
Libomemo	libomemo.c	1,197	399	190	VP	PASSED	3,206	0	135	3,071	0
Libxml2	libxml.c	3,175	1,608	134	VP	FAILED	36,002	0	3,876	32,126	0
Libxml2	libxml.c	3,175	1,632	134	VP	FAILED	35,790	0	3,862	31,928	0
Libxml2	gjobread.c	203	26	134	VP	FAILED	17,623	0	1,351	16,259	13
Libxml2	runsuite.c	980	171	134	VP	PASSED	16,528	0	1,225	15,279	24
Libxml2	runsuite.c	980	178	134	VP	PASSED	16,528	0	1,225	15,279	24
Libxml2	runsuite.c	980	191	134	VP	PASSED	16,528	0	1,225	15,279	24
Libxml2	runxmlconf.c	486	116	134	VP	FAILED	17,441	0	17,417	0	24
Libxml2	runxmlconf.c	486	123	134	VP	FAILED	17,441	0	17,417	0	24
Libxml2	xmlstring.c	572	558	134	VP	FAILED	17,706	0	17,676	0	30
Libxml2	xmlstring.c	572	584	134	VP	PASSED	15,675	1	866	14,803	6
Socket_wrapper	socket_wraper.c	6,021	1,442	134	VP	PASSED	27,978	0	218	27,758	2
Socket_wrapper	socket_wraper.c	6,021	444	134	VP	PASSED	16,270	0	0	16,269	1

4.4 Discussão

Esta seção discute com mais detalhes os resultados do estudo sobre a relevância e o escopo de aplicação do WTT.

Neste estudo, avaliamos o código-fonte de 6 projetos reais de código aberto, onde foram selecionados um total de 103 *warnings* seguindo alguns critérios, como os tipos de fraquezas abordadas neste trabalho e seu nível de risco classificado por análise estática. Consideramos o subconjunto dos tipos de fraquezas abordadas neste trabalho pois, além de estarem entre os *24 Deadly Sins of Software Security* [53], são classificados como inseridos na fase de implementação. Além disso, de acordo com o projeto CWE, eles apresentam alta e média probabilidade de serem explorados [20].

Observação 1: *WTT detectou três tipos de fraquezas em programas reais classificadas pelo CWE.*

Cheng et al. [114] e Haijung et al. [113] apresentaram exemplos de fraquezas detectadas no escopo de uma função. Uma fraqueza reportada por Cheng et al. [114] foi discutida com os desenvolvedores do projeto avaliado, que argumentaram que a entrada que causou o problema não pode ser passada para a função. No entanto, apesar desse fator, o projeto CVE classificou a fraqueza reportada como uma vulnerabilidade [9]. Isso mostra que uma organização competente o reconheceu como uma fraqueza que precisa ser corrigida, independente do estado atual de fatores externos do sistema. Da mesma forma, tentamos confirmar uma das fraquezas detectadas pelo WTT. Atualmente, nossa solicitação está classificada como "reservada" no projeto CVE com a enumeração CVE-2020-16603 [10].

Observação 2: *Uma fraqueza dentro de uma função precisa ser corrigida.*

Com WTT, podemos confirmar que alguns dos avisos relatados pela análise estática são de fato fraquezas que podem ser exploradas. O caso de teste com falha e os dados utilizados são evidências concretas de que a exploração pode ser realizada no nível de unidade. Além

[9] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17985
[10] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-16603

disso, a alta taxa de verdadeiros negativos identificados em nossos resultados corrobora com o conhecimento amplamente aceito de que as ferramentas de análise estática relatam um número relevante de falsos positivos, nestes casos, *warnings* que podem ser descartados. Com WTT os avisos podem ser analisados com mais precisão para que os desenvolvedores possam se concentrar nos avisos mais críticos a serem tratados.

Observação 3: *WTT pode apoiar a identificação de fraquezas reais entre um conjunto de warnings.*

O Código Fonte 4.1 apresenta uma fraqueza rotulada como detectada pelo WTT e confirmada pela revisão do código (verdadeiro positivo). A fase de análise estática relatou um aviso relacionado à função `vsnprintf` e identifica o *warning* como relacionado a um possível problema de *Format String* (CWE-134). Essa função envia a saída formatada para uma string usando uma lista de argumentos passada a ela e, se um invasor puder influenciar a formatação da string, ela pode ser explorada. Este tipo de fraqueza é abordado neste trabalho e apresenta um nível de risco de quatro (alto risco). Por esses motivos, esta fraqueza foi selecionada para ser avaliada. Executando o WTT, os casos de teste passaram na fase de execução do caso de teste, fornecendo valores específicos definidos pelo desenvolvedor. Na fase de *fuzz testing*, WTT identifica um *crash* causado por uma das 15.675 entradas geradas. Além disso, executando os casos de teste com as entradas geradas durante o *fuzz testing*, WTT identificou 866 entradas que fazem o teste falhar e seis entradas que causaram *crash* no sistema.

Código Fonte 4.1: Trecho de código do arquivo `xmlstring.c` do projeto Libxml.

```
1 int xmlStrVPrintf(...) {
2     int ret;
3     if((buf == NULL) || (msg == NULL)) {
4     return(-1);5
      }
6     ret = vsnprintf((char *) buf, len, (const char *) msg, ap
          );
7     buf[len - 1] = 0;
```

```
8       return(ret);
9   }
```

Durante a execução da fase de análise estática, WTT reportou um *warning* classificado como um potencial *Format String* (CWE-134) na função `adb_forward_remove`, apontando a fraqueza para a função `sprintf` na linha 4 do Código Fonte 4.2 e classificando como nível de risco alto. Considerando os guias propostos para elaboração dos casos de teste, para o tipo de fraqueza apresentado neste exemplo, foi elaborado um caso de teste onde comparamos uma cadeia de caracteres enviada como entrada com a cadeia de caracteres que foi impressa pela função em teste. No entanto, nós o rotulamos como falso positivo, pois identificamos que o formato da string `PRIu16` foi definido em outra parte do código, fazendo com que WTT identifique-o como uma fraqueza, já que o caso de teste perde a atribuição de dependência correta.

Código Fonte 4.2: Trecho de código do arquivo `command.c` do projeto Genymobile.

```
1  process_t
2  adb_forward_remove(...) {
3     char local[4 + 5 + 1]; // tcp:PORT
4     sprintf(local,"tcp:%" PRIu16, local_port);
5     const char *const adb_cmd[] = {..., local};
6  return     adb_execute(serial, adb_cmd, ARRAY_LEN(adb_cmd));
7  }
```

O Código Fonte 4.3 apresenta a fraqueza classificada como falso negativo, como apresentado na Tabela 4.3. WTT classificou uma fraqueza como `não detectado`, porém, classificamos como falso negativo durante a revisão do código. Na fase de análise estática foi apontado um *warning* do tipo *buffer overflow* relacionado à função *strcat* (linha 16) do arquivo *confdata* do projeto *LEDE*. A função `strcat` concatena duas *strings*, e o resultado é armazenado na *string* de destino. WTT classificou este caso como *Não Detectado*. No entanto, avaliando o caso de teste utilizado na execução foi identificado um erro de implementação que fez o caso de teste não falhar. Por este motivo, este *warning* foi classificado como falso negativo, impactando na taxa de *Recall* dos resultados deste estudo. O processo de revisão de código executado tem o objetivo de identificar possíveis problemas de imple-

mentação, tanto no código em teste quanto no caso de teste criado, que possam impactar no resultado da execução do WTT.

Código Fonte 4.3: Versão simplificada do código do arquivo `command.c` do projeto Lede.

```
1  static char *conf_expand_value(const char *in){
2    static char res_value[...];
3    // Lines of code
4    res_value[0] = 0;
5    while ((src = strchr(in, '$'))) {
6      strncat(res_value, in, src - in);
7      src++;
8          dst = name;
9          while (...)
10           *dst++ = *src++;
11         *dst = 0;
12         // Lines of code
13         strcat(res_value, get_string(sym));
14   in= src;15     }
16   strcat(res_value, in);
17 return    res_value;
18 }
```

Devido ao grande número de dados de entrada gerados durante a execução do teste (Fases *fuzz testing* e Execução de Casos de Teste), medimos o tempo total de execução do teste. Como esperado, as fases que dependem do AFL para geração e execução da entrada demoram mais. A Tabela 5.6 apresenta o tempo gasto em cada fase do WTT. A execução do WTT levou em média 0,05s para executar a fase ECT e em média 58,18s para executar o *fuzz testing* considerando 30 ciclos do AFL. Finalmente, WTT levou em média 74,10s para executar os casos de teste com as entradas geradas pelo AFL (ECT AFL).

Embora neste estudo não tenhamos medido o tempo gasto durante a fase de instrumentação do WTT, acreditamos que os custos do uso da técnica podem depender desta fase.

Tabela 4.5: Tempo Gasto em Cada Fase do WTT; ECT = Execução de casos de teste; FT = *fuzz testing*; ECT/AFL = Execução de Casos de Teste com entradas geradas pelo AFL;

		Tempo (s)			
Projeto	**Arquivo**	ECT	FT	RTC/AFL	Total
GenyMobile	command	0.06	64.02	36.01	100.09
GenyMobile	command	0.05	59.20	30.09	89.34
Lede	confdata	0.07	23.38	22.61	46.06
Lede	confdata	0.06	21.11	20.09	41.26
Lede	confdata	0.06	98.11	14.03	112.47
Lede	confdata	0.06	27.11	52.08	79.25
Lede	mconf	0.07	23.38	22.61	46.06
Lede	menu	0.05	40.14	30.70	70.89
Lede	util	0.08	80.13	60.52	140.30
Libexpat	outline	0.05	34.73	42.76	77.55
Libomemo	libomemo	0.03	10.43	10.79	21.25
Libomemo	libomemo	0.03	11.32	20.06	31.41
Libomemo	libomemo	0.03	10.08	16.01	26.12
Libxml2	libxml	0.05	64.30	88.60	152.95
Libxml2	libxml	0.05	63.51	67.11	140.61
Libxml2	gjobread	0.07	19.54	23.51	43.12
Libxml2	runsuite	0.06	20.32	77.13	97.51
Libxml2	runsuite	0.06	20.32	63.05	83.43
Libxml2	runsuite	0.06	20.32	72.02	92.40
Libxml2	runxmlconf	0.07	24.78	24.73	49.58
Libxml2	runxmlconf	0.07	20.62	26.65	47.34
Libxml2	xmlstring	0.06	20.68	23.77	44.51
Libxml2	xmlstring	0.06	18.76	18.90	37.72
Socket_	socket_wrap	0.03	90.29	82.77	173.09
Socket_	socket_wrap	0.05	90.55	49.85	140.45
	Média	0.05	58.18	74.10	84.80

Pudemos observar que esses custos variam dependendo de fatores como a complexidade do código, principalmente no que diz respeito ao nível em que um sistema facilita o estabelecimento de critérios de teste (testabilidade) e à expertise do testador. De modo geral, a testabilidade é determinada por dois problemas comuns: como fornecer os valores de teste para o software e como observar os resultados da execução do teste [9]. Portanto, o uso de modelos e diretrizes para lidar com questões específicas nesta fase é uma estratégia-chave do WTT para torná-lo viável.

Observação 4: *Os custos de uso do WTT dependem principalmente da testabilidade do código e das habilidades do testador.*

A partir dos resultados apresentados na Tabela 4.4, para a maioria dos casos detectados, um caso de teste falhou ou foi observada uma falha ao considerar os valores gerados pelo *fuzzer* (ver colunas IFTC e IC). Isso confirma a expectativa de que o *fuzz testing* é eficaz para expor fraquezas. No entanto, observou-se que o *fuzz testing* não detectou *crash* em todos os casos classificados como detectados pelo WTT (ver a coluna Crashes). A execução do caso de teste automatizado CMocka com valores AFL foi determinante para expor as fraquezas.

Observação 5: *O uso combinado do CMocka e AFL contribuiu para uma melhor taxa de detecção*.

Observando os tipos específicos de *warnings*, nossos resultados mostram que *Format String* é o tipo que apresenta maior ocorrência, seguido por *Buffer Overflow*. Além disso, a maioria dos avisos de *Buffer Overflow* testados (26 de 28) não foram detectados como fraqueza pelo WTT e foram classificados como verdadeiro negativo, enquanto a maioria dos *warnings* do tipo *Format String* (20 de 30) foram detectados como fraquezas pelo WTT e nós os classificamos como verdadeiros positivos. Na prática, *Buffer Overflow* foi mais reportado como vulnerabilidade do que *Format String*, especialmente em 2020 [110; 111].

Observação 6: *Format String foi o tipo de fraqueza mais detectado e Buffer Overflow foi o tipo de aviso mais descartado.*

Finalmente, com o objetivo de verificar se os casos de teste utilizados durante este estudo apresentam comportamento não determinísticos, conduzimos um experimento de apoio. Executamos duas rodadas onde cada caso de teste foi executado dez vezes utilizando os mesmos dados de teste. Fizemos o levantamento dos resultados de cada fase da técnica em todas as execuções. Identificamos dois casos em que o comportamento não determinístico foi observado. Para os casos que apresentaram comportamento não determinístico, identificamos a fase da técnica onde foi constatado o comportamento e o resultado que diverge nas execuções.

A Tabela 4.6 apresenta os dois casos em que foi observado comportamento não determinísticos. Em cada coluna de replicação (Primeira e Segunda Replicação) apresentamos apenas um resultado para cada fase pois não houve alterações entre as dez execuções de cada replicação.

Tabela 4.6: Avaliação de Comportamento Não Determinístico; IFTC = Quantidade de *fail*; IPTC = Quantidade de *pass*; IC = Quantidade de *crash*;

Projeto	Arquivo	Linha	CWE	Primeira Execução			Primeira Replicação			Segunda Replicação		
				IFTC	IPTC	IC	IFTC	IPTC	IC	IFTC	IPTC	IC
libxml2	libxml	1608	Format String	3,876	32,126	**0**	3,985	31,462	**555**	3,985	31,462	**555**
libxml2	libxml	1632	Format String	3,862	31,928	**0**	3,95	31,803	**37**	3,95	31,803	**37**

Além disso, listamos alguns fatores que podem ter contribuído para a alteração dos resultados durante a execução das replicações: (i) quantidade de recursos de armazenamento e processamento disponíveis durante cada execução; (ii) variáveis globais no projeto analisado (Isolar a função testada durante a fase de instrumentação). Para tentar mitigar este tipo de comportamento, adotamos algumas medidas, tais como: (i) utilizamos bibliotecas recomendadas pelo projeto; (ii) a máquina utilizada ficou dedicada para execução desta atividade; (iii) Nossos casos de teste, em sua maioria, são independentes, evitando que a interação entre diferentes métodos de teste interfira no resultado do teste.

4.5 Ameaças à Validade

A ameaça à validade externa do nosso estudo se refere aos projetos selecionados que podem não ser representativos. No entanto, avaliamos seis projetos de código aberto, amplamente usados e implementados principalmente em C.

Quanto às ameaças à validade interna, construímos manualmente os casos de teste para executar a nossa técnica. O autor desta tese conduziu este processo manual. Tentamos mitigar este processo por meio de consultas constantes aos orientadores do projeto. Além disso, ao avaliar os resultados do estudo e buscar falsos positivos ou falsos negativos, inspecionamos manualmente os casos de teste e o código em teste, com o objetivo de verificar algumas práticas de programação inadequadas na construção dos casos de teste. Nesse sentido, esta verificação manual depende de alguns fatores, como nossa perícia, e erros humanos podem ocorrer. Portanto, adotamos a revisão de código como prática padrão.

Consideramos os filtros utilizados na execução da fase de análise estática como ameaça à validade de construção. Sabemos que a execução de análise estática em programas reais com muitas linhas de código pode retornar uma quantidade de *warnings* impraticável de ser avaliada. Nosso objetivo foi alcançar uma quantidade de *warnings* possível de ser avaliada, considerando o tempo disponível para execução do estudo. No entanto, é possível que fraquezas de código tenham sido desconsideradas pela aplicação dos filtros.

Em relação à validade da conclusão, definimos o número de ciclos do AFL para cada avaliação. Diferentes durações de execução podem afetar o resultado devido às entradas geradas. No entanto, acreditamos que o número de entradas geradas pelo AFL em 30 ciclos é suficiente para nosso propósito. Mesmo assim, pode haver fraquezas não detectadas pelo WTT por este motivo.

4.6 Considerações Finais

Neste capítulo, apresentamos um estudo exploratório com o objetivo de avaliar a técnica proposta com um conjunto de programas reais. Por meio do estudo executado, foi observado que o WTT (i) detectou três tipos de fraquezas em sistemas reais, (ii) sua utilização pode apoiar a identificação de fraquezas entre um conjunto de *warnings*, (iii) o custo de sua exe-

cução depende principalmente da testabilidade do código e das habilidades do testador, (iv) a utilização do CMocka e AFL contribuiu para uma melhor taxa de detecção e, finalmente, (v) *Format String* foi o tipo de fraqueza mais detectado. Além disso, foi apresentado um estudo de apoio para avaliar comportamento não determinístico.

Capítulo 5

Um Estudo Empírico com Exemplos de Vulnerabilidades

Neste capítulo, apresentamos um estudo empírico com o objetivo de avaliar o WTT com um conjunto vulnerabilidades confirmadas. Para execução deste estudo foi selecionado o conjunto de dados *Juliet Test Suite*, uma coleção de exemplos de código-fonte que apresentam vulnerabilidades sintetizados em diferentes CWEs [92].

Apresentamos primeiramente a definição do estudo (Seção 5.1) e o seu planejamento (Seção 5.2). Em seguida, apresentamos os resultados na Seção 5.3 e a discussão na Seção 5.4. Finalmente, apresentamos as ameaças à validade (Seção 5.5) e as nossas considerações finais (Seção 5.6).

5.1 Definição

O objetivo deste estudo consiste em avaliar o framework WTT com o propósito de verificar sua acurácia no que diz respeito a taxa de detecção de fraquezas do ponto de vista dos testadores no contexto do conjunto de dados *Juliet Test Suite* para C/C++. Endereçamos as seguintes questões de pesquisa para guiar este estudo:

- QP_1**:** Qual a taxa de detecção do WTT para cada tipo de fraqueza testada?
  Para execução deste estudo, foram considerados os arquivos C das seguintes CWEs: *CWE-190*, *CWE-191*, *CWE-369* e *CWE-476*. Considerando os tipos de fraquezas abor-

dadas neste estudo, identificamos a taxa de detecção de cada tipo de CWE a partir do total de fraquezas testadas pelo WTT.

- QP$_2$**:** Qual a taxa de fraquezas encontradas dentro das funções testadas na fase de análise estática?

  Para cada função testada neste estudo (*bad* e *good*), avaliamos se WTT identificou a fraqueza dentro da função na fase de análise estática, considerando os filtros aplicados da ferramenta Flawfinder. Executamos a fase de análise estática de duas formas, (i) considerando os filtros adotados no estudo anterior (Capítulo 4) e (ii) sem os filtros da ferramenta de análise estática. Nossa intenção é verificar se esta fase é capaz dar indícios da presença de fraquezas no código das duas formas. Além disso, avaliar o potencial da fase de análise estática em incluir falsos positivos (funções *good*) e descartar verdadeiros positivos (funções *bad*).

- QP$_3$**:** Qual a taxa de *precision*, *recall* e *f1* o WTT apresentou?

  O conjunto de dados objeto deste estudo apresenta arquivos C contendo exemplos de código sintético com diferentes tipos de fraquezas. Os arquivos apresentam funções de dois tipos principais: `bad` (funções que apresentam fraquezas) e `good` (funções em que a fraqueza foi corrigida). Neste sentido, se WTT classificar alguma função `bad` como `detectado`, classificamos o resultado da execução como verdadeiro positivo, caso contrário, se WTT classificar como `não detectado`, classificamos como falso negativo. Por outro lado, se WTT classificar como `detectado` alguma função do tipo `good`, classificamos o resultado da execução como falso positivo, caso contrário, classificamos como verdadeiro negativo. A partir dos dados coletados de cada execução, calculamos as taxas de *precision*, *recall* e *f1 score*. Utilizamos as mesmas equações utilizadas no estudo anterior para calcular as métricas consideradas neste estudo (Equações 4.1, 4.2 e 4.3).

5.2 Planejamento

Nesta seção descrevemos o conjunto de dados selecionados para execução do estudo, a metodologia adotada e o ambiente de execução.

5.2.1 Seleção do Conjunto de Dados

A proposta inicial para execução deste estudo seria comparar WTT com outras abordagens presentes na literatura. Para este fim, consideramos critérios para seleção, como tipos de fraquezas abordadas, dados disponíveis na web, detecção de fraquezas a nível de unidade e que, assim como o WTT, considerasse dados de entrada enviados pelo console ou por arquivos. Tivemos dificuldade em selecionar um trabalho para execução de um estudo comparativo, na maioria das vezes, pela indisponibilidade dos dados e ferramentas de execução dos estudos.

Desta forma, consideramos o conjunto de dados utilizado no trabalho apresentado por Mouzarani e Sadeghiyan [82] para execução deste estudo. Eles apresentaram uma proposta inicial de um método que utiliza *concolic execution* (Execução concreta + simbólica) para detecção de vulnerabilidades específicas e utilizaram o conjunto de dados *Juliet Test Suite* para C e C++ versão 1.2, que corresponde a um conjunto de dados que apresenta 61.387 arquivos com exemplos de 118 diferentes CWEs. Este conjunto de dados foi utilizados em diversos trabalhos [109; 96; 95; 82; 72; 118].

No estudo selecionado, foram consideradas fraquezas de quatro tipos: *CWE-190: Integer Overflow or Wraparound*, *CWE-191: Integer Underflow (Wrap or Wraparound)*, *CWE-369: Divide By Zero* e *CWE-476: NULL Pointer Dereference*. Além disso, foram considerados apenas arquivos C que recebem dados de entrada por meio do teclado, utilizando as funções de entrada de dados `fgets` e `fscanf`, ou por meio de arquivos. No entanto, analisando os arquivos do tipo *CWE-476: NULL Pointer Dereference*, é possível perceber que suas funções não recebem entrada de dados. Além disso, Mouzarani e Sadeghiyan [82] afirmaram que fizeram modificações nos arquivos do tipo CWE-476 tornando o ponteiro dependente dos dados de entrada.

A Tabela 5.1 apresenta uma visão geral da disposição dos arquivos do tipo *CWE-190: Integer Overflow or Wraparound*. O conjunto de dados *Juliet Test Suite* apresenta uma série de 56 arquivos onde o objetivo é executar as mesmas operações. Por exemplo, para a CWE-190, *Juliet Test Suite* apresenta 56 arquivos que, utilizando a função de entrada *fscanf*, recebe um valor do tipo inteiro, executa uma operação de adição e apresenta o resultado. Entre esses 56 arquivos que executam a mesma operação, são efetuadas pequenas modificações, como por exemplo, avaliar variáveis definidas estaticamente ou chamar funções para efetuar as operações em outros arquivos.

Tabela 5.1: Visão geral dos arquivos analisados do tipo *CWE-190: Integer Overflow or Wraparound*.

Fraqueza	Entrada de Dados	Tipo	Operação
CWE-190	fscanf	char	Adição
	fscanf	char	Multiplicação
	fscanf	char	Raiz
	fscanf	int	Adição
	fscanf	int	Multiplicação
	fscanf	int	Raiz
	fscanf	int64	Adição
	fscanf	int64	Multiplicação
	fscanf	int64	Raiz
	fscanf	short	Adição
	fscanf	short	Multiplicação
	fscanf	short	Raiz
	fscanf	uint	Adição
	fscanf	uint	Multiplicação
	fscanf	uint	Raiz
	fgets	int	Adição
	fgets	int	Multiplicação
	fgets	int	Raiz

Ademais, pela indisponibilidade dos dados do estudo e pela falta de clareza de quais arquivos considerados, não foi possível realizar um estudo comparativo do WTT com a técnica apresentada em [82]. Chegamos a entrar em contato com os autores solicitando os dados para replicação do estudo, porém, sem sucesso.

5.2.2 Metodologia

Este estudo propõe uma avaliação do framework WTT com o conjunto de dados *Juliet Test Suite* para C/C++ versão 1.2. Diferentemente do estudo de Mouzarani e Sadeghiyan [82],

neste estudo foram consideramos todos os arquivos dos tipos de fraquezas selecionadas.

Cada arquivo C do conjunto de dados selecionado apresenta funções do tipo *bad*, que apresenta uma fraqueza, e do tipo *good*, que a fraqueza está corrigida. O Código Fonte 5.1 apresenta uma versão simplificada da função *bad* presente no arquivo CWE190_Integer_Overflow_int_fgets_add_01.c. As linhas 6 a 8 apresentam uma verificação da leitura dos dados de entrada pela função *fgets()*. A linha 10 apresenta um "problema potencial" onde é armazenado o resultado da soma de *data*, recebido como dado de entrada, com o valor 1 na variável *result*. Em seguida, na linha 11 é impresso o resultado da operação de adição.

Código Fonte 5.1: Versão simplificada da função *bad* do arquivo CWE190_Integer_Overflow_int_fgets_add_01.c

```
1  void CWE190_Integer_Overflow__int_fgets_add_01_bad(){
2      int data;
3      data = 0;
4      char inputBuffer[CHAR_ARRAY_SIZE] = "";
5      /* POTENTIAL FLAW: Read data from the console using fgets() */
6      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL){
7          data = atoi(inputBuffer);
8      }else{printf("fgets() failed.");}
9      /* POTENTIAL FLAW: Adding 1 to data could cause an overflow */
10     int result = data + 1;
11 printf(result);12}
```

O Código Fonte 5.2 apresenta uma versão simplificada da função *good* do arquivo CWE190_Integer_Overflow_int_fgets_add_01.c. Da mesma forma apresentada no código fonte anterior, as linhas 6 a 8 apresentam uma verificação da leitura dos dados de entrada pela função *fgets()*. No entanto, as linhas 10 a 15 apresentam uma verificação para prevenir a ocorrência da falha. Neste caso, a operação de adição e impressão do resultado só ocorre se a variável data for menor que o valor máximo possível para um inteiro.

Código Fonte 5.2: Versão simplificada da função *good* do arquivo CWE190_Integer_Overflow_int_fgets_add_01.c

```
1  static void goodB2G(){
2      int data;
3      data = 0;
4      char inputBuffer[CHAR_ARRAY_SIZE] = "";
5      /* POTENTIAL FLAW: Read data from the console using fgets() */
6      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL){
7          data = atoi(inputBuffer);
8      }else{printf("fgets() failed.");}
9      /* FIX: Add a check to prevent an overflow from occurring */
10     if (data < INT_MAX){
11         int result = data + 1;
12         printIntLine(result);
13     }else{
14         printf("data value is too large to perform arithmetic
               safely.");
15     }
16 }
```

Durante a execução do estudo anterior (Capítulo 4), estávamos interessados em encontrar novas fraquezas de código em programas reais, com muitas linhas de código. Desta forma, aplicamos os filtros da ferramenta de análise estática com o objetivo de chegar a um número de warnings possível de ser avaliado. Neste estudo, tratamos com exemplos pequenos de vulnerabilidades. Por este motivo, executamos a fase de análise estática de duas formas neste estudo: (i) aplicando os filtros da ferramenta Flawfinder com as *flags* que não inclui *warnings* que podem ser falsos positivos, avaliados pela ferramenta de análise estática e que define o nível de risco mínimo como 3 para inclusão na lista de ocorrências que pode ser de 0 (“nenhum risco”) a 5 (“risco máximo”) e (ii) sem os filtros da ferramenta de análise estática. Nossa intenção é avaliar se a fase de análise estática do WTT é capaz de identificar a presença das fraquezas indicadas no código.

Para execução deste estudo, foi implementada uma função dentro da fase de instrumentação do WTT com o objetivo de gerar os arquivos de teste automaticamente a partir de templates predefinidos. Esta função é específica para gerar arquivos de teste para *Juliet Test*

Suite. Neste sentido, utilizamos o *Apache FreeMarker*, uma biblioteca Java para gerar saída de texto com base em modelos e dados variáveis.

O Código Fonte 5.3 apresenta uma versão simplificada do template definido para criação dos arquivos de teste utilizados na fase ECT. A classe Java que por meio do template gera os arquivos de teste, se encarrega de substituir as variáveis definidas, por exemplo *${externVar}*. Neste exemplo, a classe Java tem a responsabilidade de definir o arquivo testado (linha 2), variáveis externas (linha 4), funções *mock* (linha 5) e função testada (linha 14). As demais funcionalidades apresentadas no template, são comuns à série de arquivos que serão testados utilizando o mesmo template. Similaridades, como por exemplo a operação de adição (linha 9), será utilizada para testar os demais arquivos da série.

Código Fonte 5.3: Versão simplificada do arquivo de template para criação de casos de teste utilizados na fase ECT

```
1  // Lines of code here
2  #include "${pathDataSet}${fileName}"
3  //Mock functions
4  ${externVar}
5  ${mockedFunctions}
6  static void test_juliet_rtc(void **state){
7      int data;
8      data = INT_MAX;
9      int result = data + 1;
10     sprintf(inputBuffer, "%d", data);
11     char buf[BUFSIZ];
12     freopen("/dev/null", "a", stdout);
13     setbuf(stdout, buf);
14     ${testedFunction}();
15     freopen("/dev/tty", "a", stdout);
16 assert_int_equal(atoi(buf),result);17   }
```

Código Fonte 5.4 apresenta uma versão simplificada do template definido para criação dos arquivos de teste utilizados na fase FT. A diferença para o template anterior é a forma

como recebe os dados de entrada (linha 9), característica da fase FT, apresentada anteriormente (Seção 3.2.3).

Código Fonte 5.4: Versão simplificada do arquivo de template para criação de casos de teste utilizados na fase FT

```
1  // Lines of code here
2  #include "${pathDataSet}${fileName}"
3  //Mock functions
4  ${externVar}
5  ${mockedFunctions}
6  static void test_juliet_rtc(void **state){
7      int data;
8      data = 0;
9      scanf("%d\n", &data);
10     int result = data + 1;
11     sprintf(inputBuffer, "%d", data);
12     char buf[BUFSIZ];
13     freopen("/dev/null", "a", stdout);
14     setbuf(stdout, buf);
15     ${testedFunction}();
16     freopen("/dev/tty", "a", stdout);
17 assert_int_equal(atoi(buf),result);18   }
```

Com o mesmo objetivo geral apresentado nos Códigos Fonte 5.1 e 5.2, ou seja, receber um dado de entrada do tipo inteiro, efetuar a operação de adição ao valor 1, armazenar em uma variável e imprimir o resultado, *Juliet Test Suite* apresenta casos em que as operações são efetuadas em funções de arquivos diferentes. Neste caso, foi necessário a utilização de funções *mocks*. Esta operação foi realizada manualmente, inserindo a implementação da função *mock* dentro da classe Java, responsável por criar os arquivos de teste. Porém, como nos arquivos que apresentam o mesmo objetivo, apresentam uma série de semelhanças, foi preciso definir apenas uma vez para execução de uma série de 50 arquivos. Para as outras 6 execuções da série, foram feitos pequenos ajustes na implementação da função *mock* como

a definição de um vetor.

O Código Fonte 5.5 apresenta uma versão simplificada da função *bad* do arquivo CWE190_Integer_Overflow_int_fgets_add_51a.c. Nas linhas 7 a 9, a função realiza uma verificação da leitura dos dados de entrada pela função *fgets()*. Na linha 10 a função realiza uma chamada de uma função que foi definida em outro arquivo, passando a variável *data*, recebida como entrada de dados.

Código Fonte 5.5: Versão simplificada da função *CWE190_Integer_Overflow__int_fgets_add_51_bad* do arquivo CWE190_Integer_Overflow_int_fgets_add_51a.c

```
1  void CWE190_Integer_Overflow__int_fgets_add_51b_badSink(int data);
2  void CWE190_Integer_Overflow__int_fgets_add_51_bad(){
3      int data;
4      data = 0;
5      char inputBuffer[CHAR_ARRAY_SIZE] = "";
6      /* POTENTIAL FLAW: Read data from the console using fgets() */
7      if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL){
8          data = atoi(inputBuffer);
9      }else{printf("fgets() failed.");}
10 CWE190_Integer_Overflow__int_fgets_add_51b_badSink(data);11
   }
```

O Código Fonte 5.6 apresenta a versão simplificada da função definida no arquivo CWE190_Integer_Overflow_int_fgets_add_51b.c, que recebe a variável *data*. Esta função se encarrega de realizar a operação de adição (linha 3) e imprimir o resultado (linha 4). Com o objetivo de testar o arquivo CWE190_Integer_Overflow_int_fgets_add_51a.c, a implementação desta função foi levada para o arquivo de teste pela função *mock*.

Código Fonte 5.6: Função *CWE190_Integer_Overflow_int_fgets_add_51b_badSink* do arquivo CWE190_Integer_Overflow_int_fgets_add_51b.c

```
1  void CWE190_Integer_Overflow__int_fgets_add_51b_badSink(int data){
2      /* POTENTIAL FLAW: Adding 1 to data could cause an overflow */
3      int result = data + 1;
4      printf(result);
```

```
5 }
```

Juliet Test Suite apresenta alguns arquivos apenas para referenciar funções em arquivos diferentes. A Figura 5.1 apresenta um exemplo deste tipo de arquivo. Este tipo de arquivo foi descartado do estudo (*Arquivo B*).

Arquivo A

```
funcA(){
   int data;
   fgets(data, ...);
   funcB(data);
}
```

Arquivo B

```
funcB(int data){
   funcC(data);
}
```

Arquivo C

```
funcC(int data){
   int result = data + 1;
   printf(result);
}
```

Figura 5.1: Exemplo simplificado de arquivos descartados neste estudo.

5.2.3 Ambiente de Execução

Executamos o estudo no sistema operacional Ubuntu 20.04.2 LTS (2.9 GHz, i5 e 8GB RAM). Utilizamos o Flawfinder 1.31 como ferramenta de análise estática. Para instrumentar o código, gerar casos de teste e executar *fuzz testing*, utilizamos o GCC 5.4.0 (Ubuntu 9.3.0-17ubuntu1 20.04), cmocka 1.1.3 e o American Fuzzy Lop (2.52b), respectivamente. Neste estudo, definimos o tempo de 20 segundos para execução do *fuzz testing*. Utilizamos o Apache FreeMarker versão 2.3.31 como *engine* para definição de templates para criação dos casos de teste.

5.3 Resultados

A seguir, descrevemos os principais resultados que fundamentam as respostas às questões de pesquisa definidas para este estudo. A versão do WTT utilizada [1] e os arquivos gerados durante a execução do estudo [2] estão disponíveis online.

Neste estudo, avaliamos um total de 5.295 arquivos de código fonte C do conjunto de dados *Juliet Test Suite* com fraquezas dos tipos: *CWE-190: Integer Overflow or Wraparound*,

[1] https://github.com/raphaeldecm/wtt_juliet
[2] https://github.com/raphaeldecm/wtt_juliet_catalog

CWE-191: Integer Underflow (Wrap or Wraparound), *CWE-369: Divide By Zero*. Deste total de arquivos, foram descartados 3.878 arquivos por não utilizarem a forma de entrada de dados adotada neste estudo: funções de entrada de dados *fscanf* ou *fgets* e entrada de dados por meio da leitura de arquivos. Restando um total de 1.417 arquivos selecionados. Para cada arquivo selecionado, testamos duas funções separadamente: *bad* e *good*. Desta forma, testamos um total de 2.834 funções. Para cada função testada, foram criados dois arquivos de teste utilizados nas fases FT e ECT do WTT.

A Tabela 5.2 apresenta o quantitativo dos arquivos selecionados para cada tipo de CWE. Considerando que no estudo apresentado por Mouzarani e Sadeghiyan [82] foram realizadas alterações nos arquivos do tipo *CWE-476: NULL Pointer Dereference* para que pudessem ser testados, optamos por descartar os arquivos deste tipo de CWE pois esta é uma prática que não adotamos na execução do WTT.

Tabela 5.2: Visão geral dos arquivos selecionados para execução do estudo.

Fraqueza	**Arquivos C**	**Descartados**	**Testados**
CWE-190	3.029	2.210	819
CWE-191	1.542	1.232	310
CWE-369	352	64	288
CWE-476	372	372	0
Total	**5.295**	**3.878**	**1.417**

A Tabela 5.3 apresenta uma visão geral dos nossos resultados para cada fase do WTT por tipo de CWE. Todas as funções do tipo *good* foram classificadas como *não detectado* e todas as funções do tipo *bad* foram classificadas como *detectado* na fase ECT (execução de casos de teste) do WTT, onde o testador define os dados de teste. Do total de funções *bad* com fraquezas do tipo CWE-191 (310), 10,96% foram classificadas como *não detectadas* na fase AFL ECT (Execução de casos de teste com entradas geradas pelo AFL). No entanto, o resultado do WTT depende do resultado de todas as fases. Como estas funções foram classificadas como *detectado* na fase ECT, consideramos que o WTT conseguiu detectar a fraqueza. Na fase FT (*fuzz testing*), WTT não identificou entradas que pudessem causar *crash*.

Tabela 5.3: Resultado das funções testadas para cada tipo de CWE.

			ECT		AFL ECT		FT
Fraqueza	Tipo	Funções Testadas	Detectado	Não Detectado	Detectado	Não Detectado	Detectado
CWE-190	bad	819	819	0	819	0	0
	good	819	0	819	0	819	0
CWE-191	bad	310	310	0	276	34	0
	good	310	0	310	0	310	0
CWE-369	bad	288	288	0	288	0	0
	good	288	0	288	0	288	0
Total		2834	1417	1417	1383	1451	0

A Tabela 5.4 apresenta o número de arquivos que WTT identificou fraquezas dentro das funções testadas (*bad e good*). A proporção por tipo de função se manteve a mesma, ou seja, sempre que a análise estática identificou um *warning* dentro de uma função do tipo *bad*, também foi identificado na função *good* no mesmo arquivo.

Neste estudo avaliamos se o WTT, durante a execução da fase de análise estática, seria capaz de indicar o código suspeito dentro das funções testadas. Nossos resultados indicam que utilizando os filtros da ferramenta de análise estática, WTT não identificou fraquezas dentro das funções testadas. Já considerando todos os possíveis *warnings* (sem filtros), WTT foi capaz de identificar fraquezas dentro das funções em 280 dos arquivos testados.

Tabela 5.4: Número de arquivos com fraquezas dentro das funções testadas por tipo de CWE.

	Sem Filtro	
Fraqueza	**Identificado**	**Não identificado**
CWE-190	105	714
CWE-191	68	242
CWE-369	107	181
Total	280	1137

Com o objetivo de avaliar a acurácia da execução do WTT no conjunto de dados *Juliet*

Test Suite, calculamos métricas de avaliação, como *precision*, *recall* e *f1 score*. A Tabela 5.5 resume as métricas para cada tipo de fraqueza.

Tabela 5.5: Métricas por tipo de fraqueza: Verdadeiro Positivo (VP); Falso Positivo (FP); Verdadeiro Negativo (VN); Falso Negativo (FN); *Precision* (P),*Recall* (R) e *F1 score*.

Fraqueza	**VP**	**FP**	**VN**	**FN**	**P**	**R**	**F1**
CWE-190	819	0	819	0	1.00	1.00	1.00
CWE-191	310	0	310	0	1.00	1.00	1.00
CWE-369	288	0	288	0	1.00	1.00	1.00
Total	1.417	0	1.417	0	1.00	1.00	1.00

Após a apresentação dos resultados relativos às fraquezas de código analisadas, é possível responder as questões de pesquisa da seguinte forma.

- QP_1**:** Qual a taxa de detecção do WTT para cada tipo de fraqueza testada?
  Foram testadas o total de 2.834 funções *bad* e *good*. Deste total, 57,79% são fraquezas do tipo CWE-190, 21,88% do tipo CWE-191 e 20,33% do tipo CWE-369. WTT classificou como detectadas todas as funções rotuladas como *bad* e classificou como não detectadas todas as funções rotuladas como *good*.

- QP_2**:** Qual a taxa de fraquezas encontradas dentro das funções testadas na fase de análise estática?
  Considerando a execução da fase de análise estática sem a aplicação dos filtros, foram identificadas fraquezas dentro das funções em 19.76% dos 1.417 arquivos avaliados.

- QP_3**:** Qual a taxa de *precision*, *recall* e *f1* o WTT apresentou?
  Como apresentado na Tabela 5.5, nossos resultados apresentaram o valor 1 para as métricas de *precision*, *recall* e *f1 score*, desconsiderando a fase de análise estática, já que não foi possível detectar 100% dos casos

5.4 Discussão

Nesta seção, discutimos com mais detalhes os resultados da execução do WTT em um conjunto de arquivos com fraquezas confirmadas.

Os dados selecionados para execução deste estudo apresentam 61.387 arquivos com exemplos de 118 diferentes CWEs. Em cada arquivo contém funções *bad* e *good*. Neste estudo, executamos o WTT em funções *bad* e *good* de um total de 1.417 arquivos C do conjunto de dados *Juliet Test Suite*. Como é possível verificar na Tabela 5.3, considerando cada fase, WTT detectou fraquezas em 100% das funções na fase ECT e 97,6% das fraquezas na fase AFL FT. WTT não detectou 34 casos do total de 310 do tipo CWE-191 na fase AFL FT.

Mouzarani e Sadeghiyan [82] apresentaram um estudo comparativo sobre detecção de fraquezas entre o *smart fuzzer CATCHCONV* e o método proposto por eles. *CATCHCONV* utiliza *concolic execution* para detectar *bugs* relacionados a inteiros em códigos executáveis. [79]. No estudo apresentado, foi feita a comparação apenas com os arquivos do tipo CWE-190 onde, de acordo com Mouzarani e Sadeghiyan [82], não foi possível detectar vulnerabilidade nos programas que manipulam o tipo de dados *char* com o CATCHCONV, atingindo 0% de *precision* para este grupo de arquivos.

Observação 1: *Considerando as fases de execução dos casos de teste, WTT detectou 100% dos casos para os tipos de fraquezas abordadas no estudo.*

Mouzarani e Sadeghiyan [82] apresentaram informações limitadas sobre os resultados obtidos. Além disso, eles não disponibilizaram os detalhes da execução de modo que nos permitisse replicar o estudo apresentado. Em contato com os autores, solicitamos os dados do estudo. No entanto, eles nos passaram apenas outros artigos publicados sobre o método proposto por eles [83; 84; 85; 86].

Fizemos o levantamento do número de arquivos nos quais a fase de análise estática conseguiu detectar fraquezas dentro das funções. Desta forma, identificamos que sempre que a ferramenta de análise estática reportou um *warning* em uma função *bad*, também reportou um *warning* na função *good*. Considerando o número total de arquivos avaliados no estudo (1.417), WTT reportou indícios de fraquezas na fase de análise estática em 19,76% dos casos.

Como estamos avaliando um conjunto de dados com fraquezas confirmadas, este resultado nos revela, mais uma vez, a necessidade de melhoria das ferramentas de análise estática na identificação de indícios de fraquezas em código C. Esta fase é fundamental para o WTT,

já que é por meio dela que direcionamos a execução dos casos de teste para partes específicas do código fonte.

Observação 2: *A fase de análise estática apresentou baixa taxa de detecção.*

Durante os testes iniciais, observamos que o WTT não identificou entradas que pudessem causar *crash* à função testada para as fraquezas consideradas no estudo. Como é possível perceber nos Código Fonte 5.1, 5.2, 5.5 e 5.6 as funções testadas apresentam poucas linhas de código, objetivos bem definidos (coesão) e poucas dependências (acoplamento). Acreditamos que estes são alguns dos fatores que impactaram na não ocorrência de *crash* durante a execução da fase FT. Neste sentido, definimos o tempo de 20 segundos para execução do *fuzz testing*, o que gerou uma média de 713 *inputs*. É possível que, considerando um maior tempo para execução do *fuzz testing*, WTT seja capaz de identificar entradas que causem *crash*. No entanto, devido ao grande número de funções testadas e as demais fases do WTT detectarem 100% das fraquezas avaliadas, não consideramos este um fator determinante para execução do estudo.

Observação 3: *WTT não identificou entradas que pudessem causar crash.*

A versão atual do WTT implementado para testar o conjunto de dados *Juliet Test Suite*, com exceção da criação do *templates*, apresenta todas as suas fases automáticas. Neste sentido, contabilizamos o tempo de execução do WTT a partir da criação dos casos de teste baseado nos *templates* previamente elaborados. Este processo foi automatizado dentro do processo de execução do *framework*. A Tabela 5.6 apresenta o tempo média de execução do WTT para cada tipo de fraqueza. Considerando o somatório de todas as fases automáticas, o tempo médio de execução do WTT é de 22.58s para arquivos do tipo CWE-190, 23.95s para arquivos do tipo CWE-191 e 22.40s para arquivos do tipo CWE-369.

Observação 4: *Considerando as fases automáticas, WTT demorou uma média de 0.23s a mais para testar funções do tipo bad.*

Tabela 5.6: Média de tempo gasto em segundos na execução do WTT por tipo de função testada (bad ou good) e tipo de CWE.

	Tempo (s)	
Fraqueza	**Bad**	**Good**
CWE190	22.69	22.47
CWE191	24.09	23.81
CWE369	22.50	22.32
Média	**23.10**	**22.87**

Além disso, fizemos a contabilização manual do tempo entre a configuração do *framework* e o início da execução para o melhor e pior caso (mais simples e mais complexo). Para o caso mais simples, consideramos o arquivo CWE190_Integer_Overflow__int_fgets_add_01.c onde é recebida uma entrada utilizando a função *fgets*, armazena a entrada somando ao valor 1 e imprimindo o resultado. Para este caso, foram gastos 81,26s. Para o caso mais complexo, consideramos o arquivo CWE190_Integer_Overflow__int_fgets_add_67a.c onde é recebida uma entrada utilizando a função *fgets*, armazena a entrada em um *struct* e passa o *struct* para uma função *mock*, que adiciona 1 ao valor e imprime o resultado. Para este caso, foram gastos 111,34s.

5.5 Ameaças à Validade

Nesta seção, descrevemos as principais ameaças à validade do estudo.

A utilização do conjunto de dados *Juliet Test Suite* apresenta uma série de vantagens, como muitos exemplos de fraquezas de diferentes tipos. Contudo, também apresenta algumas limitações. Como ameaça à validade externa consideramos que as funções apresentadas pelo conjunto de dados selecionados são bastante pequenas e contém uma única fraqueza. Neste sentido, as funções não apresentam realismo encontrado em funções de programas reais. Porém, no estudo anterior, apresentado no Capítulo 4, podemos avaliar o comportamento do WTT em programas reais.

Quanto às ameaças à validade interna, fizemos uma análise inicial do código em teste e, baseados em nossos conhecimentos, construímos manualmente os templates para criação au-

tomática dos casos de teste. O autor desta tese conduziu este processo manual, consultando os orientadores quando necessário. No entanto, sabemos que o sucesso destes processos manuais depende diretamente de fatores, como nossa expertise. Neste sentido, inspecionamos manualmente os templates e o código em teste, com o objetivo de verificar alguma prática de programação inadequada.

Em relação à validade da conclusão, definimos o tempo de execução do AFL de 20 segundos para cada execução. Em virtude do número de entradas geradas, a definição diferente de tempo de execução pode afetar no resultado. Este fator pode ter impactado na não detecção de *crash* na execução da fase FT do WTT. No entanto, WTT detectou 100% das fraquezas nas demais fases.

5.6 Considerações Finais

Neste capítulo, apresentamos um estudo empírico com o objetivo de avaliar a técnica proposta com o conjunto de dados *Juliet Test Suite* versão 1.2. Para execução deste estudo, foi implementada uma funcionalidade dentro da fase de instrumentação do WTT com o objetivo de gerar os arquivos de teste automaticamente. Esta funcionalidade se aplica apenas para o conjunto de dados utilizados neste estudo.

Por meio do estudo executado, foi observado que (i) do total de 2.834 funções *bad* e *good* testadas, 57,79% correspondem à CWE-190, 21,88% à CWE-191 e 20,33% correspondem à CWE-369 e o WTT classificou como detectado todas as funções *bad* e classificou como não detectado todas as funções *good*. (ii) Desta forma, WTT atingiu o valor 1 para as métricas de *precision*, *recall* e *f1 score*. (iii) A fase de análise estática do WTT identificou fraquezas em apenas 19,76% das funções testadas.

Capítulo 6

Estado da Arte

Neste capítulo, apresentamos o planejamento e execução de uma revisão sistemática da literatura que apresenta como objetivo identificar e discutir o estado da arte no contexto de detecção de fraquezas em programas C (Seção 6.1). Além disso, relacionamos nosso trabalho a uma série de abordagens (Seção 6.2).

6.1 Revisão Sistemática da Literatura

Nos primeiros estágios desta pesquisa, estávamos interessados no contexto de sistemas configuráveis. Por este motivo, esta revisão sistemática apresenta objetivo relacionado ao contexto de sistemas configuráveis. No entanto, com a evolução da pesquisa foi decidido considerar todos os sistemas desenvolvidos em linguagem C. Com o objetivo de mitigar este problema, executamos *snowballing* e consultas frequentes às bases de dados para cobrir trabalhos que não foram selecionados na revisão sistemática por estarem fora de seu escopo ou por terem sido publicados em uma data posterior à conclusão deste estudo. Esta seção está dividida em: Subseção 6.1.1 descreve o planejamento e execução da revisão sistemática, a Subseção 6.1.2 apresenta os resultados e a Subseção 6.1.3 apresenta análises e discussões sobre os resultados, a Subseção 6.2 apresenta uma revisão dos principais trabalhos que discutem conceitos relacionados à esta pesquisa. Finalmente, a Subseção 6.3 apresenta as considerações finais do capítulo.

6.1.1 Método de Pesquisa

Este estudo foi baseado nas diretrizes de Kitchenham e Charters [63] para realizar revisões sistemáticas em engenharia de software. O objetivo principal deste estudo consiste em analisar estratégias de teste de software que apresentam o propósito de detectar fraquezas em programas desenvolvidos em linguagem C do ponto de vista de desenvolvedores no contexto de sistemas configuráveis. Neste sentido, foram endereçadas as seguintes questões de pesquisa:

- RQ_1**:** Quais técnicas de teste são capazes de detectar fraquezas em sistemas configuráveis?
- RQ_2**:** Quais critérios de teste foram aplicados neste contexto?
- RQ_3**:** Técnicas de teste de software podem detectar quais tipos de fraquezas?

Primeiramente, executamos um estudo exploratório com o objetivo de selecionar trabalhos norteadores para a condução da revisão. Este estudo inicial é fundamental para definição de elementos como bases de dados, principais termos e sinônimos, utilizados na construção da *string* de busca. De cada estudo selecionado, identificamos e extraímos informações sobre as técnicas e critérios aplicados e/ou propostos pelos autores, bem como as fraquezas que foram abordadas em cada trabalho. A Figura 6.1 apresenta uma visão geral do processo seguido. Os trabalhos utilizados no estudo exploratório podem ser encontrados na Tabela 6.1.

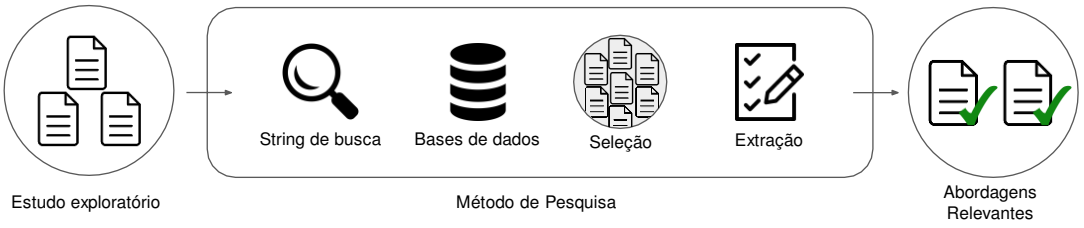

Figura 6.1: Visão geral do processo adotado na execução da revisão sistemática da literatura.

Seguimos os seguintes passos para a execução da revisão sistemática:

a) Derivar os principais termos para população, intervenção e resultado;

b) Identificar ortografia alternativa e sinônimos para os principais termos;

Tabela 6.1: Trabalhos selecionados no estudo exploratório.

Estudo	Título
Spencer and Collyer [99]	*#ifdefs* considered harmful, or portability experience with C news
Ferreira et al. [34]	Do *#ifdefs* influence the occurrence of vulnerabilities? an empirical study of the linux kernel
Godefroid et al. [44]	Automated Whitebox Fuzz Testing
Moraes et al. [80]	A family of test selection criteria for Timed Input-Output Symbolic Transition System models
Pfleeger et al. [94]	Improving Memory Management Security for C and C++
NVD [5]	National Vulnerability Database
CWE [20]	Common Weakness Enumeration
CVE [4]	Common Vulnerabilities and Exposures

c) Identificar a base de dados para realizar a revisão sistemática;

d) Construir uma *string* de busca para cada base de dados;

e) Executar a pesquisa;

f) Executar *snowballing* (analisar documentos referenciados pelos artigos recuperados e os artigos que fazem referência aos documentos encontrados na revisão). Para *forward snowballing*, selecionamos artigos críticos conforme sugerido por Kitchenham e Brereton [63] com base no número de citações e relevância para nosso estudo.

Em relação ao item **a)**, os termos de pesquisa são derivados da população, intervenção e resultados. Para a execução deste estudo, consideramos a população como sendo os desenvolvedores que utilizam técnicas para detectar fraquezas no código-fonte. A intervenção são as técnicas para detectar fraquezas em sistemas configuráveis e os resultados são os tipos de fraquezas detectadas por cada técnica.

Em relação ao item **b)**, utilizamos os trabalhos selecionados durante a execução do estudo exploratório. A Tabela 6.2 apresenta os termos comuns e sinônimos identificados.

Tabela 6.2: Sinônimos e ortografia alternativa

Termos	**Sinônimos e ortografia alternativa.**
Testing	Software testing, test select criteria, testing strategies
Variability	Configurable systems, software product lines, variability-aware, macros, *#ifdefs*, configurations
Vulnerability	Weakness, security, exploit, explore, cyber attack, attacker

No item **c)**, também utilizamos os artigos identificados durante o estudo exploratório com o objetivo de selecionar as principais bases de dados. Conforme sugerido na literatura [63], escolhemos ACM, IEEE, Google Academic, Engineering Village e Scopus como bases de dados. A tabela 6.3 apresenta as bases de dados selecionadas, o idioma, a estratégia de seleção dos estudos, responsáveis pela avaliação e período de publicação selecionado.

Tabela 6.3: Bases de Dados e Critérios Considerados na Execução da Revisão Sistemática.

Critério	**Valor**
Idioma	Inglês
Método de Busca	O artigo deve estar disponível online. Rastreamos os artigos que não disponíveis online em uma lista WIP.
Bases de Dados	IEEE Explore ACM Digital Library Google Academic Engineering Village Scopus
SLR avaliação	Autores
Período	Trabalhos publicados entre os anos de 2008 e 2018.

String de Busca

A construção da *string* de busca depende de cada base de dados selecionada. Cada base de dados permite buscar atributos específicos dos artigos, como título e resumo, mas a sintaxe para expressar a busca é diferente. A *string* de busca padrão para nossas questões de pesquisa é a seguinte:

```
(''test'') and
(''variability'' OR ''configurable systems'' OR
  ''software product lines'' OR
  ''variability-aware'' OR ''macros'' OR
  ''ifdefs'' OR ''configurations'') and
(''vulnerability'' OR ''weakness'' OR
  ''security'' OR ''exploit'' OR ''explore'' OR
  ''cyber attack'' OR ''attacker'')
```

Para cada base de dados, realizamos a busca com base no título e no resumo. Se a base de dados não permitir essas características, realizamos a busca com base no texto completo (se permitido) ou na busca padrão da base de dados. A tabela 6.4 apresenta uma versão simplificada das strings de buscas aplicadas em cada base de dados.

Critérios de Inclusão e Exclusão

Durante a execução desta revisão sistemática, nós estávamos interessados em trabalhos relacionados ao domínio da engenharia de software que respondessem às nossas questões de pesquisa. A Tabela 6.5 apresenta os critérios de inclusão e exclusão para nossa revisão sistemática. Em particular, estávamos interessados em estudos de teste em sistemas configuráveis com o objetivo de detecção de fraquezas.

Metodologia de Seleção

Após a execução da busca nas bases de dados selecionadas, importamos a lista de trabalhos retornados para a ferramenta *StArt*, uma ferramenta de apoio à condução de revisões sistemáticas [1].

Tabela 6.4: *String* de busca por base de dados.

Base de Dados	String de Busca	Artigos
ACM	(+test) AND (+variability+"configurable systems" +macros+ifdefs+configurations) AND (+vulnerability+weakness+security+exploit)	2,337
IEEE	(test) AND (variability OR "configurable systems" OR macros OR ifdefs OR configurations) AND (vulnerability OR weakness OR security OR exploit)	201
Engineering Village	(test) AND (variability OR "configurable systems"OR macros OR ifdefs OR configurations) AND (vulnerability OR weakness OR security OR exploit)	447
SCOPUS	(test) AND (variability OR "configurable systems"OR macros OR ifdefs OR configurations) AND (vulnerability OR weakness OR security OR exploit)	801
Google Academic	variability OR "configurable systems"OR macros OR ifdefs OR configurations OR vulnerability OR weakness OR security OR exploit "software testing"	83
	Snowballing	20
	Total	**3,675**

Na fase de seleção, os artigos foram rotulados como não classificados (aguardando classificação em aceitos ou rejeitados), duplicados (os trabalhos que apareceram mais de uma vez na busca) e aceitos ou rejeitados, de acordo com os critérios de inclusão e exclusão. A ferramenta *StArt* possui um recurso de classificação automática para identificar documentos duplicados.

Para classificar cada artigo recuperado como aceito ou rejeitado, avaliamos seu título e resumo. Se o artigo se enquadrar nos critérios de exclusão, ele é classificado como rejeitado. Se o artigo se enquadrar nos critérios de inclusão, ele é classificado como aceito. Após esta classificação inicial, selecionamos apenas os artigos classificados como aceitos para serem lidos na íntegra. Após a leitura completa, classificamos novamente os artigos de acordo com

Tabela 6.5: Critérios de Inclusão e Exclusão.

Critério	Descrição
Inclusão	a) O trabalho responde a uma das questões de pesquisa. b) O trabalho trata uma técnica de teste no contexto da linguagem C. c) O estudo tem um nível de qualidade superior a 0,5.
Exclusão	a) O artigo não trata de teste de software. b) O artigo é externo à engenharia de software. c) O documento não faz parte de uma conferência ou periódico. d) O documento não está disponível online. e) O artigo é anterior a 2008.
Tipos de Estudo	Estudos de caso, estudos empíricos e estudos secundários
Estudos duplicados	Selecionar a versão mais recente

os critérios de inclusão e exclusão.

Extração de Dados

Na fase de extração de dados, procuramos determinar formas de coletar informações dos artigos selecionados para registrar informações relevantes identificadas nos artigos selecionados.

Para cada estudo aceito em nossa revisão sistemática, coletamos informações gerais, como: tipo (jornal, conferência), título, autor(es), volume, problema, páginas e ano. Para extrair essas informações de cada estudo, utilizamos a ferramenta *Mendeley* [1], uma ferramenta de apoio à organização de referências. A partir de estudos empíricos, além dos dados padronizados como título, autor e conferência, coletamos dados para responder às nossas questões de pesquisa, como:

[1]https://www.mendeley.com/

1. **Características**: Características da estratégia de teste utilizada;
2. **Tipo de teste de caixa branca**: Tipo de estratégia de caixa branca (se aplicável);
3. **Tipo de teste de caixa preta**: Tipo de estratégia de caixa preta (se aplicável);
4. **Ferramentas utilizadas**: Ferramentas utilizadas no estudo;
5. **Grupo de pesquisa**: Grupos de pesquisa que executou o trabalho;
6. **Sujeito**: Sujeitos avaliados no estudo;
7. **Tamanho dos dados**: Tamanho dos dados avaliados no estudo;
8. **Tipo de estudo empírico**: Tipo de estudo realizado (se aplicável);
9. **Metodologia**: Metodologia aplicada no estudo;
10. **Tipos de fraquezas**: Tipos de fraquezas abordados no estudo;
11. **Número de fraquezas**: Número de fraquezas detectados por uma técnica de teste;
12. **Linguagem**: Linguagem de programação utilizada (por exemplo, Java, C#, C);
13. **Critérios de teste**: Critérios de teste utilizados para definir quando parar o teste;
14. **Entrada**: Entradas utilizadas na estratégia.
15. **Saída**: Saídas produzidas pelas estratégias.

Se o estudo analisado apresentar algum dado considerado relevante ausente, definimos o valor NP. Além disso, se algum dado não puder ser identificado, definimos o valor NA.

Ao final da classificação dos trabalhos retornados, foram classificados 31 artigos relacionados às nossas questões de pesquisa. A Figura 6.2 apresenta a quantidade de artigos excluídos em cada fase de nossa revisão. Como pode ser visto, três filtros foram aplicados no processo. O primeiro, que corresponde aos artigos duplicados, excluiu 325 artigos.

Na fase de seleção, onde selecionamos os artigos baseados nos critérios de inclusão e exclusão, por meio da leitura do título e resumo, excluímos 3.236 artigos. Por fim, na fase de extração, por meio da leitura completa dos artigos foram excluídos 83 artigos, restando ao final 31 artigos aceitos.

Figura 6.2: Processo de classificação dos artigos retornados na revisão sistemática.

Avaliação de Qualidade

Após o processo de extração de dados, definimos a avaliação da qualidade de cada um dos artigos selecionados seguindo o processo de avaliação de qualidade apresentado por Kitchenham e Charters [64]. Selecionamos um conjunto de características com o objetivo de definir a qualidade de cada trabalho recuperado. A avaliação da qualidade dos artigos selecionados também foi considerada no processo de aceitação ou rejeição dos artigos. Para aumentar a confiabilidade da seleção, esse processo foi conduzido pelo primeiro autor e revisado pelos demais autores do estudo. Nesta fase, excluímos apenas os artigos que têm um estudo pouco claro ou ambíguo ou não é um trabalho completo. Para os outros critérios utilizados, aplicamos valores que resultam na avaliação da pontuação de qualidade. A Tabela 6.6 apresenta os critérios de qualidade. Não encontramos trabalhos que propusessem questionários.

6.1.2 Resultados

Nesta seção, objetivamos responder às questões de pesquisa. Na busca realizada de acordo com o protocolo definido, foram recuperados artigos relevantes entre os anos 2008 e 2018. No entanto, documentos considerados relevantes publicados anterior ao período definidos foram adicionados ao processo manualmente por *snowballing*. Foram encontrados 20 artigos candidatos por meio de *snowballing* nas referências dos artigos inicialmente selecionados. Depois de avaliar cada artigo, concordamos que quatorze deles deveriam ser incluídos. Ao final de nossa revisão sistemática, selecionamos 31 artigos, dos quais 24 foram classificados como artigos de conferência, seis como artigos de periódicos e um como capítulo de livro. A Tabela 6.7 apresenta uma síntese dos artigos aceitos em nossa revisão sistemática da literatura.

Tabela 6.6: Critérios para avaliação da qualidade dos trabalhos selecionados.

Item	Critério de Avaliação	Pontuação
01	Claro, não ambíguo	Sim = 1, Não = Descartar
02	Trabalho Completo	Sim = 1, Não = Descartar
03	O estudo é realizado para proposição de estratégias de teste de software?	Sim = 1, Não = 0
04	O estudo considera muitos tipos de fraquezas?	>=10 (1), <=7 (.7), <=3 (.3), 0 (0)
05	É possível replicar o estudo?	Sim = 1, Não = 0
06	Para um questionário, quantos responderam?	>30 (1), 10 <= N <= 30 (.5), <10 (0)
07	O artigo está bem/apropriadamente referenciado?	Sim = 1, Moderado = .5, Não = 0

QP_1: Quais técnicas de teste são capazes de detectar fraquezas em sistemas configuráveis?

Com o objetivo de responder a primeira questão de pesquisa, analisamos cada artigo aprovado, verificamos quais técnicas de teste foram propostas ou utilizadas. Verificamos que a maioria dos artigos aceitos (81,25%) discutiram algum tipo de técnica de teste. Classificamos estes de acordo com a abordagem seguida, sendo 75,86% como teste de caixa branca e 24,14% como teste de caixa preta.

Avaliamos o nível de teste abordado em cada artigo aceito. Foi verificado que seis artigos abordaram o teste de integração, três abordaram teste de sistema e vinte abordaram testes de unidade. A Figura 6.3 apresenta esta classificação. Nesta classificação não foram considerados trabalho que não propõem técnicas de teste, como as revisões sistemáticas propostas por Ahmed et al. [7] e Khatibsyarbini et al. [60].

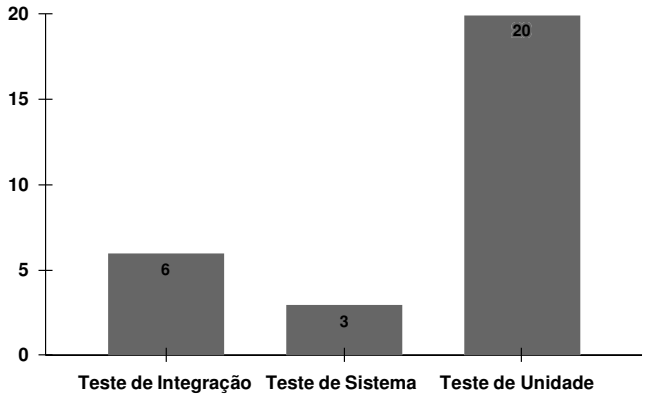

Figura 6.3: Classificação dos trabalhos aceitos quanto ao nível de teste.

Os artigos aceitos na revisão foram classificados também quanto às estratégias de teste abordadas. A Figure 6.4 apresenta uma visão geral desta classificação. Estratégias que não conseguimos identificar, foram classificadas como "*others*".

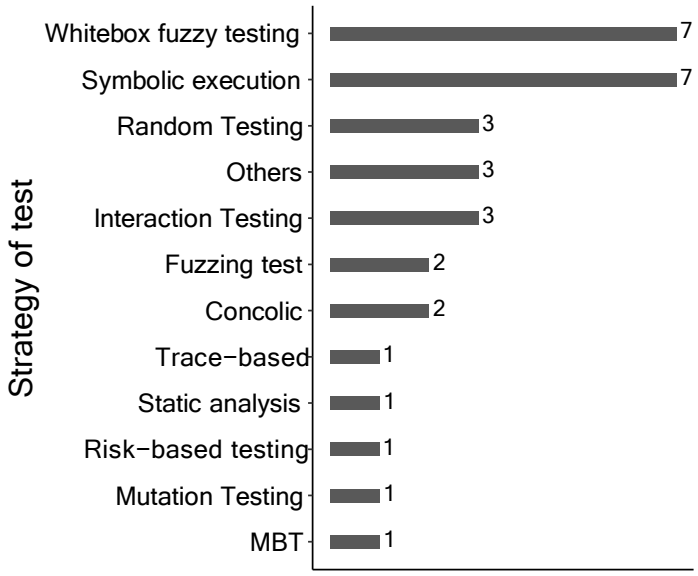

Figura 6.4: Estratégias de teste abordadas pelos trabalhos selecionados.

QP$_2$: Quais critérios de teste foram aplicados neste contexto?

Para responder a **QP$_2$**, analisamos cada técnica de teste com relação à identificação dos critérios de teste aplicados. Chakrabarti e Godefroid [15], utilizaram a detecção de uma falta ou o número de testes executados como critérios de teste. Nguyen et al. [91] utilizou o tempo

como critério de teste. O tempo limite ocorre quando o tempo de execução é superior a 400 segundos.

Cohen, Dwyer e Shi [23] estudaram os desafios fundamentais do tratamento de restrições em métodos de Teste de Interação Combinatória (CIT). Eles usaram como critério de parada a cobertura de todos os conjuntos-t necessários na amostra CIT. Zhang et al. [117], verificam se não há mais conjunções de PC dependentes de restrições de segurança.

Nossa revisão sistemática nos leva a afirmar que os desenvolvedores não aplicam um padrão de métricas para determinar se o sistema já foi suficientemente testado. Esta é uma lacuna na literatura que pode ser atacada no futuro.

QP_3: Técnicas de teste de software podem detectar quais tipos de fraquezas?

Finalmente, para responder QP_3, identificamos cada tipo de fraqueza detectada pelas abordagens recuperadas em nossa revisão sistemática. Mais de 40% dos artigos selecionados discutem algum tipo de fraqueza. A maioria deles é classificada como *CWE-120: Buffer Copy without Checking Size of Input (Classic Buffer Overflow) pelo Common Weaknesses Enumeration*. Além disso, os artigos selecionados discutem outros tipos de pontos fracos, como *CWE-401: Missing Release of Memory after Effective Lifetime, CWE-134: Use of Externally-Controlled Format String, CWE-190: Integer Overflow or Wraparound e CWE-369: Divide By Zero.*

A Tabela 6.8 apresenta uma relação entre o tipo de testes e os trabalhos que discutem sobre alguns tipos de fraquezas. De todos os trabalhos que discutem fraquezas, a maioria (92,3%) utilizou teste unitário para detectá-las.

6.1.3 Análises e Discussões

Garantir a qualidade e a segurança do software é um desafio devido ao custo de testar um programa. Este cenário se torna ainda pior quando se trata de sistemas configuráveis. Por meio da execução da revisão sistemática, não foi possível identificar técnicas de teste aplicadas no contexto de sistemas configuráveis. No entanto, identificamos trabalhos que trataram sobre técnicas de teste no contexto da linguagem C que, potencialmente, poderiam ser discutidos. Nossos resultados indicam que autores, como Godefroid et al. [45] e Zhang et al. [117], tem

usado estratégias de teste com relação à detecção de fraquezas em programas C.

Alguns trabalhos [59; 56; 67; 76] se preocuparam em propor uma abordagem para gerar as configurações adequadas para detectar fraquezas em sistemas configuráveis. Porém, para verificar a presença de fraquezas em cada configuração, eles utilizaram a análise estática, que gera muitos alarmes falsos. No entanto, de acordo com Medeitos et al. [77] ferramentas de análise estática consideram apenas uma configuração por vez. Desta forma, essas ferramentas não focam em problemas relacionados ao uso do pré-processador.

Alguns trabalhos recuperados em nossa revisão sistemática [41; 43], apresentaram bons resultados aplicando a análise estática e o teste de software juntos. Nesse sentido, eles utilizaram ferramentas de análise estática no que diz respeito à detecção de algumas fraquezas e, em seguida, usaram as técnicas de teste para expor as fraquezas identificadas pela análise estática.

Nossos resultados mostraram que o teste de unidade é a técnica de teste mais utilizada para detectar fraquezas nos trabalhos recuperados em nossa revisão sistemática. A técnica de teste de unidade foi usada para detectar fraquezas, como CWE-120: Buffer Overflow, CWE-369: Divide By Zero, CWE-134: String Errors, CWE-190: Integer Overflow e CWE-457: Use of Uninitialized Variable. Além disso, o CWE-120, CWE-134 e CWE-190 foram classificados como o 3º, 23º e 24º erro de software mais perigoso, respectivamente [19].

6.1.4 Ameaças à Validade

Assim como em outras revisões sistemáticas, esta apresenta algumas ameaças potenciais associadas à coleção de estudos e sua avaliação imprecisa. Estas ameaças foram reduzidas seguindo diretrizes bem conhecidas sobre a realização de estudos de literatura.

Os autores não podem garantir a cobertura de 100% dos artigos que respondem às nossas perguntas de pesquisa. No entanto, essa ameaça é diminuída por pesquisas estruturadas nas bases de dados. Além disso, realizamos uma busca *snowballing* para garantir a máxima cobertura possível.

Outra ameaça significativa presente nas revisões sistemáticas ocorre quando apenas um autor avalia os artigos recuperados no processo de extração. Para evitar o viés de um único autor, nosso processo de extração de dados foi verificado por todos os autores e solicitamos a revisão por pelo menos dois autores.

6.2 Trabalhos Relacionados

Esta seção apresenta uma revisão de trabalhos considerados relevantes que discutem conceitos relacionados à pesquisa.

Os resultados apresentados em nossa revisão sistemática apontam para uma carência de estudos sobre proposição de estratégias e ferramentas com o objetivo de evitar problemas relacionados à segurança. Os trabalhos considerados relevantes apresentam pesquisas empíricas sobre questões relacionadas à segurança, ferramentas de análise estática para programas C, *fuzz testing*, abordagens para testes de caixa branca, execução simbólica e abordagens que consideraram métricas de software.

6.2.1 Análise Estática

No contexto da linguagem C, existem trabalhos que propõem ferramentas que realizam análises estáticas e dinâmicas, como CPPCheck e FlawFinder, que visam dar suporte aos desenvolvedores na detecção de fragilidades no código.

Neste trabalho, utilizamos análise estática como parte de nossa técnica para identificar *warnings* que podem representar fraquezas no código. Conforme apresentado nos capítulos anteriores (Capítulos 1.1 e 3), ferramentas de análise estática são frequentemente imprecisas para detectar fraquezas, causando vários alarmes falsos. WTT utiliza o FlawFinder, uma ferramenta de análise estática que detecta possíveis fraquezas e é oficialmente compatível com o CWE [35].

Alternativamente, CppCheck [26] é uma ferramenta de propósito geral que realiza análise estática no código C, suportando a identificação de *warnings* relacionados à segurança. Além disso, Evans [31] apresenta o Splint, uma ferramenta para verificar estaticamente os programas C em busca de vulnerabilidades de segurança e erros de codificação. Além disso, Nethercote e Seward [89] apresentam Valgrind, um framework para construção de ferramentas de análise dinâmica, por exemplo, para suportar *warnings* relacionados à segurança.

Goseva-Popstojanova e Perhinschi [47] apresentaram um estudo empírico com o objetivo de avaliar a capacidade de três ferramentas de análise estática amplamente utilizadas em detectar vulnerabilidades de segurança. Eles indicaram que o estado da arte das ferramentas de análise estática não é efetivo na detecção de vulnerabilidades de segurança, mostrando

que ainda são necessários aprimoramentos em técnicas e ferramentas para análise de código estático.

Vassallo et al. [107] apresentaram um estudo com 56 desenvolvedores e 11 especialistas da indústria sobre a utilização de ferramentas automáticas de análise estática. Além disso, foi avaliado o uso de ferramentas de análise estática no fluxo de trabalho de projetos de código aberto. Eles identificaram que 71% dos desenvolvedores consideram diferentes categorias de *warnings*, dependendo do contexto de desenvolvimento, 63% dos entrevistados afirmaram que avaliam fatores específicos para priorizar *warnings* a serem corrigidos e 66% dos projetos definem como usar ferramentas específicos, mas apenas 37% impõem seu uso para novas contribuições.

De acordo com uma revisão sistemática recente sobre vulnerabilidade de software [50], abordagens convencionais para testes de segurança que se concentram em análises estáticas, dinâmicas e híbridas sofrem de limitações, como altas taxas de falsos positivos e altos custos computacionais. O WTT aborda essas limitações concentrando-se em uma abordagem de análise híbrida e nos benefícios dos padrões de teste de unidade. Usamos análise estática para identificar código suspeito e estratégias de teste para determinar se algum código apresenta uma fraqueza de código.

Ferramentas de análise estática são úteis em identificar possíveis fraquezas no código e direcionam a execução do WTT. No entanto, como discutido em seções anteriores, as ferramentas de análise estática sozinhas são imprecisas para detectar fraquezas no código devido a vários alarmes falsos.

6.2.2 Fuzz Testing

Fuzzing ou *Fuzz testing* vem sendo amplamente utilizado em testes de segurança [65]. Empresas como Microsoft e Google vem aplicando esta técnica com sucesso a testes de segurança e garantia de qualidade [108; 30]. O objetivo principal do *fuzz testing* é gerar entradas inválidas, aleatórias e inesperadas, especialmente a partir de entradas válidas, que podem desencadear comportamentos errôneos ou de falha, incluindo a exploração de vulnerabilidades [65]. De acordo com Liang et al. [74], o *fuzzer American fuzzy lop (AFL)* [2] é um dos *fuzzer* mais populares. Wang e Kang [112] atribuíram a maior parte da popularidade do AFL à facilidade de uso e configuração. Neste sentido, alguns trabalhos visaram a implementação

de *fuzzers* mais elaborados.

Wang e Kang [112] apontaram uma limitação do AFL relacionada à profundidade de exploração do programa em teste. Neste sentido, eles propuseram o *ADFL ou American Deep Fuzzy Lop*, um método de otimização com o objetivo solucionar este problema. *ADFL* prioriza os ramos de baixo impacto das sementes em teste e, para ramificações de baixa ocorrência, os casos de teste que atingem a ramificação de destino são gerados em uma frequência mais alta.

Wang et al. [113] apresentaram o *UAFL*, um fuzzer guiado por estado de tipo para detectar vulnerabilidades que violam determinadas propriedades de estado de tipo, por exemplo *CWE-416: Use After Free*. Primeiramente são coletadas sequências de operações que violam determinada propriedade de estado de tipo, que são utilizadas para guiar o processo de *fuzzing* e para gerar de casos de teste em direção às sequências identificadas.

Wen et al. [114] propuseram técnica de *fuzz testing* guiada pelo uso de memória, denominada *MemLock*, que apresenta o objetivo de detectar bugs relacionado ao consumo de memória. Primeiramente, MemLock executa a análise estática para identificar instruções e operações relacionadas ao consumo de memória. Então, realiza cobertura de ramificação para orientar a exploração de diferentes caminhos do programa, e coleta informações de consumo de memória para orientar o processo de *fuzzing*.

Neste trabalho, não estamos interessados em propor uma nova técnica de *fuzz testing*, mas sim, utilizar uma técnica já existente. Neste sentido, optamos pela utilização do AFL por se tratar de uma solução já consolidada e bastante utilizada na detecção de vulnerabilidades em grandes projetos, como Mozilla Firefox, OpenSSL e Linux [2].

6.2.3 Whitebox Fuzz Testing

Particularmente, a técnica *whitebox fuzz testing* é discutida em alguns trabalhos por Godefroid et al. [44; 11; 42; 46; 18; 45]. Esta técnica foi proposta como o objetivo de melhorar a técnica *blackbox fuzz testing*, que modifica aleatoriamente entradas válidas do programa e o teste com as entradas modificadas [36]. O objetivo do *whitebox fuzz testing* é executar simbolicamente o sistema em teste de forma dinâmica com entrada bem formada, capturando restrições em ramificações condicionais e gerando dados de teste dinamicamente. As restrições capturadas são negadas e resolvidas com um *constraint solver*, cujas soluções são

mapeadas para novas entradas que exercem diferentes caminhos de execução do programa.

O código-fonte 6.1 apresenta um exemplo de um programa para descrever a técnica *whitebox fuzz testing*. A função `func` recebe três bytes como entrada e apresenta um erro quando o valor de `i` é maior ou igual a três. Fornecendo como entrada a palavra `cab`, na primeira execução, o caminho do programa é formado por todos os ramos *else*. Portanto, as restrições para esta execução é a conjunção das restrições $i_0 /= a$, $i_1 /= b$ e $i_2 /= c$, onde i_0 éa posição zero da entrada, e assim por diante. Então, todas as três restrições são negadas e resolvidas, e as soluções encontradas são mapeadas para novas entradas [44]. Godefroid et al. [45], implementaram a técnica *whitebox fuzz testing* em uma ferramenta chamada `SAGE`, abreviação de *automated guided execution*, para detectar fraquezas do tipo *buffer overflow*. No entanto, a ferramenta proposta encontra pontos fracos em aplicações do Windows x86. Em nossa abordagem, avaliamos fraquezas no contexto da linguagem C.

Código Fonte 6.1: Trecho de código de um programa que pode ocorrer uma falta.

```
void func(char input[3]) {
  int i=0;
  if (input[0] == "a") i++;
  if (input[1] == "b") i++;
  if (input[2] == "c") i++;
  if (i >= 3) abort(); // error
}
```

Whitebox fuzz testing apresenta algumas limitações que podemos citar. A primeira ocorre quando é aplicada a programas reais. A execução sistemática de todos os caminhos de grandes programas pode não ser viável. Este problema é atenuado testando funções isoladas. Outra limitação apresentada por esta estratégia é quando precisa lidar com instruções complexas (manipulações de ponteiro, operações aritméticas etc.). Nestes casos, a aplicação de valores concretos é sugerida para simplificar as restrições [44]. Em Bounimova et al. [11], os autores apresentam um estudo em maior escala com a ferramenta e em Christakis e Godefroid [18] os autores descrevem o uso da ferramenta para testar o *ANI Windows Image Parser*. O WTT, além de detectar mais tipos de fraquezas, não está limitado a algum ambiente, como a técnica proposta em [44].

6.2.4 Execução Simbólica

Execução simbólica é outra estratégia utilizada por alguns trabalhos selecionados em nossa SLR [15; 100; 106; 43]. A execução simbólica corresponde a uma ferramenta poderosa para testar e depurar programas que executam o programa com valores simbólicos em vez de concretos [62]. Xu et al. [116] propuseram uma ferramenta chamada `Splat`, que utiliza execução simbólica para rastrear todos os valores simbólicos para `buffers` de entrada com o objetivo de detectar estouro de `buffer`. Uma limitação desta abordagem é a necessidade de explorar todos os caminhos quando o programa em teste é grande e complexo.

Zhang et al. [117] apresentaram `SecTAC`, uma ferramenta baseada em técnica de rastreamento (*trace-based technique*) para detectar vulnerabilidades em programas C. Eles utilizaram casos de teste existentes para gerar um rastreamento de execução. Em seguida, a execução simbólica é aplicada e as restrições do programa (PC) e as restrições de segurança (SC) são geradas. Para esta abordagem, eles levam em consideração que uma vulnerabilidade de segurança é detectada se houver uma atribuição de valores às variáveis que satisfaça o PC, mas viole o SC. Duas limitações foram apresentadas pelos autores. A primeira é que a abordagem depende diretamente da existência de casos de teste. Os casos de teste precisam ser implementados antes de executar a abordagem. A segunda está relacionada ao tamanho de um caminho a ser executado a partir de um programa extenso. Em nosso trabalho, apresentamos uma técnica que utiliza análise estática para indicar código suspeito no código-fonte do programa e estratégias de teste para determinar se algum código apresenta uma fraqueza de código.

A Figura 6.5 apresenta uma representação dos passos para aplicar a abordagem. Na Etapa I, os rastreamentos de execução são gerados a partir de casos de teste existentes. Na Etapa II, cada rastreamento de execução é analisado para extrair PC e SC em cada ponto do rastreamento. Finalmente, na Etapa III, eles procuram entradas que possam detectar vulnerabilidades de segurança.

6.2.5 Métricas de Software

No contexto de correções de vulnerabilidades, Li e Paxson [71] usaram o NVD em um estudo empírico em grande escala de correções de segurança. Eles investigaram mais de

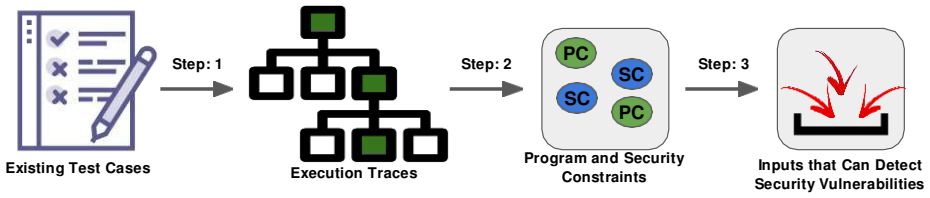

Figura 6.5: Técnica baseada em rastreamento para detecção de vulnerabilidades em programas C.

4.000 correções de *bugs* para mais de 3.000 vulnerabilidades que afetaram um conjunto diversificado de 682 projetos de software de código aberto. Eles também caracterizaram as correções de segurança em comparação com as correções de *bugs* não relacionados à segurança, investigando a complexidade de diferentes tipos de correções.

Farris et al. [32] analisaram vulnerabilidades conhecidas com o objetivo de propor uma abordagem de gerenciamento de vulnerabilidade. Eles verificam duas métricas de desempenho: tempo para correção da vulnerabilidade e exposição total da vulnerabilidade. Eles usaram um algoritmo para priorizar as vulnerabilidades de correção, de modo que as métricas de desempenho sejam otimizadas de acordo com as restrições de recursos fornecidas. A abordagem proposta orienta para melhorar os processos de resposta à vulnerabilidade. Analisar vulnerabilidades conhecidas é essencial para verificar os padrões de ocorrência de vulnerabilidade. Neste trabalho, estamos interessados em detectar fraquezas durante os estágios iniciais de desenvolvimento.

Chowdhury et al. [17] propuseram um estudo empírico com mais de quatro anos de histórico de vulnerabilidade do Mozilla Firefox. Eles visavam verificar a relação entre métricas de complexidade e ocorrências de vulnerabilidade. Eles descobriram que as métricas de complexidade estão correlacionadas ao número de vulnerabilidades. Além disso, Neuhaus et al. [90] realizaram um estudo para verificar se componentes que compartilhavam conjuntos semelhantes de chamadas de função são mais propícios a serem vulneráveis. Abal et al. [6] propuseram um estudo qualitativo de bugs de variabilidade do repositório Linux Kernel que fornece *insights* sobre a natureza e ocorrências. Eles apresentaram maneiras que a variabilidade afeta e aumenta a complexidade dos bugs de software. Williams et al. [115] discutiram 55 práticas de software usadas para prevenir, detectar e responder a erros de implementação e falhas de design. Eles analisam os dados de um subconjunto de 113 práticas

de segurança de software de 109 empresas por 42 meses, conforme relatado no Building Security In Maturity Model [75]. O estudo indica que as empresas costumam adotar práticas de resposta, seguidas de práticas de detecção, seguidas de práticas de prevenção. Além disso, pesquisadores devem desenvolver ferramentas com o objetivo de habilitar equipes de desenvolvimento a prevenir e detectar bugs de implementação e falhas de design de forma eficiente. Em nosso trabalho estamos propondo uma técnica para ser inserida no processo de desenvolvimento para detectar fragilidades do código.

Clemente et al. [22] apresentaram um modelo baseado em *deep learning* para prever bugs de segurança. Eles acreditam que existe uma relação entre bugs de segurança e métricas de software. Eles examinam essa relação com relação à produção de novos *insights* sobre a formação de bugs de segurança. Eles usaram a lista CVE para rastrear a versão do software onde o bug foi relatado e filtraram os bugs apenas para incluir bugs relacionados à segurança. Eles dividiram as métricas adotadas em três categorias: complexidade, volume e métricas orientadas a objetos. Eles afirmam que os desenvolvedores têm outra ferramenta para prever a propensão a bugs de segurança; portanto, eles podem alocar recursos de teste de forma mais prudente. Peters et al. [93] apresentaram um modelo de predição baseado em teste para identificar relatórios de bugs relacionados à segurança com rapidez e precisão. Eles avaliam 45.940 relatórios de bugs de dois projetos com relação à identificação de quais estão relacionados à segurança.

Os trabalhos citados, apresentam algumas abordagens que visam coletar intuições sobre correções, métricas e padrões de vulnerabilidade. Acreditamos que esses trabalhos são essenciais para aumentar o conhecimento sobre vulnerabilidades. Além disso, podemos determinar características fundamentais com o objetivo de gerar casos de teste automaticamente.

6.3 Considerações Finais

Este capítulo apresentou uma revisão sistemática da literatura sobre trabalhos que apresentaram abordagens com o objetivo de detectar problemas relacionados à segurança e uma discussão mais detalhada sobre os trabalhos considerados mais relevantes para esta pesquisa. Foi possível perceber que a identificação de problemas relacionados à segurança de sistemas de software vem sendo discutido, porém, trabalhos anteriores mostraram que não há con-

senso entre os desenvolvedores sobre qual ferramenta ou estratégia pode ser utilizada para evitar fraquezas de código. Além disso, foram identificados poucos trabalhos relacionados explicitamente à detecção de vulnerabilidades em sistemas C. Neste sentido, desenvolvedores precisam usar muitas ferramentas e abordagens para tentar mitigar problemas relacionados à segurança.

Tabela 6.7: Lista de trabalhos aceitos na revisão sistemática.

Autores	Tipo de Estudo	Referência
Godefroid, P.	Empírico	[40]
Xu, R., et al.	Empírico	[116]
Nguyen, D., et al.	Empírico	[91]
Bryce, R., et al.	Empírico	[13]
Al-Hajjaji, M., et al.	Empírico	[8]
Arindam, C. and Godefroid, P.	Empírico	[15]
Godefroid, P., Nils K., and Koushik S.	Empírico	[43]
Zhang, D. et al.	Empírico	[117]
Henard C., et al.	Empírico	[51]
Zitser, M., Richard L., and Tim L.	Empírico	[119]
Dan, H. and Hierons, R. M.	Empírico	[27]
Cohen, M. B. et al.	Empírico	[24]
Bounimova, E., Godefroid, P., & Molnar, D.	Empírico	[11]
Christakis M. & Godefroid, P.	Empírico	[18]
Godefroid, P.	Empírico	[39]
Godefroid, P., Levin, M. Y., & Molnar, D. A.	Empírico	[44]
Vanoverberghe, D., Tillmann, N., & Piessens, F.	Empírico	[106]
Sahaf, Z. et al.	Empírico	[97]
Lachmann, R. et al.	Empírico	[70]
Godefroid, P., & Luchaup, D.	Empírico	[46]
Li, H. et al.	Empírico	[73]
Stricker, V., Metzger, A., & Pohl, K.	Empírico	[100]
Murphy, C. et al.	Empírico	[88]
Mouelhi, T., Le Traon, Y., & Baudry, B.	Empírico	[81]
Molnar, D., Li, X. C. & Wagner, D.	Empírico	[78]
Cohen, M. B., Matthew B. D., and Shi, J.	Empírico	[23]
Catelani, M. et al.	Estudo Teórico	[14]
Godefroid, P. et al.	Empírico	[42]
Godefroid, P., Levin, M. Y., & Molnar, D.	Empírico	[45]
Ahmed, B. S. et al.	Revisão Sistemática	[7]
Huang, S. K. et al.	Empírico	[55]

Tabela 6.8: Lista de técnicas de teste utilizadas para detectar tipos de fraquezas.

Tipo de teste	Fraquezas	Trabalhos
Teste de Unidade	CWE-120: Buffer Overflow	[116; 117; 119]
		[45; 11; 18]
		[39; 44; 46]
		[73]
	CWE-134: String Errors	[55]
	CWE-190: Integer Overflow	[78]
	CWE-369: Division by Zero	[18]
Teste de Sistema	CWE-401: Memory Leak	[14]

Capítulo 7

Considerações Finais

Neste capítulo sumarizamos os principais resultados deste trabalho (Seção 7.1), uma síntese das principais contribuições (Seção 7.2) e apresentamos algumas sugestões para trabalhos futuros (Seção 7.3).

7.1 Conclusões

O principal objetivo desta tese foi propor uma técnica baseada em análise estática, automação de casos de teste e *fuzz testing* para auxiliar desenvolvedores/testadores na detecção de fraquezas de código em programas C durante os primeiros estágios do processo de desenvolvimento de software. Além disso, a técnica proposta foi implementada no framework WTT.

Considerando as questões de pesquisa definidas no Capítulo 1, os seguintes resultados foram alcançados:

Questão de pesquisa 1: *Quais os tipos de fraquezas podem ser detectados utilizando teste de software em programas C?*

Para responder esta questão de pesquisa, apresentamos uma visão do estado da arte no contexto deste trabalho (Capítulo 6). Apresentamos inicialmente uma revisão sistemática da literatura para identificar técnicas utilizadas na detecção de fraquezas de software (Seção 6.1) e uma discussão detalhada dos trabalhos considerados mais relevantes (Seção 6.2).

Por meio da revisão sistemática, verificamos que 81,25% dos artigos aceitos discutiram técnicas de teste. Classificamos estes de acordo com a abordagem, sendo 75,86% como teste

de caixa branca e 24,14% como teste de caixa preta. Além disso, verificamos o nível de teste abordado em cada trabalho, onde foi visto que seis artigos abordaram o teste de integração, três abordaram teste de sistema e vinte abordaram testes de unidade.

Finalmente, consideramos o tipo de fraqueza abordada nos trabalhos selecionados. Verificamos que mais de 40% dos artigos selecionados discutem algum tipo de fraqueza. A maioria deles é classificada como *CWE-120: Buffer Copy without Checking Size of Input (Classic Buffer Overflow)*. Além disso, os artigos selecionados discutem abordam *CWE-401: Missing Release of Memory after Effective Lifetime, CWE-134: Use of Externally-Controlled Format String, CWE-190: Integer Overflow or Wraparound e CWE-369: Divide By Zero.* Do total de trabalhos que discutem fraquezas, a maioria (92,3%) utilizou teste unitário para detectá-las.

Questão de pesquisa 2: *Teste de software combinado com a análise estática pode fornecer a base para uma técnica de detecção de fraquezas eficaz?*

Considerando a técnica proposta e os resultados obtidos por meio de sua aplicação nos estudos apresentados, concluímos que teste de software combinado com análise estática pode fornecer a base para uma técnica de detecção de fraquezas eficaz.

Objetivando alcançar a resposta para esta questão de pesquisa, inicialmente, estudamos como detectar fraquezas em código fonte de sistemas de softwares desenvolvidos em linguagem C, por meio de estratégias de verificação e validação. Em seguida, foi proposta uma técnica baseada em análise estática e *fuzz testing* para detectar fraquezas em código-fonte C (Capítulo 3).

Após a implementação da técnica em um framework, foi possível executar um estudo para avaliar a aplicação da técnica em programas reais (Capítulo 4). Neste estudo, executamos WTT em 103 *warnings* e rotulamos 24 deles como detectados, 37 como não detectados e 42 como não testados. Além disso, categorizamos os *warnings* em três tipos de fraquezas: *Buffer Overflow*, *Integer Overflow* e *Format String*.

Foi executado um segundo estudo com o conjunto de dados *Juliet Test Suite* (Capítulo 5). Neste estudo, selecionamos 1.417 arquivos de código fonte C do conjunto de dados *Juliet Test Suite* com fraquezas dos tipos: *CWE-190: Integer Overflow or Wraparound*, *CWE-191: Integer Underflow (Wrap or Wraparound)*, *CWE-369: Divide By Zero*. Dos arquivos selecionados, foram testadas um total de 2.834 funções (*bad* e *good*). Do total de funções

testadas, 57,79% são fraquezas do tipo CWE-190, 21,88% do tipo CWE-191 e 20,33% do tipo CWE-369. WTT classificou como detectadas todas as funções rotuladas como *bad* e classificou como não detectadas todas as funções rotuladas como *good*.

7.2 Contribuições

Em resumo, obtivemos as seguintes contribuições neste trabalho de doutorado:

1. Uma revisão sistemática da literatura pela qual foi possível identificar (i) Tipos de fraquezas mais recorrentes; (ii) Tipos de fraquezas que podem ser detectadas por técnicas específicas; (iii) Técnicas e ferramentas que vem sendo aplicadas na detecção de fraquezas;

2. Guias para implementação de casos de teste para detecção de fraquezas específicas, considerando as principais características de uma fraqueza. Os guias propostos foram aplicados em *templates* para geração automática de casos de teste. A atual versão do WTT apresenta o processo de construção de casos de teste de forma manual. No entanto, a elaboração dos guias para implementação dos casos de teste reduziu consideravelmente o esforço neste processo, indicando os elementos necessários aos casos de teste. Além disso, foi possível automatizar este processo para um conjunto de dados específico, como apresentado no Capítulo 5.

3. Uma técnica para detectar fraquezas em programas C. Esta técnica foi implementada em um *framework* chamado *Weaknesses TesTing ou WTT*;

4. Um estudo sobre a utilização do WTT em programas reais.

5. Publicação do artigo *Towards a Technique to Detect Weaknesses in C Programs* no *XXXV Simpósio Brasileiro de Engenharia de Software* [1].

6. Uma avaliação empírica com uma base de código que apresenta exemplos de fraquezas de código;

[1] https://dl.acm.org/doi/abs/10.1145/3474624.3474633

De modo geral, esperamos que as contribuições apresentadas neste trabalho possam proporcionar suporte para desenvolvedores/testadores na detecção de fraquezas de código durante o processo de desenvolvimento de software.

7.3 Trabalhos Futuros

Com a conclusão deste trabalho, existem algumas oportunidades para trabalhos futuros. A seguir, algumas ideias são descritas:

1. **Evolução do WTT:** A versão atual do WTT apresenta implementação automática de casos de teste baseada em templates predefinidos baseados no conjunto de dados *Juliet Test Suite*. Neste sentido, uma proposta de trabalho futuro seria evoluir este processo com o objetivo de generalizar a implementação automática de casos de teste;

2. **Estudo empírico:** Nós avaliamos o WTT com um conjunto específicos de programas reais. Realizar estudos empíricos incluindo mais exemplos de programas;

3. **WTT na indústria:** O objetivo principal do *framework* proposto é detecção de fraquezas nos estágios iniciais do processo de desenvolvimento. Neste sentido, pretendemos realizar um estudo para avaliar a utilização do WTT por desenvolvedores/testadores da indústria;

4. **Detecção de outras fraquezas de código:** Por meio da execução dos estudos apresentados neste trabalho (Capítulo 4 e Capítulo 5), foi possível verificar que o WTT detecta alguns tipos de fraquezas. Outra possibilidade de trabalho futuro seria avaliar se o WTT é capaz de detectar tipos de fraquezas não abordadas neste trabalho;

5. **WTT para outras linguagens de programação:** Como é possível perceber, WTT corresponde a um conjunto de *scripts*, ferramentas e técnicas que, neste trabalho, foi aplicado no contexto de programas C. No entanto, acreditamos que seja possível expandir o *framework* para detecção de fraquezas em outras linguagens.

Bibliografia

[1] LaPES. start: State of the art through systematic review. Disponível em: `http://lapes.dc.ufscar.br/tools/start_tool`, 2017. [Acesso em: 12 de Novembro de 2017].

[2] American fuzzy lop 2.52b. Disponível em: `https://lcamtuf.coredump.cx/afl/`, 2019. [Acesso em: 20 de Setembro de 2019].

[3] Cmocka - unit Testing Framework for C. Disponível em: `https://cmocka.org/`, 2019. [Acesso em: 20 de Setembro de 2019].

[4] Common Vulnerabilities and Exposures. Disponível em: `http://cve.mitre.org/`, 2019. [Acesso em: 20 de Setembro de 2019].

[5] National Vulnerability Database. Disponível em: `https://nvd.nist.gov`, 2019. [Acesso em: 20 de Setembro de 2019].

[6] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability bugs in highly configurable systems: a qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–34, 2018.

[7] B. Ahmed, K. Zamli, W. Afzal, and M. Bures. Constrained interaction testing: A systematic literature study. *IEEE Access*, 5:25706–25730, 2017.

[8] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool demo: testing configurable systems with featureide. *ACM SIGPLAN Notices*, 52(3):173–177, 2017.

[9] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[10] Gilles Bernot. Testing against formal specifications: A theoretical view. In *International Joint Conference on Theory and Practice of Software Development*, pages 99–119. Springer, 1991.

[11] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 13rd International Conference on Software Engineering*, pages 122–131, 2013.

[12] Pierre Bourque and Richard Fairley. Guide to the software engineering body of knowledge version 3.0 (swebok guide v3.0). Disponível em: `https://www.computer.org/education/bodies-of-knowledge/software-engineering`, 2004. [Acesso em: 12 de Novembro de 2020].

[13] Renée C Bryce, Charles J Colbourn, and Myra B Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th international conference on Software engineering*, pages 146–155. ACM, 2005.

[14] Marcantonio Catelani, Lorenzo Ciani, Valeria L Scarano, and Alessandro Bacioccola. Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use. *Computer Standards & Interfaces*, 33(2):152–158, 2011.

[15] A. Chakrabarti and P. Godefroid. Software partitioning for effective automated unit testing. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 262–271. ACM, 2006.

[16] B. Chess and J. West. *Secure programming with static analysis*. Pearson Education, 2007.

[17] I. Chowdhury and M. Zulkernine. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 25th Symposium on Applied Computing*, pages 1963–1969, Sierre, Switzerland., 2010. ACM.

[18] M. Christakis and P. Godefroid. Proving memory safety of the ani windows image parser using compositional exhaustive testing. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 373–392. Springer, 2015.

[19] S. Christey, M. Brown, D. Kirby, B. Martin, and A. Paller. Cwe/sans top 25 most dangerous software errors. *Common Weakness Enumeration*, 2011.

[20] S. Christey, J. Kenderdine, J. Mazella, and B. Miles. Common Weakness Enumeration. *Mitre Corporation*, 2013.

[21] T. Clarke. Fuzzing for software vulnerability discovery. *Department of Mathematic, Royal Holloway, University of London, Tech. Rep. RHUL-MA-2009-4*, 2009.

[22] Caesar Jude Clemente, Fehmi Jaafar, and Yasir Malik. Is predicting software security bugs using deep learning better than the traditional machine learning algorithms? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 95–102. IEEE, 2018.

[23] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[24] Myra B Cohen, Peter B Gibbons, Warwick B Mugridge, and Charles J Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th international conference on Software engineering*, pages 38–48. IEEE Computer Society, 2003.

[25] Microsoft Corporation. Simplified implementation of the microsoft sdl. Disponível em: `https://www.microsoft.com/en-us/download/details.aspx?id=12379`, 2010. [Acesso em: 12 de Novembro de 2019].

[26] Cppcheck. A tool for static c/c++ code analysis. Disponível em: `http://cppcheck.sourceforge.net/`, 2021. [Acesso em: 15 de Maio de 2021].

[27] Haitao Dan and Robert M Hierons. Smt-c: A semantic mutation testing tools for c. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 654–663. IEEE, 2012.

[28] René G de Vries and Jan Tretmans. Towards formal test purposes. *Formal Approaches to Testing of Software–FATES*, 1:61–76, 2001.

[29] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[30] C. Evans, M. Moore, and T. Ormandy. Google online security blog – fuzzing at scale. Disponível em: `https://security.googleblog.com/2011/08/fuzzing-at-scale.html`, 2021. [Acesso em: 17 de Maio de 2021].

[31] D. Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN Notices*, volume 31, pages 44–53. ACM, 1996.

[32] Katheryn A Farris, Ankit Shah, George Cybenko, Rajesh Ganesan, and Sushil Jajodia. Vulcon: A system for vulnerability prioritization, mitigation, and management. *ACM Transactions on Privacy and Security (TOPS)*, 21(4):1–28, 2018.

[33] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property oriented test case generation. In *International Workshop on Formal Approaches to Software Testing*, pages 147–163. Springer, 2003.

[34] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 65–73, New York, NY, USA, 2016. ACM.

[35] Flawfinder. Disponível em: `https://dwheeler.com/flawfinder/`, 2021. [Acesso em: 17 de Maio de 2021].

[36] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, volume 4, pages 59–68. Seattle, 2000.

[37] The Apache Software Foundation. Apache FreeMarker. Disponível em: `https://freemarker.apache.org/`, 2021. [Acesso em: 15 de Maio de 2021].

[38] S. Frei, D. Schatzmann, B. Plattner, and B. Trammell. *Modeling the security ecosystem - the dynamics of (In)security*, pages 79–106. Springer US, 2010.

[39] P. Godefroid. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*, pages 539–549. ACM, 2014.

[40] P. Godefroid. Between testing and verification: Dynamic software model checking., 2016.

[41] P. Godefroid, P. de Halleux, A. Nori, S. Rajamani, W. Schulte, N. Tillmann, and M. Levin. Automating software testing using program analysis. *IEEE software*, 25(5), 2008.

[42] P. Godefroid, P. Halleux, A. Nori, S. Rajamani, W. Schulte, N. Tillmann, and M. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.

[43] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[44] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 16th Network and Distributed Systems Security*, pages 151–166, 2008.

[45] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[46] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 23–33, 2011.

[47] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[48] D. Graham, Erik V. V., and I. Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[49] ISDW Group et al. 1044-2009-ieee standard classification for software anomalies. *IEEE, New York*, 2010.

[50] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications*, page 103009, 2021.

[51] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 188–197. IEEE, 2013.

[52] G. Hoglund and G. McGraw. *Exploiting software: How to break code*. Pearson Education India, 2004.

[53] M. Howard, D. LeBlanc, and J. Viega. *24 deadly sins of software security: programming flaws and how to fix them*. McGraw-Hill, Inc., 2010.

[54] M. Howard and S. Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[55] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability*, 63(1):270–289, 2014.

[56] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55, 2012.

[57] Paul C Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2013.

[58] L. Juranić. Using fuzzing to detect security vulnerabilities. *Retrieved Apr*, 26:2012, 2006.

[59] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, pages 805–824, 2011.

[60] M. Khatibsyarbini, M. Isa, D. Jawawi, and R. Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 2017.

[61] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[62] James King. Symbolic execution and program testing. *Commun. ACM*, pages 385–394, 1976.

[63] B. Kitchenham and P. Brereton. A systematic review of systematic review process research in software engineering. *Inf. Softw. Technol.*, 55(12):2049–2075, 2013.

[64] B. Kitchenham and S Charters. Guidelines for performing systematic literature reviews in software engineering, 2007.

[65] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. ACM.

[66] Ivan Victor Krsul. *Software vulnerability analysis*. Purdue University West Lafayette, IN, 1998.

[67] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, pages 418–421, 2004.

[68] Thien La. Secure software development and code analysis tools. *SANS Inst. InfoSec Read. Room*, pages 1–51, 2002.

[69] Bill Laboon. *A Friendly Introduction to Software Testing*. CreateSpace Independent Publishing Platform, 2016.

[70] Remo Lachmann, Simon Beddig, Sascha Lity, Sandro Schulze, and Ina Schaefer. Risk-based integration testing of software product lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 52–59. ACM, 2017.

[71] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215. ACM, 2017.

[72] Hongzhe Li, Jaesang Oh, Hakjoo Oh, and Heejo Lee. Automated source code instrumentation for verifying potential vulnerabilities. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 211–226. Springer, 2016.

[73] Hongzhe Li, Jaesang Oh, Hakjoo Oh, and Heejo Lee. Automated source code instrumentation for verifying potential vulnerabilities. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 211–226. Springer, 2016.

[74] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[75] Gary McGraw, Brian Chess, and Sammy Migues. Building security in maturity model. *Fortify & Cigital*, 2009.

[76] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 643–654, 2016.

[77] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the c preprocessor: An interview study. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[78] D. Molnar, X. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[79] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[80] A. Moraes, W. Andrade, and P. Machado. A family of test selection criteria for timed input-output symbolic transition system models. *Science of Computer Programming*, 2016.

[81] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. Transforming and selecting functional test cases for security policy testing. In *2009 International Conference on Software Testing Verification and Validation*, pages 171–180. IEEE, 2009.

[82] Maryam Mouzarani and Babak Sadeghiyan. Towards designing an extendable vulnerability detection method for executable codes. *Information and Software Technology*, 80:231–244, 2016.

[83] Maryam Mouzarani, Babak Sadeghiyan, and Mohammad Zolfaghari. A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 42–49. IEEE, 2015.

[84] Maryam Mouzarani, Babak Sadeghiyan, and Mohammad Zolfaghari. A smart fuzzing method for detecting heap-based vulnerabilities in executable codes. *Security and Communication Networks*, 9(18):5098–5115, 2016.

[85] Maryam Mouzarani, Babak Sadeghiyan, and Mohammad Zolfaghari. Smart fuzzing method for detecting stack-based buffer overflow in binary codes. *Iet Software*, 10(4):96–107, 2016.

[86] Maryam Mouzarani, Babak Sadeghiyan, and Mohammad Zolfaghari. Detecting injection vulnerabilities in executable codes with concolic execution. In *2017 8th IEEE*

International Conference on Software Engineering and Service Science (ICSESS), pages 50–57. IEEE, 2017.

[87] R. Muniz, L. Braz, R. Gheyi, W. Andrade, B. Fonseca, and M. Ribeiro. A qualitative analysis of variability weaknesses in configurable systems with *#ifdefs*. In *Proceedings of the 12th International Workshop on Variability Modeling of Software-Intensive Systems*, pages 51–58, 2018.

[88] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. In *2009 International Conference on Software Testing Verification and Validation*, pages 111–120. IEEE, 2009.

[89] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[90] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th Conference on Computer and Communications Security*, pages 529–540, 2007.

[91] D. Nguyen, P. Hung, and V. Nguyen. A method for automated unit testing of c programs. In *Information and Computer Science (NICS), 2016 3rd National Foundation for Science and Technology Development Conference on*, pages 17–22. IEEE, 2016.

[92] National Institute of Standards and Technology. NSA Center for Assured Software. Disponível em: `https://samate.nist.gov/SRD/testsuite.php`, 2021. [Acesso em: 15 de Junho de 2021].

[93] Fayola Peters, Thein Tun, Yijun Yu, and Bashar Nuseibeh. Text filtering and ranking for security bug report prediction. *IEEE Transactions on Software Engineering*, 2017.

[94] C. P. Pfleeger, S. L. Pfleeger, and J. Margulies. *Security in computing*. Prentice Hall Professional Technical Reference, 2015.

[95] Jiadong Ren, Zhangqi Zheng, Qian Liu, Zhiyao Wei, and Huaizhi Yan. A buffer overflow prediction approach based on software metrics and machine learning. *Security and Communication Networks*, 2019, 2019.

[96] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[97] Zahra Sahaf, Vahid Garousi, Dietmar Pfahl, Rob Irving, and Yasaman Amannejad. When to automate software testing? decision support based on system dynamics: an industrial case study. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 149–158. ACM, 2014.

[98] Ian Sommerville. Software engineering 9th edition. *ISBN-10*, 137035152:18, 2011.

[99] H. Spencer and G. Collyer. `#ifdef` considered harmful, or portability experience with C news. In *USENIX Summer*, pages 185–197. USENIX Association, 1992.

[100] V. Stricker, A. Metzger, and K. Pohl. Avoiding redundant testing in application engineering. In *International Conference on Software Product Lines*, pages 226–240. Springer, 2010.

[101] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[102] Mohammad Tahaei, Kami Vaniea, Konstantin Beznosov, and Maria K Wolters. Security notifications in static analysis tools: Developers' attitudes, comprehension, and ability to act on them. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2021.

[103] Jan Tretmans. Testing concurrent systems: A formal approach. In *International Conference on Concurrency Theory*, pages 46–65. Springer, 1999.

[104] Jan Tretmans. Model-based testing: Property checking for real. In *International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices*, 2004.

[105] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[106] D. Vanoverberghe, N. Tillmann, and F. Piessens. Test input generation for programs with pointers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 277–291. Springer, 2009.

[107] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.

[108] Sdl Process: Verification. Disponível em: `https://www.microsoft.com/en-us/securityengineering/sdl/practices`, 2021. [Acesso em: 17 de Maio de 2021].

[109] Nicholas Visalli, Lin Deng, Amro Al-Suwaida, Zachary Brown, Manish Joshi, and Bingyang Wei. Towards automated security vulnerability and software defect localization. In *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 90–93. IEEE, 2019.

[110] National Vulnerability Database. Statistics Results for CWE-134. Disponível em: `https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-120`, 2021. [Acesso em: 15 de Maio de 2021].

[111] National Vulnerability Database. Statistics Results for CWE-134. Disponível em: `https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-134`, 2021. [Acesso em: 15 de Maio de 2021].

[112] Chenxin Wang and Shunyao Kang. Adfl: an improved algorithm for american fuzzy lop in fuzz testing. In *International Conference on Cloud Computing and Security*, pages 27–36. Springer, 2018.

[113] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010. IEEE, 2020.

[114] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 765–777, 2020.

[115] Laurie Williams, Gary McGraw, and Sammy Migues. Engineering security vulnerability prevention, detection, and response. *IEEE Software*, 35(5):76–80, 2018.

[116] R. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 27–38. ACM, 2008.

[117] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, and W. Wang. Detecting vulnerabilities in c programs using trace-based testing. In *Proceedings of the 40th International Conference on Dependable Systems and Networks (DSN)*, pages 241–250, 2010.

[118] Yanping Zhang, Yang Xiao, Kaveh Ghaboosi, Jingyuan Zhang, and Hongmei Deng. A survey of cyber crimes. *Security and Communication Networks*, 5(4):422–437, 2012.

[119] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.