UFRR

UNIVERSIDADE FEDERAL DE RORAIMA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Map2Check – Uma Abordagem para Teste de Software

Guilherme Lucas Pereira Bernardo

Boa Vista - RR

Novembro de 2024

Guilherme Lucas Pereira Bernardo

Map2Check – Uma Abordagem para Teste de Software

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Roraima como requisito parcial para a obtenção do grau de bacharel em Ciência da Computação.

Orientador(a)

Prof. Dr. Herbert Oliveira Rocha

Universidade Federal de Roraima

Departamento de Ciência da Computação

Boa Vista - RR

Novembro de 2024

DECLARAÇÃO DE AUTORIA

Eu, **Guilherme Lucas Pereira Bernardo** (código de matricula **2019004044**), autor da(o) monografia/TCC (Trabalho de Conclusão de Curso) sob o título **Map2Check – Uma Abordagem para Teste de Software**, declaro que o trabalho em referência é de minha total autoria e de minha inteira responsabilidade o texto apresentado. Declaro, ainda, que as citações e paráfrases dos autores estão indicadas com as respectivas obras e anos de publicação. Declaro, para os devidos fins que estou ciente:

- dos Artigos 297 a 299 do Código Penal, Decreto-Lei n. 2.848 de 7 de dezembro de 1940;
- da Lei n. 9.610, de 19 de fevereiro de 1998, sobre os Direitos Autorais; e
- que plágio consiste na reprodução de obra alheia e submissão da mesma como trabalho próprio ou na inclusão, em trabalho próprio, de ideias, textos, tabelas ou ilustrações (quadros, figuras, gráficos, fotografias, retratos, lâminas, desenhos, organogramas, fluxogramas, plantas, mapas e outros) transcritos de obras de terceiros sem a devida e correta citação da referência.

O corpo docente responsável pela avaliação deste trabalho poderá não aceitar o referido trabalho caso os pontos mencionados acima sejam descumpridos, por conseguinte, considerar-me reprovado.

_____

Assinatura do acadêmico(a)
Boa Vista - RR, data (por extenso).

FOLHA DE APROVAÇÃO

Monografia de Graduação sob o título **Map2Check – Uma Abordagem para Teste de Software** apresentada por **Guilherme Lucas Pereira Bernardo** e aceita pelo Departamento de Ciência da Computação da Universidade Federal de Roraima, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Titulação e nome do(a) orientador(a)

Orientador(a)

Departamento

Universidade

Titulação e nome do(a) membro da banca examinadora

Co-orientador(a), se houver

Departamento

Universidade

Titulação e nome do membro da banca examinadora

Departamento

Universidade

Titulação e nome do membro da banca examinadora

Departamento

Universidade

Boa Vista - RR, data de aprovação (por extenso).

Agradecimentos

Gostaria de expressar minha profunda gratidão à minha família, especialmente aos meus pais, pelo amor incondicional, apoio contínuo e pelas valiosas lições de vida que me proporcionaram. Sem o apoio e incentivo de vocês eu não teria terminado este curso. Também desejo estender meus sinceros agradecimentos aos meus amigos e minha namorada Vitoria, que se tornaram uma rede de apoio essencial, proporcionando momentos de descanso, risadas e conselhos sábios. Cada conversa e gesto de amizade fortaleceu minha determinação em completar este trabalho. Aos meus colegas de curso, que compartilharam comigo momentos de estudo, discussões enriquecedoras e a busca conjunta pelo conhecimento, expresso minha gratidão pela camaradagem e pelos *insights* valiosos que enriqueceram minha jornada acadêmica. Não poderia deixar de mencionar meu orientador, cuja experiência e paciência foram cruciais para a formação deste trabalho. Suas orientações precisas e *feedback* construtivo foram fundamentais para superar os desafios da pesquisa e da escrita. Dedico por último, agradecimentos à minha falecida avó Elza, que em vida me incentivou de diversas maneiras, seja comprando doces ou salgados, conversando comigo enquanto estive sozinho ou desmotivado.

Te amo vó.

Map2Check – Uma Abordagem para Teste de Software

Autor: Guilherme Lucas Pereira Bernardo

Orientador: Prof. Dr. Herbert Oliveira Rocha

Resumo

O processo de teste de software desempenha um papel crucial na verificação da conformidade do software desenvolvido com as especificações do projeto, além de garantir a qualidade do produto final. Uma abordagem dinâmica amplamente adotada é o fuzzing, que cria casos de teste para avaliar as propriedades de segurança do software. Em particular, o fuzzing baseado em mutação guiada por cobertura se destaca como uma técnica eficaz para identificar automaticamente entradas que revelam falhas e comportamentos indefinidos no sistema em teste. Um exemplo de ferramenta que utiliza essa técnica é o Map2Check, que combina fuzzing, execução simbólica e invariantes indutivos para verificar software de forma automatizada. Neste contexto, este projeto propõe avaliar o software Map2Check para geração de dados de teste com fuzzing utilizando o padrão da Competição de Teste de Software (Test-Comp) com foco em programas na linguagem C.

***Palavras-chave*: Teste de Software, Fuzzing, Execução Simbolica.**

Map2Check – Uma Abordagem para Teste de Software

Autor: Guilherme Lucas Pereira Bernardo

Orientador: Prof. Dr. Herbert Oliveira Rocha

Abstract

The software testing process plays a crucial role in verifying the conformity of the software developed with the project specifications, as well as guaranteeing the quality of the final product. A widely adopted dynamic approach is fuzzing, which creates test cases to evaluate the software's security properties. In particular, fuzzing based on coverage-driven mutation stands out as an effective technique for automatically identifying inputs that reveal flaws and undefined behavior in the system under test. An example of a tool that uses this technique is Map2Check, which combines fuzzing, symbolic execution and inductive invariants to verify software in an automated way. In this context, this project proposes evaluate the Map2Check software for generating test data with fuzzing using the Software Testing Competition (Test-Comp) standard with a focus on C language programs.

***Keywords*: Software testing, Fuzzing, Symbolic Execution**

1 Introdução

A verificação de software é uma fase crucial no processo de Engenharia de Software, fundamental para garantir a qualidade do produto desenvolvido. Este processo visa identificar e corrigir erros antes da entrega final, assegurando que o produto atenda às expectativas dos usuários e funcione conforme o esperado. A importância dos testes de software é particularmente evidente em sistemas críticos, que são essenciais para a segurança, saúde e operação de sistemas vitais. A falta de testes adequados pode resultar em falhas catastróficas, como no caso do Therac-25 (LEVESON; TURNER, 1993), um dispositivo de tratamento por radiação para câncer que, devido à ausência de verificação e testes adequados, resultou em seis mortes entre 1985 e 1987 devido à emissão de doses elevadas de radiação.

Diante dos diferentes tipos de sistemas, os softwares críticos, frequentemente escritos na linguagem de programação C, são aqueles que, devido à sua natureza, têm um impacto significativo na segurança, saúde ou funcionamento de sistemas vitais (HE, 2009). A qualidade desses softwares é fundamental, pois seu mau funcionamento pode ter consequências graves. A verificação de software é, portanto, um componente essencial para garantir a qualidade desses sistemas, envolvendo a realização de testes em todas as fases do desenvolvimento.

A importância da qualidade do software se manifesta em três dimensões fundamentais: confiança, funcionalidade e desempenho (PARNAS et al., 1990a). Confiança refere-se à capacidade do sistema de resistir a falhas durante a execução; funcionalidade indica se o sistema opera conforme especificado nos requisitos; e desempenho avalia se o sistema mantém tempos de resposta adequados e aceitáveis, mesmo sob carga máxima. Portanto, a implementação de testes de software eficazes é crucial para atender a essas dimensões e assegurar a segurança e a confiabilidade dos softwares.

No estudo de Menezes et al. (2018), é apresentado o Map2Check, uma

ferramenta desenvolvida para verificar automaticamente propriedades de segurança em programas escritos em C. Esta ferramenta utiliza Clang/LLVM (FANDREY, 2010) para simplificação e instrumentação de código, além de KLEE (CADAR et al., 2008) para execução simbólica baseada em caminhos. O Map2Check foca no monitoramento de ponteiros de memória e atribuições de variáveis para verificar assertivas especificadas pelo usuário, como *overflow* e segurança de ponteiros. Implementada como uma ferramenta de transformação de código-fonte em C/C++ utilizando LLVM, o Clang atua como *front-end* para análise de programas em C e geração de bitcode LLVM. O KLEE é empregado como o motor de execução simbólica baseado em caminhos, utilizando STP[1] (v2.1.2) como solucionador SMT para verificar restrições sobre vetores de bits e arrays.

O Map2Check gera entradas concretas para as funções instrumentadas realizando a execução simbólica do código analisado em LLVM IR usando KLEE. Logo, se uma propriedade de segurança for violada, um prova (*witness*) de violação é produzido para rastrear a localização do erro.

Visando contribuir com testes de software, o contexto deste trabalho está situado no uso de metodologias e técnicas de teste de software para avaliar a eficácia e eficiência do Map2Check. Para isso, utilizaremos casos de teste em linguagem C, com o objetivo de verificar a cobertura de código, eficiência e performance da ferramenta. A avaliação será realizada com base no padrão utilizado na Competição Internacional de Teste de Software (Test-Comp) (BEYER, 2023), que define um conjunto de métricas e critérios para avaliar a eficácia e eficiência de ferramentas de teste de software. A partir desses critérios, será possível avaliar o desempenho do Map2Check e compará-lo com outras ferramentas já estabelecidas no mercado.

O problema considerado neste trabalho é expresso na seguinte questão: **Como gerar casos de teste de qualidade, visando garantir critérios de uma dada cobertura do código analisado e validar propriedades de segurança de programas escritos na linguagem de programação C?**

[1] https://stp.github.io/

1.1 Objetivos

Esta seção se divide da seguinte forma: objetivos geral e os objetivos específicos. Nas próximas subseções apresentaremos ambos os objetivos que pretendemos manter como enfoque ao longo desta pesquisa.

1.1.1 Objetivo Geral

O objetivo principal deste trabalho é adaptar e avaliar o software Map2Check para geração de dados de teste utilizando o padrão da Competição de Teste de Software (Test-Comp) para analisar os casos de testes em programas na linguagem C.

1.1.2 Objetivos Específicos

Os objetivos específicos são:

- Definir uma arquitetura para o software Map2Check, visando atender as especificações da Competição de Testes de Software (Test-Comp - (BEYER, 2023));
- Especificar e analisar as regras de transformações de código necessárias para a aplicação das entradas de teste geradas por um fuzzer para simultaneamente atender a um dado critério de cobertura de código;
- Validar a aplicação do método proposto sobre benchmarks públicos de programas em C, a fim de examinar a sua eficácia e aplicabilidade.

1.2 Organização do Trabalho

Este trabalho está dividido da seguinte forma: Introdução (Capítulo 1), que contextualiza e explica o problema proposto; Fundamentação Teórica (Capí-

tulo 2), para embasar os conceitos utilizados; Trabalhos Correlatos (Capítulo 3) para observarmos lacunas de conhecimento que possamos explorar; Método da Solução Proposta (Capítulo 4), que dispõe a forma como será resolvido o problema apresentado; Planejamento de Avaliação Experimental (Capítulo 5), para estabelecermos como se dará a avaliação dos experimentos executados neste trabalho; Cronograma para o TCC 2 (Capítulo 6), que nos dará uma previsão e direcionamentos para a segunda parte deste trabalho; Considerações Parciais (Capítulo 7), que serão dadas com base no que foi executado até agora.

2 Fundamentos Teóricos

Este capítulo apresenta conceitos e definições fundamentais para o entendimento do trabalho. São abordados tópicos como Teste de Software, Fuzzing e Execução Simbólica, que são técnicas de análise de software utilizadas para identificar falhas e vulnerabilidades. A compreensão desses conceitos é essencial para a realização de testes de segurança e depuração de programas, e são fundamentais para o desenvolvimento de ferramentas de análise estática e dinâmica.

2.1 Teste de software

Softwares são sistemas cuja falha pode resultar em consequências sérias, como perda de dados, interrupção de serviços ou prejuízos financeiros. Exemplos incluem sistemas bancários, plataformas de comércio eletrônico e softwares de gerenciamento de infraestrutura. Devido à importância desses sistemas, é fundamental a aplicação de rigorosas metodologias de teste e validação para garantir sua confiabilidade e minimizar o risco de falhas críticas (PARNAS et al., 1990b).

2.1.1 Metodologias de teste

O teste de software é caracterizado por técnicas que buscam assegurar que o sistema funciona conforme o esperado, mesmo sob condições adversas. Algumas das principais metodologias incluem:

1. **Teste de Caixa Preta:** O teste de caixa preta foca nas entradas e saídas do software, sem a necessidade de conhecer a lógica interna ou o código. Ele busca garantir que o sistema funcione conforme as especificações (KHAN; SADIQ, 2011). Um exemplo prático seria testar um aplicativo bancário que

permite transferências de dinheiro entre contas. O testador insere valores, como R$ 500,00, e verifica se o saldo da conta de origem é reduzido corretamente e se o valor aparece na conta de destino. Além disso, ele testa cenários com entradas inválidas, como transferências acima do saldo, e observa se o sistema responde adequadamente com mensagens de erro.

2. **Teste de Caixa Branca:** O teste de caixa branca é baseado no conhecimento do código-fonte do software. Ele verifica a lógica interna, como fluxo de controle, condições e *loops* (OMAR; IBRAHIM, 2010). Um exemplo seria um módulo de cálculo de impostos em um *software ERP*. O testador inspeciona o código para garantir que todas as ramificações sejam cobertas, verificando se o cálculo de impostos identifica corretamente os valores de isenção e aplica as taxas corretas. Isso inclui testar diferentes cenários para confirmar que o fluxo de execução é seguido corretamente (CLARKE; WING, 1996).

3. **Teste de Robustez e Estresse:** Aqui é avaliado a resiliência e a capacidade do software de operar sob condições adversas. O teste de robustez verifica como o sistema lida com entradas inesperadas ou inválidas, garantindo que ele mantenha sua integridade e estabilidade, como no caso de um aplicativo de reservas que precisa tratar corretamente datas no passado ou números de cartão de crédito incorretos. Já o teste de estresse submete o software a cargas extremas, além dos limites normais de operação, para avaliar seu desempenho sob alta demanda, como em uma plataforma de ensino online que recebe milhares de acessos simultâneos. Juntos, esses testes asseguram que o software seja capaz de funcionar de maneira confiável, mesmo em cenários fora do comum ou sob intensa pressão (MUSTAFA et al., 2009).

2.1.2 Competição Internacional de Teste de Software - TestComp

A competição internacional de teste de software, conhecida como TEST-COMP, é um evento anual que reúne pesquisadores e profissionais de todo o mundo para avaliar e comparar diferentes técnicas e ferramentas de teste de software. O objetivo da competição é promover o avanço da tecnologia de teste de software, identificar as melhores práticas e ferramentas, e encorajar a colaboração entre a academia e a indústria (BEYER, 2021).

2.1.2.1 Estrutura da competição

A geração de testes automatizados é um processo essencial para assegurar a qualidade de softwares, permitindo a validação de funcionalidades e a detecção de falhas. Este processo envolve a criação de casos de teste de maneira não interativa, utilizando ferramentas que garantem a conformidade com especificações previamente definidas. A seguir, são descritos os passos para a geração de uma suíte de testes segundo o padrão do TestComp, desde a entrada de um programa até a validação da cobertura obtida com o uso de ferramentas de análise automatizada, ilustrando o funcionamento e a importância dessa abordagem na verificação de programas.

A tarefa de geração de testes envolve um par composto por um programa de entrada (programa a ser testado) e uma especificação de teste. Uma execução de geração de testes é a operação não interativa de um gerador de testes em uma única tarefa, com o objetivo de gerar uma suíte de testes conforme a especificação fornecida. Uma suíte de testes é uma sequência de casos de teste, apresentada como um diretório de arquivos no formato de suítes de teste intercambiáveis.

A Figura 1 demonstra o processo de execução de um gerador de suíte de testes. Um teste para um gerador de suítes de testes recebe como entrada (i) um programa da suíte de benchmarks e (ii) uma especificação de teste (cobertura de bugs ou cobertura de ramificações), e gera como saída uma suíte de testes

(ou seja, um conjunto de casos de teste). O gerador de testes é submetido por um participante da competição como um arquivo ZIP. O validador de suíte de testes recebe a suíte de testes do gerador e valida executando o programa em todos os casos de teste: para descoberta de bugs, verifica se o bug é encontrado, e para cobertura, reporta a cobertura obtida. A competição utiliza a ferramenta TestCov(BEYER; LEMBERGER, 2019)[1] como validador de suíte de testes.

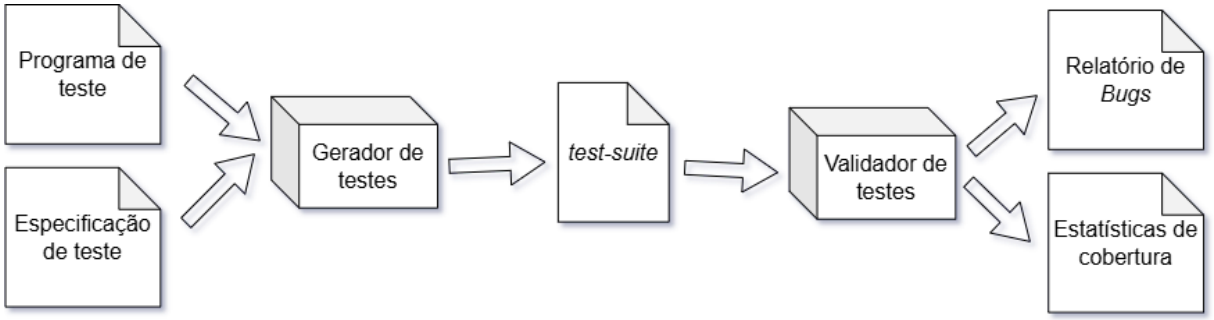

Figura 1 – Fluxo de geração de testes. Retirado de (BEYER, 2023)

2.1.2.2 Metodologia de Avaliação

Os participantes da Testcomp submetem suas ferramentas para avaliação em conjuntos de benchmarks padronizados, que incluem uma variedade de programas de teste com diferentes características e complexidades. As ferramentas são avaliadas com base em critérios como:

1. **Cobertura de Teste:** A quantidade de código testado pela ferramenta.
2. **Detecção de Falhas:** A capacidade da ferramenta de identificar defeitos no software.
3. **Eficiência:** O tempo e recursos necessários para realizar os testes.

[1] https://gitlab.com/sosy-lab/software/test-suite-validator

2.2 Fuzzing

Fuzzing, conceitualmente, envolve a geração de grandes quantidades de entradas normais e anormais para aplicações-alvo, com o objetivo de identificar exceções ao alimentar essas entradas nas aplicações e monitorar seus estados de execução (LIEW et al., 2019). Embora enfrente desvantagens como baixa eficiência e cobertura de código, o fuzzing alcança alta precisão quando realizado em condições reais (LI et al., 2018), estabelecendo-se como uma técnica eficaz para descoberta de vulnerabilidades (GODEFROID et al., 2012).

No diagrama abaixo (ver Figura 2), após a geração das entradas, os casos de teste são fornecidos aos programas-alvo. Os fuzzers iniciam e finalizam automaticamente o processo do programa-alvo e dirigem o processo de manipulação dos casos de teste pelos programas-alvo. Antes da execução, os analistas podem configurar como os programas-alvo são iniciados e finalizados, além de pré-definir parâmetros e variáveis de ambiente.

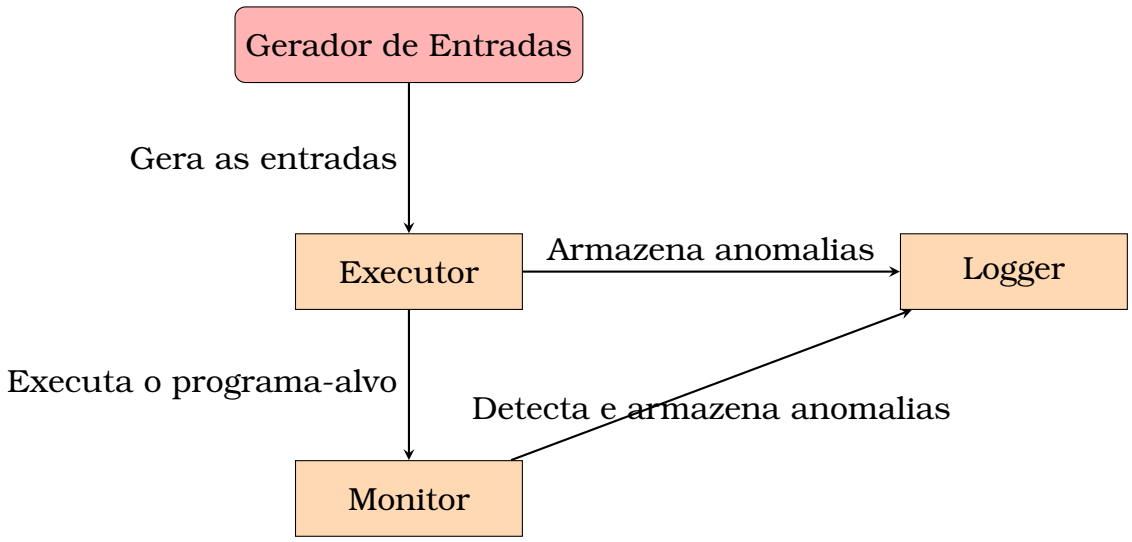

Figura 2 – Fluxo de geração de entradas

Os componentes fundamentais de um fuzzer, conforme a Figura 2, incluem o gerador de entradas, que cria os dados de teste; o executor, que executa o programa com os dados gerados; o monitor, que observa o comportamento do programa e detecta falhas; e o logger, que registra qualquer falha ou comportamento anômalo. Um teste de fuzzing começa com a criação de uma série de entradas para o programa, ou seja, casos de teste. A qualidade dos casos

de teste gerados influencia diretamente os resultados do teste. As entradas devem atender aos requisitos de formato esperados pelos programas testados o máximo possível. Ao mesmo tempo, as entradas devem ser suficientemente inválidas para que o processamento delas provavelmente cause falhas no programa. Dependendo dos programas-alvo, as entradas podem ser arquivos com diferentes formatos, dados de comunicação de rede, binários executáveis com características específicas, etc.

A geração de casos de teste suficientemente inválidos representa um desafio significativo para os fuzzers. Geralmente, dois tipos de geradores são utilizados em fuzzers modernos: geradores baseados em geração e geradores baseados em mutação.

2.2.1 Fuzzing de mutação

O fuzzing de mutação é uma técnica que altera entradas válidas existentes para gerar novas entradas de teste. Esse método começa com um conjunto de dados de entrada conhecido como corpus, que são entradas que o programa processa corretamente. O fuzzer então aplica várias mutações nessas entradas, como alterar bytes, inserir ou excluir partes da entrada, e permutar seções dos dados. A vantagem do fuzzing de mutação é que ele pode rapidamente gerar uma grande quantidade de entradas de teste a partir de um conjunto relativamente pequeno de dados iniciais, explorando assim uma vasta gama de comportamentos do programa. Esta técnica é particularmente eficaz quando há uma boa cobertura de entrada inicial, pois pode descobrir falhas sutis relacionadas a pequenas alterações nos dados (MANES et al., 2019).

A algoritmo python 2.1 exemplifica de forma simples o processo de fuzzing. Ele começa com uma entrada inicial (seed), que é alterada por mutações aleatórias em seus bytes. Cada versão mutada da entrada é passada para a função alvo, que a processa e, se encontrar um byte nulo (b'00'), levanta uma exceção. Ao repetir esse processo 100 vezes, o fuzzer verifica a robustez do

software, buscando identificar erros que ocorrem devido a alterações pequenas e aleatórias nas entradas, ilustrando como o fuzzing pode ajudar a detectar falhas e vulnerabilidades no programa.

Listing 2.1 – Exemplo simples de fuzzing de mutação em python

```
1  import random
2  def mutate(data):
3      data = bytearray(data)
4      data[random.randint(0, len(data) - 1)] = random.randint(0, 255)
5      return bytes(data)
6  def fuzz(target, seed_input):
7      for _ in range(100):
8          try:
9              target(mutate(seed_input))
10         except ValueError as e:
11             print(f"Erro encontrado: {e}")
12 def target(data):
13     if b'\x00' in data:
14         raise ValueError("Byte nulo encontrado!")
15 fuzz(target, b"hello")
```

2.2.2 Fuzzing de geração

O fuzzing de geração, por outro lado, cria entradas de teste a partir do zero com base em um modelo ou especificações detalhadas do formato de dados esperado pelo programa. Este método requer um conhecimento profundo das entradas que o programa deve processar, permitindo que o fuzzer gere dados estruturados e específicos. O fuzzing de geração é extremamente útil para testar aplicações com formatos de entrada complexos e bem definidos, como protocolos de rede ou arquivos de configuração, pois permite a criação de casos de teste altamente direcionados que podem explorar áreas específicas da lógica do programa, como Godefroid et al. (2012) afirma em seu trabalho.

Listing 2.2 – Exemplo simples de fuzzing de geração em python

```
1  import random
2  def fuzz(target, num_tests=100):
3      for _ in range(num_tests):
4          data = ''.join([chr(random.randint(48, 122)) for _ in range(random
               .randint(1, 20))])
5          try:
6              target(data)
7          except Exception as e:
8              print(f"Erro: {e} com {data}")
9  def target(data):
10     if len(data) > 10:
11         raise ValueError("Entrada muito longa!")
12 fuzz(target)
```

No algoritmo 2.2, O fuzzing de geração é ilustrado pela criação de entradas aleatórias a partir do zero. O algoritmo gera strings de comprimento variado, compostas por caracteres alfanuméricos, sem depender de entradas pré-existentes. Essas entradas geradas são passadas para a função alvo (`target`), que valida como o programa lida com dados aleatórios. A função `target` verifica o comprimento da entrada e pode levantar exceções caso a entrada seja considerada inválida, ajudando a identificar falhas ou comportamentos inesperados com dados não estruturados. Esse exemplo demonstra como o fuzzing de geração pode ser usado para explorar a robustez do software frente a uma ampla gama de entradas possíveis.

2.3 Execução Simbolica

A execução simbólica é uma técnica de análise estática que utiliza variáveis simbólicas em vez de entradas concretas para explorar todos os possíveis caminhos de execução de um programa. Essa abordagem é especialmente útil na identificação de vulnerabilidades de segurança (GODEFROID et al., 2005). Ao executar um programa com variáveis simbólicas, são geradas expressões

Característica	**Fuzzing de Mutação**	**Fuzzing de Geração**
Base de Dados	Modifica entradas válidas existentes	Gera entradas a partir do zero
Complexidade Inicial	Baixa, utiliza corpus existente	Alta, requer definição de modelos de entrada
Cobertura de Teste	Depende da qualidade do corpus inicial	Pode alcançar cobertura específica e direcionada
Aplicação	Geral, eficaz para uma ampla gama de programas	Específica, ideal para formatos de dados complexos
Tempo de Preparação	Rápido, não há modelagem de dados	Lento, devido à necessidade de modelagem
Descoberta de Falhas	Boa para falhas relacionadas a pequenas mutações	Excelente para falhas em estruturas complexas

Tabela 1 – Diferenças entre Fuzzing de Mutação e Fuzzing de Geração

simbólicas e condições de caminho que descrevem os comportamentos potenciais do software. Solucionadores de restrições são então utilizados para determinar se existem entradas que satisfaçam essas condições, auxiliando na identificação de possíveis falhas, como estouros de buffer e injeções de código (CADAR et al., 2006).

Os princípios fundamentais da execução simbólica envolvem a utilização de variáveis simbólicas para representar as entradas do programa, a formação de expressões simbólicas através das operações realizadas no código, e a coleta de condições de caminho que descrevem as possíveis execuções. Ferramentas como KLEE (CADAR; NOWACK, 2021) podem executar automaticamente estas análises, gerando casos de teste que exploram diferentes caminhos de execução. Por exemplo, em um código que verifica uma senha (ver Algoritmo 2.3), a execução simbólica pode revelar um *buffer overflow* se a entrada exceder o tamanho esperado do buffer.

Listing 2.3 – Exemplo de Código em C para Verificação de Senha

```
1 #include <stdio.h>
2 #include <string.h>
3 int main(int argc, char *argv[]) {
4     if (argc < 2) {
5         printf("Usage: %s <password>\n", argv[0]);
6         return 1;
```

```
7       }
8       if (strcmp(argv[1], "secure123") == 0) {
9           printf("Access granted!\n");
10      } else {
11          printf("Access denied!\n");
12      }
13      return 0;
14  }
```

Durante o processo de execução simbólica, as entradas do programa (neste caso, `argv[1]`) são tratadas como variáveis simbólicas. A função `strcpy`, que copia a entrada para o `buffer password`, apresenta um risco se a entrada for maior que o tamanho do `buffer` (9 caracteres mais o terminador nulo). A execução simbólica rastreia as operações realizadas com essas variáveis e gera condições de caminho que descrevem os comportamentos possíveis do programa. Utilizando um solucionador de restrições, é possível identificar se há entradas que fazem com que a condição de caminho seja verdadeira, revelando vulnerabilidades como `buffer overflows` quando a entrada excede o tamanho do `buffer`, potencialmente permitindo a execução de código malicioso.

3 Trabalhos Correlatos

Este capitulo apresenta os principais trabalhos com softwares aplicados a teste que funcionam de forma similar ao Map2check ou que executam seus testes em conformidade com o padrão internacional da competição de teste de softwares - TestComp (BEYER, 2021).

3.1 Executor Simbólico KLEE

Cadar e Nowack (2021) apresentam uma análise da ferramenta KLEE, um executor simbólico dinâmico, e os resultados dos testes feitos para o Test-Comp 2019 (BEYER, 2021).O KLEE opera no nível do *bitcode* LLVM, uma linguagem intermediária da infraestrutura de compiladores LLVM, amplamente utilizada, por exemplo, pelo compilador Clang(FANDREY, 2010).

O KLEE fornece um interpretador capaz de executar quase qualquer código representado em LLVM IR, tanto de forma concreta quanto simbólica. Uma das principais forças do KLEE é sua arquitetura modular e extensível. Por exemplo, o KLEE oferece uma variedade de diferentes heurísticas de busca para explorar o espaço de estados do programa. Uma dessas heurísticas é a busca em profundidade (Depth-First Search, DFS), que explora um caminho até o fim antes de retroceder e explorar outros caminhos. Além disso, o KLEE também utiliza heurísticas mais complexas, como a busca guiada por cobertura (Coverage-Guided Search), que prioriza caminhos que aumentam a cobertura de código, e a busca heurística baseada em custo (Cost-Based Search), que prioriza caminhos com base em uma função de custo definida(CADAR; NOWACK, 2021).

Para o Test-Comp 2019, a versão do KLEE submetida foi baseada no commit *b845877a* da branch main do repositório [1] (de janeiro de 2019). Esta versão usou as opções padrão da época, com algumas modificações específicas

[1] https://github.com/klee/klee/

para se adequar à natureza dos benchmarks do Test-Comp. Essas modificações incluíram a configuração para diferentes categorias, como `Cover-Errors` e `Cover-Branches`, assim como a extensão do software para lidar com um grande número de variáveis simbólicas e para suportar a geração de casos de teste baseados em XML. O KLEE terminou em segundo lugar nas categorias de detecção de bugs e cobertura de código do Test-Comp 2019(BEYER, 2021), demonstrando sua eficácia na geração de testes de alta cobertura e na identificação de vulnerabilidades em programas complexos. Suas fraquezas apresentadas estão ligadas ao método de execução simbólica que ele utiliza, podendo ocorrer *path explosions* e problemas com resolução de restrições. Em edições mais recentes do Test-Comp, o KLEE classifica-se em quinta posição na categoria de detecção de bugs e décimo em cobertura de código(BEYER, 2023).

3.2 FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing

O trabalho de Alshmrany et al. (2022) destaca a evolução do gerador de testes FuSeBMC para sua versão 4, com o objetivo de encontrar vulnerabilidades de segurança em programas C, focando na geração inteligente de sementes para *fuzzing* híbrido. A abordagem anterior do FuSeBMC utilizava uma combinação de *fuzzing* e verificação de modelos limitados (BMC) para produzir casos de teste. Com a nova versão, FuSeBMC v4, esse processo foi aprimorado ao integrar ambos os motores de execução para criar sementes inteligentes e usar memória compartilhada entre as sementes.

Essa melhoria é alcançada através de uma rodada inicial de execução rápida dos motores para gerar as sementes, seguida por uma execução mais longa com limites de tempo estendidos, utilizando as sementes geradas na rodada anterior. O subsistema Tracer do FuSeBMC gerencia esse processo, registrando os rótulos cobertos por cada caso de teste e avaliando os casos de

teste para identificar aqueles de alto impacto, que são convertidos em sementes para *fuzzing* subsequente. Há também a integração de memória compartilhada que permite que diferentes instâncias do *fuzzing* compartilhem informações de cobertura e estados internos, o que melhora significativamente a eficiência da busca por vulnerabilidades(ALSHMRANY et al., 2022).

Os resultados da competição Test-Comp 2022(BEYER, 2022), mostram que o FuSeBMC v4 não apenas liderou nas categorias `Cover-Branches` e `Bug-Finding`, mas também demonstrou maior eficácia em benchmarks complexos, evidenciando a capacidade de lidar com programas reais de grande escala, destacando a eficácia da combinação de *fuzzing* e verificação de modelos limitados(BMC) para a geração de testes inteligentes e a importância do teste automatizado na detecção de vulnerabilidades de segurança.

Apesar de suas capacidades avançadas, a complexidade do algoritmo de geração de sementes e a necessidade de recursos computacionais significativos para análises extensas permanecem como desafios. Estudos futuros podem focar em otimizações para reduzir o tempo de execução e o consumo de recursos, talvez através de técnicas como paralelismo distribuído ou otimizações de memória. Na edição de 2023 do Test-Comp, o FuSeBMC v4 foi classificado em primeiro lugar em todas as categorias (BEYER, 2023), mostrando que a ferramenta ainda se consolida como a melhor da competição.

3.3 ESBMC 6.1: Geração Automotizadas de Casos de Teste usando Bounded Model Checking

O trabalho de Gadelha et al. (2021) apresenta o software ESBMC v6.1, um verificador de modelos baseado em SMT capaz de verificar programas de thread único e múltiplo. Neste estudo, foram avaliadas as capacidades de verificação em programas C, bem como seu funcionamento no modo de verificação de modelo limitado incremental (incremental BMC), com foco em encontrar violações de propriedades para produzir automaticamente conjuntos

de testes (test suites). A melhoria em relação às versões anteriores está na capacidade de gerar automaticamente conjuntos de testes para programas C single-thread, permitindo a verificação de bugs específicos relacionados à acessibilidade, segurança da memória e afirmações especificadas pelo usuário.

O processo de análise de programas pelo ESBMC ocorre por meio de três componentes principais: o front-end, o conversor GOTO e o mecanismo simbólico. O front-end converte o programa em uma árvore sintática abstrata (AST). Após a geração da AST, o conversor GOTO transforma essa estrutura em um sistema de transição de estados denominado programa GOTO, que pode ser modificado para adicionar verificações de propriedades e instruções específicas de indução k (BALDONI et al., 2018). Em seguida, o mecanismo simbólico converte o programa GOTO em uma sequência de atribuições estáticas simples (SSA), desenrolando os laços e propagando constantes para gerar um conjunto mínimo de SSAs, podendo também adicionar verificações de propriedades, muitas delas relacionadas à memória alocada dinamicamente.

A codificação SMT do ESBMC converte o conjunto de SSAs em fórmulas SMT e verifica sua satisfatibilidade. Caso a fórmula seja satisfatível, o solucionador SMT é consultado para obter informações relevantes para construir o contraexemplo correspondente. O ESBMC é escrito em C++ e usa Clang(LOPES; AULER, 2014) como um de seus front-ends, suportando vários solucionadores SMT como back-ends, incluindo Boolector(BRUMMAYER; BIERE, 2009), Z3(MOURA; BJØRNER, 2008), Yices(DUTERTRE, 2014), entre outros. A versão v3.0.0 do Boolector foi utilizada para verificar a satisfatibilidade dos testes. Na competição Test-Comp 2019, segundo o trabalho de (BEYER, 2021) o ESBMC-Kind alcançou o penúltimo lugar na categoria *Cover-Error* e não participou da categoria *Cover-Branches*. Na edição de 2023, o ESBMC-Kind ficou em ultimo lugar da competição, concorrendo apenas na categoria *Cover-Error*(BEYER, 2023).

3.4 Correlações entre os trabalhos

A partir dos trabalhos apresentados nas seções anteriores, pode-se observar que ferramentas com metodologias de teste de software baseadas em fuzzing e execução simbólica se mostram eficazes na busca de erros em programas. Embora essas ferramentas tenham sido executadas em edições distintas do Test-Comp, todas participaram na edição de 2023. O Test-Comp 2023 possui características e benchmarks idênticos aos dos anos de 2020, 2021 e 2022(BEYER, 2023), proporcionando uma base de comparação consistente que pode ser usada para avaliar os trabalhos relatados e o trabalho em desenvolvimento.

Um ponto importante a ser observado a partir da Tabela 2 e dos artigos de Beyer (2021) é que, embora o ESBMC seja uma ferramenta estabelecida na verificação de software e tenha participado de muitas edições do SV-Comp(BEYER, 2013), seu desempenho em testes de software no Test-Comp é superado por ferramentas que utilizam fuzzing, execução simbólica ou uma combinação de ambos. Isso sugere que essas metodologias são mais eficazes na detecção de falhas em programas.

Observa-se que este trabalho se concentra na compreensão dessas abordagens por meio do desenvolvimento e inserção de novas ferramentas que maximizem suas vantagens sem trazer as desvantagens associadas. Com o Map2Check (ROCHA et al., 2020) adaptado ao padrão do Test-Comp, será possível avaliar se sua inclusão na lista de ferramentas padronizadas de teste de software ajudará a comunidade, oferecendo uma nova solução para encontrar falhas em softwares críticos.

KLEE	
Ano de entrada no Test-Comp	2019
Método de Verificação	Execução Simbólica Dinâmica
Posição nas Categorias	5º em Bug-Finding e 10º em Cover-Branches
Tempo de existência da ferramenta	Em desenvolvimento desde 2015
Solução de SMT (Satisfability Modulo Theory) suportada	STP, Boolector, CVC4, Z3 e Yices 2.
FuSeBMC	
Ano de entrada no Test-Comp	2021
Método de Verificação	Fuzzing + Bounded Model Checking
Posição nas Categorias	Líder em Cover-Branches e Bug-Finding
Tempo de existência da ferramenta	Em desenvolvimento desde 2020
Solução de SMT (Satisfability Modulo Theory) suportada	Z3 e Boolector (Baseado no ESBMC)
ESBMC	
Ano de entrada no Test-Comp	2019
Método de Verificação	Execução simbólica
Posição nas Categorias	Último em Bug-Finding
Tempo de existência da ferramenta	Em desenvolvimento desde 2015
Solução de SMT (Satisfability Modulo Theory) suportada	Z3 e Boolector

Tabela 2 – Comparação entre KLEE, FuSeBMC e ESBMC

4 Método da Solução Proposta

Nesta seção, detalhamos a metodologia adotada para desenvolver e implementar a solução proposta. Nosso objetivo é apresentar de maneira clara e concisa os passos, as técnicas empregadas e as justificativas para as escolhas metodológicas. Essa explicação é crucial para a reprodução dos resultados e para a validação do processo por outros pesquisadores na área.

4.1 Arquitetura

A arquitetura do Map2check é uma ferramenta de verificação de código implementada como um transformador de código escrito em C/C++ usando um *framework* para compiladores LLVM [1] (LATTNER; ADVE, 2004) (v6.0) para rastrear endereços de ponteiros e variáveis. A solução utilizará como *front-end* para programas escritos em C o Clang [2] (v6.0) (FANDREY, 2010) para gerar código LLVM-bitcode que será usado para a transformação de código. As ferramentas LibFuzzer [3] (LI et al., 2018) (v6.0) e KLEE [4] (CADAR et al., 2008) (v2.0) serão utilizadas para gerar entrada de testes para os códigos que serão analisados. Por fim, tendo o MetaSMT [5] (RIENER et al., 2017)(v4.rc2) que será usado como API para motores de solucionadores de satisfabilidade. É possível visualizar os tipos de solucionadores SMT utilizados na ferramenta pela Tabela 3, como o Z3, o yices2 e o Boolector. Adaptada para, ao final do ciclo de análise, gerar dois arquivos .XML padronizados contendo os resultados da análise. Esses arquivos serão posteriormente analisados, e as informações extraídas resultarão em um arquivo .zip em formato padronizado conforme o Test-comp.(BEYER, 2023)

[1] https://llvm.org/
[2] https://clang.llvm.org/
[3] https://llvm.org/docs/LibFuzzer.html
[4] https://klee-se.org/
[5] https://github.com/agra-uni-bremen/metaSMT

Tipo	**Comando de execução**	**Característica**
Z3	`./map2check -smt-solver z3 -target-function ./programa.c`	Bom para operações com inteiros, arrays, e bit-vectors.
Yices2	`./map2check -smt-solver yices2 -target-function ./programa.c`	Especializa-se em problemas com inteiros e álgebra linear.
Boolector	`./map2check -smt-solver btor -target-function ./programa.c`	Focado em bit-vectors e arrays.

Tabela 3 – Soluções SMT solvers no Map2Check

A abordagem de verificação do Map2Check envolve a conversão de código C em LLVM IR usando Clang, aplicando otimizações de código, adicionando funções da biblioteca Map2Check para rastrear ponteiros e assertivas no bitcode LLVM, e então conectar o código instrumentado para suportar essas funções, são feitas otimizações adicionais para melhorar a execução simbólica e provar a taxa de sucesso na detecção de erros dos códigos ou assertir propriedades falsas.

As dependências necessárias para a execução do Map2Check, como Clang e Yices, estão incluídas na distribuição da ferramenta. O módulo de informações da ferramenta no BenchExec é chamado map2check.py[6].

4.2 Fluxo de execução da ferramenta

O fluxo de execução do Map2check aplicado à competição é ilustrado através dos seguintes passos. Este fluxo representa todo o processo envolvido na execução da ferramenta. Destaca-se que o trabalho adapta a ferramenta a partir da etapa 4 do fluxo de execução:

1. Identifica a categoria o qual o código a ser testado pertence através da propriedade recebida pela linha de comando.

2. Executa o solucionador correspondente à categoria selecionada, podendo

[6] https://github.com/sosy-lab/benchexec/tree/main/benchexec/tools

ser Execução simbólica ou Fuzzing para testar a propriedade.

3. A partir dos testes feitos, cria o `witness.graphml`, contendo as informações essenciais para identificação da possível violação da propriedade.

4. Executa um script externo, que identifica as informações do `witness.graphml` selecionando suas propriedades baseado no padrão do test-comp.

5. Cria uma pasta chamada `testsuite` contendo `metadata.xml` com as informações principais do teste realizado e `testcase.xml` que possui os dados necessários para se recriar a falha encontrada.

6. Gera o `testsuite.zip` a partir das informações organizadas pela etapa anterior. Compatível com a entrada do benchmark da competição de software.

7. Executa o comando de verificação dos resultados da ferramenta através do TestCov, que recebe o arquivo da etapa anterior e o código original que foi inserido na ferramenta.

4.3 Ferramentas e Implementações

Para adaptar a ferramenta Map2Check na competição Test-Comp, é necessário padronizar suas entradas e saídas de acordo com os requisitos da competição (BEYER, 2023). Isso envolve a configuração das propriedades a serem verificadas, bem como a formatação dos resultados gerados pela ferramenta. O fluxo observado na Figura 3 mostra como funciona a entrada e saída dos arquivos durante a execução da ferramenta.

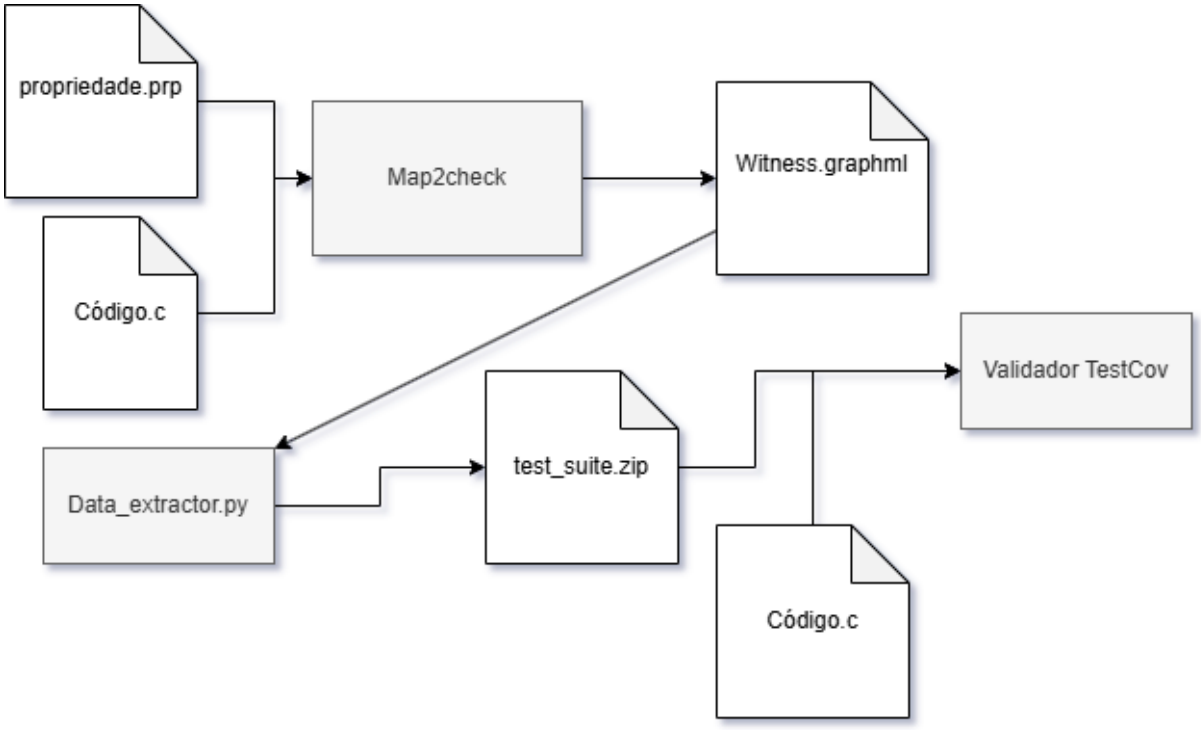

Figura 3 – Fluxo de execução da ferramenta aplicado à testes.

4.3.1 Adaptando as entradas de propriedades do Map2check para o formato do Test-Comp 2023

O map2check possui um script python que executa seu binário com as propriedades, flags e configurações mais adequadas para a categoria que está sendo avaliada. Para o Test-Comp, o script `map2check-wrapper.py` precisa definir um arquivo de propriedades (-p) e a tarefa a ser verificada. O arquivo de propriedades especifica as condições e restrições que o programa deve satisfazer durante a execução do teste. Essas propriedades são configuradas conforme os padrões do Test-Comp para as categorias *Cover Errors* e *Cover Branches*[7], com 13 e 16 subcategorias, respectivamente, conforme Tabela 4.

4.3.2 Adaptando os arquivos de saída para o test-comp

Os resultados do script `map2check-wrapper.py` são formados por um dos seguintes tipos de formatos: TRUE + witness.graphml + testsuite.zip, FALSE

[7] https://test-comp.sosy-lab.org/2023/benchmarks.php

Tabela 4 – Subcategorias na Competição de Teste de Software

Propriedade	**Subcategorias**
Cover Errors (Finding Bugs)	
`COVER ( init ( main ( ) ), FQL ( COVER EDGES ( @CALL ( reach_error ) ) ) )`	c/ReachSafety-Arrays c/ReachSafety-BitVectors c/ReachSafety-ControlFlow c/ReachSafety-ECA c/ReachSafety-Floats c/ReachSafety-Heap c/ReachSafety-Loops c/ReachSafety-ProductLines c/ReachSafety-Sequentialized c/ReachSafety-XCSP c/ReachSafety-Hardware c/SoftwareSystems-BusyBox-MemSafety c/SoftwareSystems-DeviceDriversLinux64-ReachSafety
Cover Branches (Code Coverage)	
`COVER ( init ( main ( ) ), FQL ( COVER EDGES ( @DECISIONEDGE ) ) )`	c/ReachSafety-Arrays c/ReachSafety-BitVectors c/ReachSafety-ControlFlow c/ReachSafety-ECA c/ReachSafety-Floats c/ReachSafety-Heap c/ReachSafety-Loops c/ReachSafety-ProductLines c/ReachSafety-Recursive c/ReachSafety-Sequentialized c/ReachSafety-XCSP c/ReachSafety-Combinations c/SoftwareSystems-BusyBox-MemSafety c/SoftwareSystems-DeviceDriversLinux64-ReachSafety c/SoftwareSystems-SQLite-MemSafety c/Termination-MainHeap

+ witness.graphml + testsuite.zip, ou UNKNOWN. Para cada erro identificado ou conclusão bem-sucedida, um arquivo chamado witness.graphml é gerado na raiz do projeto, contendo os dados de teste. Além disso, um arquivo zip chamado testsuite.zip é gerado, contendo dois arquivos: metadata.xml e testcase.xml. Esses arquivos são extraídos do witness.graphml e organizados para facilitar a análise dos resultados.

5 Projeto da Avaliação Experimental

Esta seção descreve o planejamento e a concepção para a execução de um estudo empírico com o objetivo de avaliar a solução proposta para a verificação de software crítico utilizando a ferramenta Map2Check. O estudo será realizado aplicando o método proposto sobre *benchmarks* públicos de programas em C, com a intenção de validar a eficácia e a aplicabilidade do método dentro dos padrões estabelecidos pela Competição de Teste de Software (Test-Comp) (BEYER, 2023).

5.1 Objetivos da Avaliação

Esta avaliação empírica terá como objetivo analisar a capacidade do Map2Check de atender aos seguintes objetivos específicos da pesquisa:

- **Definir uma arquitetura para o Map2Check**: Avaliar se a arquitetura desenvolvida está alinhada com as especificações da Competição de Teste de Software.
- **Especificar e analisar as regras de transformações de código**: Verificar a eficácia das regras de transformação de código para aplicar entradas de teste geradas por um *fuzzer* e atender a um critério de cobertura de código.
- **Validar a aplicação do método proposto**: Testar o método proposto sobre *benchmarks* públicos de programas em C para examinar sua eficácia e aplicabilidade em comparação com outras ferramentas de verificação.

5.2 Metodologia do Experimento

A execução do experimento será dividida em duas partes principais para atender aos objetivos da avaliação:

5.2.1 Primeira Parte

Como parte da pesquisa, precisamos determinar se o Map2Check é capaz de realizar os testes padronizados pela competição. Neste âmbito, serão utilizados não mais que 25 programas em C, provenientes das seguintes subcategorias retiradas da Tabela 4[1] mencionados pelo Capítulo 4:

Subcategoria	**Quantidade de testes**
c/ReachSafety-Arrays	9
c/ReachSafety-BitVectors	7
c/ReachSafety-Loops	9

Tabela 5 – Quantidade de cada entrada

Essas três categorias foram selecionados com a característica de se relacionarem diretamente com os pontos fortes do Map2check, que trabalha bem com loops, arrays e bitvectors. Ou seja, se o Map2Check conseguir executar os programas e for validado que seus testes estão em conformidade com o Test-Comp, cumpriremos o objetivo do trabalho. Cada programa da Tabela 5 será submetido no Map2Check para verificar a sua capacidade de detecção de erros e cobertura de código. O objetivo será testar a implementação da arquitetura proposta e avaliar as regras de transformação de código estabelecidas.

5.2.2 Segunda Parte

A segunda parte dos experimentos consistirá em:

1. Selecionar a ferramenta com as melhores métricas para as subcategorias dos programas de todas categorias do Test-Comp entre o ESBMC, o

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/testcomp23?ref_type=tags

FuseBMC e o Map2Check, com base em critérios como taxa de detecção de erros e cobertura de código.

2. Comparar os resultados dos testes das ferramentas(reprodução da primeira parte para cada ferramenta) a fim de validar a abordagem trabalhada do Map2Check.

Essa etapa terá como objetivo avaliar a eficácia do Map2Check na detecção de erros e na cobertura de código, comparando-o com outras ferramentas estabelecidas de verificação de software.

5.3 Coleta de Dados e Métricas de Avaliação

Durante a execução dos experimentos, serão coletados os seguintes dados e métricas para avaliar a eficácia do Map2Check:

- **Taxa de Detecção de Erros**: Número de erros encontrados pelo Map2Check em comparação com outras ferramentas como ESBMC, KLEE e FuseBMC. A taxa de detecção de erros será calculada com base na capacidade da ferramenta de identificar falhas em programas C dentro dos *benchmarks* selecionados.

- **Cobertura de Código**: Percentual de código coberto pelos testes gerados. A cobertura de código será medida utilizando ferramentas de cobertura como o *TestCov*[2] para avaliar a extensão dos testes aplicados aos programas.

- **Desempenho**: Tempo de execução e uso de recursos (CPU e memória) durante a verificação dos programas. O desempenho será avaliado registrando o tempo total de execução dos testes e o consumo de memória e CPU durante o processo de verificação.

[2] https://gitlab.com/sosy-lab/software/test-suite-validator/

5.4 Configuração Experimental do Projeto

Os experimentos para avaliar a eficiência e eficácia da ferramenta Map2Check no novo padrão da competição serão conduzidos em uma configuração controlada e replicável. A execução dos testes ocorrerá em uma máquina com as seguintes especificações:

- **Sistema Operacional**: Linux Ubuntu 22.04.5 LTS (WSL)
- **Processador**: Ryzen 7 5700G
- **Memória RAM**: 32 GB 3600MHz

Essa configuração será escolhida para garantir um ambiente de teste robusto e eficiente, adequado para a execução de testes complexos e a análise de ferramentas de verificação de software.

5.5 Descrição, Quantidade de Testes e Métricas Avaliadas

Um total de **25** testes serão executados, distribuídos igualmente entre as categorias e subcategorias definidas, com base no Test-Comp 2023 [3]. As categorias estão dispostas na Tabela 4 e o Map2Check será executado em todas as subcategorias mencionadas na Tabela 5. Durante a execução dos testes, várias métricas serão coletadas para fornecer uma avaliação completa da ferramenta:

- **Tempo Médio de Execução**: O tempo médio necessário para a conclusão de um teste, medido em segundos. Essa métrica ajudará a avaliar a eficiência da ferramenta em termos de tempo.
- **Consumo Médio de Memória**: A quantidade média de memória utilizada durante a execução dos testes, medida em megabytes. Essa métrica avaliará o uso de recursos da ferramenta.

[3] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

- **Taxa de Detecção de Erros**: A porcentagem de erros encontrados em comparação com o número total de erros esperados, com base nos benchmarks e testes realizados.
- **Cobertura de Casos de Borda**: A extensão da cobertura de casos de borda nos testes realizados, verificando a capacidade da ferramenta de identificar situações extremas e limites nos programas.

Essas métricas serão selecionadas para fornecer uma visão abrangente da performance e eficácia do Map2Check em diferentes cenários de teste e para garantir uma comparação justa com outras ferramentas de verificação. Cada etapa do planejamento experimental será cuidadosamente projetada para contribuir para a validação da ferramenta, garantindo que ela não apenas atenda aos padrões internacionais de testagem de software, mas também demonstre superioridade em termos de eficiência e eficácia em comparação com outras ferramentas concorrentes.

6 Cronograma de execução para o TCC 2

Neste capitulo é apresentado o cronograma proposto para as próximas etapas do trabalho de conclusão de curso, bem como, a descrição das atividades.

6.1 Metas e Atividades

As atividades propostas para a continuação deste trabalho são apresentadas a seguir.

1. **Avaliação experimental das novas propriedades suportadas pelo método:** Testes empíricos sobre o comportamento do experimento com as novas funcionalidades.
2. **Realização de experimentos com *benchmarks* públicos de programas escritos em C:** Com esses experimentos podemos verificar a eficácia real do método.
3. **Comparação dos resultados dos experimentos com o desempenho de outras ferramentas:** Com esta etapa, podemos comparar o desempenho do método com soluções já estabelecidas pela comunidade.
4. **Finalização da monografia:** Escrita sobre tudo o que foi desenvolvido e estudado no trabalho.
5. **Apresentação final:** Defesa do trabalho de conclusão de curso.

6.2 Cronograma

A Tabela 6 apresenta o cronograma contendo as atividades apresentadas na Seção 5.

Tabela 6 – Cronograma de atividades

Atividade	**Outubro**	**Novembro**	**Dezembro**	**Janeiro**	**Fevereiro**
Avaliação experimental das novas propriedades suportadas pelo método	×	×			
Realização de experimentos com benchmarks públicos de programas escritos em C		×	×	×	
Comparação dos resultados dos experimentos com o desempenho de outras ferramentas			×	×	
Finalização da monografia					×
Apresentação final					×

Fonte: Autor.

7 Considerações Parciais

Esta sessão é responsável por mostrar os avanços que obtivemos na pesquisa, assim como discutir observações encontradas e abordagens futuras.

7.1 Discussão, conclusão e perspectivas Futuras

Foi realizado o estudo das técnicas de *fuzzing* e execução simbólica, conforme discutido nas seções, 2.2 e 2.3. A aplicação dos métodos descritos nessas seções revelou um entendimento aprofundado das abordagens atuais para verificação de softwares. A análise mostrou que o *fuzzing* contribui para a geração de um conjunto diversificado de entradas, enquanto a execução simbólica se mostrou eficaz na identificação de erros em condições específicas de execução.

No entanto, um desafio foi identificado na adaptação da entrada da ferramenta para o padrão internacional exigido pelo Test-Comp, conforme mencionado na subseção 2.1.2 e explorado na seção 4.3. Esta adaptação é essencial para garantir que os resultados obtidos com o Map2Check sejam comparáveis aos das ferramentas competidoras. Devido a problemas de cronograma, essa adaptação ainda não foi possível realizar.

Para lidar com esta limitação e concluir o estudo, os próximos passos para avançar na pesquisa incluem:

- **Implementação Completa do Script Python**: Desenvolver um script Python que adapte as informações do arquivo de saída atual `witness.graphml` gerado pelo Map2Check para o formato exigido pelos padrões internacionais do Test-Comp. Este script será fundamental para transformar as saídas da ferramenta em dados comparáveis com os de outras ferramentas.

- **Execução da Ferramenta com Testes nos Benchmarks Públicos**: Aplicar

o Map2Check aos *benchmarks* públicos de programas em C para avaliar sua eficácia e eficiência. Os *benchmarks* serão selecionados com base nas categorias definidas pelo Test-Comp 2023 (BEYER, 2023).

- **Comparação dos Resultados de Cobertura com Outras Ferramentas de Testagem de Software**: Realizar uma análise comparativa entre o Map2Check e ferramentas competidoras, como FuseBMC e ESBMC, utilizando o TestCov como ferramenta validadora para métricas de cobertura de código e taxa de detecção de erros. A comparação ajudará a verificar a eficácia do Map2Check em relação a ferramentas estabelecidas no campo da verificação de software.

A conclusão desta etapa do projeto permitirá avançar para a fase final da pesquisa, com a expectativa de demonstrar que o Map2Check pode atender aos padrões internacionais de testagem de software e se destacar como uma ferramenta eficaz para a verificação de software crítico.

Referências

ALSHMRANY, K. M.; ALDUGHAIM, M.; BHAYAT, A.; CORDEIRO, L. C. Fusebmc v4: Smart seed generation for hybrid fuzzing. In: JOHNSEN, E. B.; WIMMER, M. (Ed.). **Fundamental Approaches to Software Engineering**. Cham: Springer International Publishing, 2022. p. 336–340. ISBN 978-3-030-99429-7.

BALDONI, R.; COPPA, E.; D'ELIA, D. C.; DEMETRESCU, C.; FINOCCHI, I. A survey of symbolic execution techniques. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 51, n. 3, 2018.

BEYER, D. Second competition on software verification. In: PITERMAN, N.; SMOLKA, S. A. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 594–609. ISBN 978-3-642-36742-7.

BEYER, D. First international competition on software testing. **International Journal on Software Tools for Technology Transfer**, Springer, v. 23, n. 6, p. 833–846, December 2021. Disponível em: <https://doi.org/10.1007/s10009-021-00613-3>.

BEYER, D. Advances in automatic software testing: Test-comp 2022. In: JOHNSEN, E. B.; WIMMER, M. (Ed.). **Fundamental Approaches to Software Engineering**. Cham: Springer International Publishing, 2022. p. 321–335. ISBN 978-3-030-99429-7.

BEYER, D. Software testing: 5th comparative evaluation: Test-comp 2023. In: LAMBERS, L.; UCHITEL, S. (Ed.). **Fundamental Approaches to Software Engineering**. Cham: Springer Nature Switzerland, 2023. p. 309–323. ISBN 978-3-031-30826-0.

BEYER, D.; LEMBERGER, T. Testcov: Robust test-suite execution and coverage measurement. In: **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2019. p. 1074–1077.

BRUMMAYER, R.; BIERE, A. Boolector: An efficient smt solver for bit-vectors and arrays. In: KOWALEWSKI, S.; PHILIPPOU, A. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p. 174–177. ISBN 978-3-642-00768-2.

CADAR, C.; DUNBAR, D.; ENGLER, D. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: **Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation**. USA: USENIX Association, 2008. (OSDI'08), p. 209–224.

CADAR, C.; GANESH, V.; PAWLOWSKI, P. M.; DILL, D. L.; ENGLER, D. R. Exe: automatically generating inputs of death. In: **Proceedings of the 13th ACM Conference on Computer and Communications Security**. New York, NY, USA: Association for Computing Machinery, 2006. (CCS '06), p. 322–335. ISBN 1595935185. Disponível em: <https://doi.org/10.1145/1180405.1180445>.

CADAR, C.; NOWACK, M. Klee symbolic execution engine in 2019. **International Journal on Software Tools for Technology Transfer**, Springer, v. 23, n. 6, p. 867–870, December 2021. Disponível em: <https://doi.org/10.1007/s10009-020-00570-3>.

CLARKE, E. M.; WING, J. M. Formal methods: state of the art and future directions. **ACM Computing Surveys**, Association for Computing Machinery, New York, NY, USA, v. 28, n. 4, p. 626–643, dec 1996. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/242223.242257>.

DUTERTRE, B. Yices 2.2. In: BIERE, A.; BLOEM, R. (Ed.). **Computer Aided Verification**. Cham: Springer International Publishing, 2014. p. 737–744. ISBN 978-3-319-08867-9.

FANDREY, D. **Clang/LLVM Maturity Report**. 2010. <http://www.iwi.hs-karlsruhe.de>. Accessed: 2024-06-16.

GADELHA, M. R.; MENEZES, R. S.; CORDEIRO, L. C. Esbmc 6.1: automated test case generation using bounded model checking. **International Journal on Software Tools for Technology Transfer**, Springer, v. 23, n. 6, p. 857–861, December 2021. Disponível em: <https://doi.org/10.1007/s10009-020-00571-2>.

GODEFROID, P.; KLARLUND, N.; SEN, K. Dart: directed automated random testing. In: **Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 2005. (PLDI '05), p. 213–223. ISBN 1595930566. Disponível em: <https://doi.org/10.1145/1065010.1065036>.

GODEFROID, P.; LEVIN, M. Y.; MOLNAR, D. Sage: whitebox fuzzing for security testing. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 55, n. 3, p. 40–44, mar 2012. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/2093548.2093564>.

HE, Y. The course choice between c language and c++ language. In: **2009 4th International Conference on Computer Science Education**. [S.l.: s.n.], 2009. p. 1588–1590.

KHAN, M. A.; SADIQ, M. Analysis of black box software testing techniques: A case study. In: **The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11)**. [S.l.: s.n.], 2011. p. 1–5.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: **Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)**. Palo Alto, California: [s.n.], 2004.

LEVESON, N. G.; TURNER, C. S. An investigation of the therac-25 accidents. **Computer**, IEEE, v. 26, n. 7, p. 18–41, 1993.

LI, J.; ZHAO, B.; ZHANG, C. Fuzzing: a survey. **Cybersecurity**, Springer, v. 1, n. 1, p. 6, June 2018. Disponível em: <https://doi.org/10.1186/s42400-018-0002-y>.

LIEW, D.; CADAR, C.; DONALDSON, A. F.; STINNETT, J. R. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 521–532. ISBN 9781450355728. Disponível em: <https://doi.org/10.1145/3338906.3338921>.

LOPES, B. C.; AULER, R. **Getting Started with LLVM Core Libraries**. [S.l.]: Packt Publishing, 2014. ISBN 1782166920.

MANES, V. J. M.; HAN, H.; HAN, C.; CHA, S. K.; EGELE, M.; SCHWARTZ, E. J.; WOO, M. **The Art, Science, and Engineering of Fuzzing: A Survey**. 2019.

MENEZES, R.; ROCHA, H.; CORDEIRO, L.; BARRETO, R. Map2check using llvm and klee. In: BEYER, D.; HUISMAN, M. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Cham: Springer International Publishing, 2018. p. 437–441. ISBN 978-3-319-89963-3.

MOURA, L. de; BJØRNER, N. Z3: An efficient smt solver. In: RAMAKRISHNAN, C. R.; REHOF, J. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 337–340. ISBN 978-3-540-78800-3.

MUSTAFA, K. M.; AL-QUTAISH, R. E.; MUHAIRAT, M. I. Classification of software testing tools based on the software testing methods. In: **2009 Second International Conference on Computer and Electrical Engineering**. [S.l.: s.n.], 2009. v. 1, p. 229–233.

OMAR, F.; IBRAHIM, S. Designing test coverage for grey box analysis. In: **2010 10th International Conference on Quality Software**. [S.l.: s.n.], 2010. p. 353–356.

PARNAS, D. L.; SCHOUWEN, A. J. van; KWAN, S. P. Evaluation of safety-critical software. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 6, p. 636–648, jun 1990. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/78973.78974>.

PARNAS, D. L.; SCHOUWEN, A. J. van; KWAN, S. P. Evaluation of safety-critical software. **Communications ACM**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 6, p. 636–648, jun 1990. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/78973.78974>.

RIENER, H.; HAEDICKE, F.; FREHSE, S.; SOEKEN, M.; GROSSE, D.; DRECHSLER, R.; FEY, G. metasmt: focus on your application and not on solver integration. **International Journal on Software Tools for Technology Transfer**, Springer Science and Business Media LLC, v. 19, n. 5, p. 605–621, 2017. Disponível em: <https://doi.org/10.1007/s10009-016-0426-1>.

ROCHA, H.; MENEZES, R.; CORDEIRO, L. C.; BARRETO, R. Map2check: Using symbolic execution and fuzzing. In: BIERE, A.; PARKER, D. (Ed.). **Tools and Algorithms for the Construction and Analysis of Systems**. Cham: Springer International Publishing, 2020. p. 403–407. ISBN 978-3-030-45237-7.