UFRR

UNIVERSIDADE FEDERAL DE RORAIMA

PRÓ-REITORIA DE ENSINO E EXTENSÃO

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

RAFAEL SÁ MENEZES

VERIFICAÇÃO DE PROPRIEDADES DE GERENCIAMENTO DE MEMÓRIA DE PROGRAMAS EM C BASEADO EM TRANSFORMAÇÕES DE CÓDIGO

Boa Vista - RR

2017

Dados Internacionais de Catalogação na publicação (CIP)
Biblioteca Central da Universidade Federal de Roraima

M543v

Menezes, Rafael Sá.

Verificação de propriedades de gerenciamento de memória de Programas em C baseado em transformações de código / Rafael Sá Menezes. – Boa Vista, 2017.

74 f. : il.

Orientador: Prof. Dr. Herbert Oliveira Rocha.

Monografia (graduação) - Universidade Federal de Roraima, Curso de Ciência da Computação.

1- Verificação de software. 2- Segurança de memória. 3- Programa em C. 4- Transformação de código. I- Título. II- Rocha, Herbert Oliveira (orientador).

CDU 681.3

RAFAEL SÁ MENEZES

VERIFICAÇÃO DE PROPRIEDADES DE GERENCIAMENTO DE MEMÓRIA DE PROGRAMAS EM C BASEADO EM TRANSFORMAÇÕES DE CÓDIGO

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

Orientador: Dr. Herbert Oliveira Rocha

Boa Vista - RR

2017

RAFAEL SÁ MENEZES

VERIFICAÇÃO DE PROPRIEDADES DE GERENCIAMENTO DE MEMÓRIA DE PROGRAMAS EM C BASEADO EM TRANSFORMAÇÕES DE CÓDIGO

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Roraima como requisito para a obtenção do grau de Bacharel em Ciência da Computação.
Apresentada em 02 de Agosto de 2017, a seguinte banca examinadora:

Prof. Dr. Herbert Oliveira Rocha
Orientador / Curso de Ciência da Computação - UFRR

Prof. MSc. Felipe Leite Lobo
Curso de Ciência da Computação - UFRR

Prof. MSc. Miguel Raymundo Flores Santibañez
Curso de Ciência da Computação - UFRR

Este trabalho é dedicado a meu irmão Lucas que,
me mostrou seu jeito único de pensar.

AGRADECIMENTOS

Primeiramente, quero agradecer a minha mãe Ana Paula Sá Menezes por sempre me apoiar em todas as minhas decisões e sempre me aconselhando, mesmo nos momentos difíceis e sempre visando meu futuro.

Gostaria de agradecer ao meu orientador, professor Herbert Oliveira Rocha, por sempre estar dando seu máximo para me auxiliar, aconselhar e incentivar.

Aos professores Felipe Leite Lobo e Miguel Raymundo Flores Santibañez, por participarem da minha banca no trabalho de conclusão de curso.

À professora Delfa Mercedes Huatuco Zuasnabar por ter me escolhido no projeto do MobileApps, onde pude ter minha primeira experiência com desenvolvimento móvel, o que foi algo muito útil ao decorrer do curso.

À Universidade Federal de Roraima por ter me cedido diversas bolsas, graças a isso pude me dedicar mais as atividades acadêmicas.

"Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2)

RESUMO

Um dos principais desafios durante o desenvolvimento de um sistema é garantir a corretude de um *software*, uma vez que, em sistemas críticos um falha pode causar situações catastróficas. Pela gravidade disso, a verificação de *software* é fundamental. Esse trabalho propõe melhorias na ferramenta Map2Check v6 que é um método de automático para gerar casos de teste e verificar propriedades de segurança de memória em programas escritos em C. A nova versão, nomeada Map2Check v7, utiliza execução simbólica para geração dos casos de teste, *framework* LLVM para transformação do código e uma biblioteca para verificar propriedades de memória. Para medir a eficácia do novo método foi feita uma análise empírica utilizando um *benchmark* público da *Competition on Software Verification* (SV-COMP) de verificação de *software* e os resultados são comparados com algumas ferramentas e o Map2Check v6. Os resultados experimentais mostraram que o Map2Check v7 foi eficaz em validar propriedades de segurança relacionadas a gerenciamento de memória, além de não apresentar nenhum resultado inválido. Ao final do trabalho são sugeridos alguns trabalhos futuros.

Palavras-chaves: verificação de software, segurança de memória, programa em C, transformação de código, model checking.

ABSTRACT

One of the main challenges in the system development is to ensure the correctness of the software, for instance, in the critical systems a failure can cause catastrophic situations. Therefore, the software verification plays an important role. This study proposes enhancements in Map2Check v6 tool, a automatic method for test case generation and memory safety propriety validator for C programs. The proposed tool, Map2Check v7, now uses symbolic execution for test case generation, LLVM for code transformation, and a library for memory safety verification. To mesure the method effectiveness it was performed an empirical evaluation using a public benchmark from the Competition on Software Verification (SV-COMP) for software verification, and compared the results with the other tools and Map2Check v6. The experiment results showed that Map2Check v7 was the best tool in proving program correctness and second in finding program violation, and Map2Check v7 did not generated any false positive or false negative. Finally, in the end of this work some future works are presented.

Keywords: software verification, memory safety, C program, code transformation, model checking.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

V&V	Verificação e Validação de *Software*
LLVM	Low-Level Virtual Machine
CC	Condição de Caminho
PC	Program Counter
BMC	Bounded Model Checking
CV	Condição de Verificação
GAD	Grafo Acíclico Direcionado
ESBMC	Efficient SMT-Based Context-Bounded Model Checker
CFG	Grafo de Controle de Fluxo
SV-COMP	Competition on Software Verification
LTL	Lógica Temporal Linear
LAC	Lógica de Árvore de Computação
DBA	*Dynamic Binary Analysis*
SMT	*Satisfability Modulo Theories*
CLI	*Command-Line Interface*
SSA	*Static Single Assignment*

SUMÁRIO

1 INTRODUÇÃO

Com o aumento da complexidade dos sistemas computacionais, diversas companhias e organizações estão rotineiramente lidando com software que contêm milhares de linhas de código, escritos por diferentes pessoas, e que usam diferentes linguagens, ferramentas e estilos (HODER et al., 2011). Adicionalmente, estes sistemas de software, devido ao curto espaço de tempo de liberação do produto ao mercado, precisam ser desenvolvidos rapidamente e atingir um alto nível de qualidade. Porém, os programadores cometem enganos.

Isso pode ser visto quando um programador se equívoca ao escrever um requisito do sistema, como a alteração de uma condição de $x \leq 10$ para $x < 10$, tempo e esforço são gastos para encontrar e corrigir estes tipos de erro (GUPTA; RYBALCHENKO, 2009). Como consequência destes fatores, os erros durante o desenvolvimento de software se tornam mais comuns. Desta forma, faz-se necessário que as aplicações sejam projetadas tomando em consideração os requisitos de previsibilidade e confiabilidade, em aplicações de sistemas críticos esses requisitos devem ser ainda mais restritos, onde diversas restrições (como tempo de resposta e precisão dos dados) devem ser atendidas e mensuradas de acordo com os requisitos do usuário, caso contrário uma falha pode conduzir a situações catastróficas (ROCHA et al., 2015). Por exemplo, o erro de cálculo da dose de radiação no Instituto Nacional de Oncologia do Panamá que resultou na morte de 23 pacientes (WONG et al., 2010; MERZ et al., 2012).

Inspecionar manualmente um *software* é uma tarefa complexa, assim, é comum defeitos no software passarem despercebidos que podem interromper o seu funcionamento. Por exemplo, em sistemas embarcados com uso continuo que podem gerar vários estados possíveis de execução. Uma forma de evitar isso é a utilização de técnica de verificação formal automatizadas, através delas é possível verificar o sistema garantindo a qualidade do *software*, pois através de métodos formais é possível gerar todos os casos de testes necessários para validação do sistema (D'SILVA et al., 2008). Teste é um modo objetivo de verificar a corretude da implementação de um sistema, detectando algum comportamento inesperado em um caminho, testes não garantem que todo o programa está funcionando, apenas garantem que o programa está correto para determinado conjunto de testes. Métodos formais são considerados a forma rigorosa de verificar *software* e analisar os sistemas com mais precisão, pois são capazes de gerar um conjunto de testes ideal para analisar o sistema (DING et al., 2008).

A verificação formal tem desempenhado um papel importante para assegurar a confiabilidade e a qualidade no desenvolvimento de aplicações críticas, como *softwares* para piloto automático de aviões. Segundo Bensalem e Lakhnech (1999), *model checking* é uma técnica baseada em métodos formais utilizados para provar propriedades de programas. O *model checking* é uma técnica automatizada que, dado um modelo de estados finitos de um sistema e uma pro-

priedade formal, sistematicamente checa se a propriedade é verdadeira para o modelo (BAIER; KATOEN, 2008). Esta técnica gera uma busca exaustiva no espaço de estados do modelo para determinar se uma dada propriedade é válida ou não (BAIER; KATOEN, 2008). A principal razão para o sucesso da técnica *model checking* se dá por funcionar de maneira automática, não havendo necessidade da intervenção do usuário. Entretanto, a técnica *model checking* ainda possui algumas dificuldades, tais como: lidar com a explosão do espaço de estados do modelo, integração com outros ambientes de testes e tratamento e análise de contra-exemplos (CLARK, 2008).

Este trabalho visa aperfeiçoar o método Map2Check (ROCHA et al., 2015) que visa a geração automática de casos de testes para a verificação de gerenciamento de memória de programas escritos em C, sendo a geração desses testes baseadas em assertivas extraídas de propriedades de seguranças geradas por ferramentas de *Bounded Model Checking*. No trabalho do Map2Check, Rocha et al. (2015) comentam sobre a dificuldade em se fazer a análise diretamente em códigos fontes escritos em C e indica a utilização de uma representação intermediário do código do programa ou o uso de instrumentação em código binário nos seus trabalhos futuros, como uma solução para o problema, outro problema com o uso da linguagem C é percorrer o código com estruturas de alto nível, sendo complexo adicionar execução simbólica de forma automática. Neste sentido, as melhorias para o método Map2Check consiste em adicionar: 1) a ferramenta Clang como *front-end* para programas em C (LLVM Foundation, 2017); 2) o *framework* LLVM como base para aplicações de transformações de código utilizando a representação intermediária LLVM *bitcode* (LLVM Foundation, 2017); e 3) o Klee para execução simbólica de código baseado em LLVM (CADAR et al., 2008);

O *Low-Level Virtual Machine* (LLVM) é um *framework* de compilação cujo objetivo é fazer análises de programas e transformações que fiquem disponível para *softwares* arbitrários de uma maneira que seja transparente ao programa original, por meio de: (a) o uso de uma representação intermediária (LLVM IR) de código capaz de descrever ações de análise e transformação de forma simples; e (b) uma infraestrutura de compilação capaz de tirar o maior proveito dessa linguagem (LATTNER; ADVE, 2004). O *framework* LLVM é utilizado por empresas como *Apple Inc.*, *Intel*, *NVIDIA*, *Sony Interactive Entertainment* (LLVM Foundation, 2009) e *Google* (LLVM Foundation, 2011) por ser uma plataforma estável e robusta. Kim et al. (2015) utiliza LLVM para fazer transformações de código (para otimizações) Android.

O aprimoramento do Map2Check (ROCHA et al., 2015) neste trabalho, tem como base métodos adotados em trabalhos como o LEAKPOINT (CLAUSE; ORSO, 2010) e o SYMBIOTIC 3 (CHALUPA et al., 2016) que exploram o uso de técnicas de compiladores para analisar códigos-fonte em C para fazer a instrumentação dos mesmos. A diferença para as versões anteriores do Map2Check será na utilização de código intermediário usando LLVM IR, ou seja, a partir de um programa em C, será gerado um código LLVM IR, onde faremos a instrumentação do mesmo e assim poderemos validar as propriedades de segurança desejadas, como desalocações inválidas e

vazamentos de memória. Utilizar uma linguagem intermediária tem a vantagem de que embora o foco do projeto seja a linguagem C, qualquer código em uma linguagem de programação que for compilada para LLVM IR, poderá ser analisado pelo Map2Check.

1.1 Motivação

A grande quantidade de áreas distintas para sistemas de *software* de alto risco cria uma necessidade de um alto grau de confiabilidade. Uma falha em um sistema ambiental, militar, médico ou aerospacial pode criar um problema catastrófico. Sendo assim, um dos motivos deste trabalho é a necessidade de garantir a corretude de sistemas, que é acompanhada pelo aumento da complexidades dos mesmo, ao mesmo tempo em que, por pressão econômica os prazos de entrega diminuem. O aumento da complexidade juntamente com a diminuição no prazo tornam mais difícil a tarefa de verificação do sistema a ser entrega. Isto motiva a pesquisa por formas automáticas de verificação e correção de *software* (D'SILVA et al., 2008).

Um tipo de falha que pode impactar na performance e na corretude da aplicação é vazamentos de memória. Em linguagens de programação como C, uma função de alocação reserva um espaço previamente livre de memória (chamaremos de `m` esse espaço) e retornam um ponteiro (chamaremos de `p` esse ponteiro) que aponta para o primeiro endereço desse espaço alocado. Normalmente, um programa armazena e somente então utiliza `p`, ou algum outro ponteiro derivado de `p`, para interagir com `m`. Quando `m` não for mais necessário, o programa deverá mandar `p` para uma função de desalocação. Um vazamento ocorre se, por algum erro durante o gerenciamento da memória, `m` não é desalocado no tempo correto (CLAUSE; ORSO, 2010).

Os vazamentos por não gerarem um erro fatal (onde um sistema para sua execução) muitas vezes passam despercebidos (em 2009 mais de 100 vazamentos de memória foram reportados no navegador Firefox), tipicamente esse vazamentos só são detectados após eles consumirem uma grande parte da memória do sistema, causando assim impacto em todas as aplicações rodando no sistema. Por essas sérias consequência e ocorrências comuns, muitas técnicas foram criadas para os detectar, porém em muitas situações não são técnicas simples de se aplicar (CLAUSE; ORSO, 2010).

1.2 Definição do Problema

A verificação de gerenciamento de memória é uma tarefa importante para evitar comportamentos inesperados de programas, por exemplo, uma violação na propriedade de segurança de um ponteiro resulta em um endereço errado, que pode acabar produzindo uma saída incorreta do programa e não necessariamente um erro. Um vazamento de memória não produz nenhum sintoma que seja visível facilmente ou detectado imediatamente como um erro ou a saída de um

valor errado. Entretanto, vazamentos de memória geralmente continuam sem ser observados até consumirem uma grande porção da memória disponível no sistema, com isso, pode-se gerar em um impacto negativo em outras aplicações que estiverem sendo executadas no mesmo sistema (CLAUSE; ORSO, 2010). Devido as sérias consequências e ocorrências comuns de erros de gerenciamento de memória, ainda existem campos de pesquisa aberto para aprimorar a detecção de error (ROCHA et al., 2015)

O problema considerado neste trabalho é expresso na seguinte questão: **Como complementar e aprimorar a verificação de propriedades de segurança de memória, com foco em aritmética de ponteiros e vazamentos de memória, com aplicação na linguagem de programação C?**

1.3 Objetivos

O objetivo principal deste trabalho é aprimorar um método para a verificação de propriedades de segurança de memória em software escritos na linguagem de programação C, por meio de transformações de código e uso da técnica *model checking*.

Os objetivos específicos são:

1. Demonstrar melhorias em um método de geração e verificação de casos de teste estruturais, baseado nas propriedades de segurança de gerenciamento de memória.
2. Propor uma técnica para instrumentação de programas escrito em C, adotando técnicas de compiladores como análise de representações intermediaria de código.
3. Propor uma técnica baseada em execução simbólica para gerar dados de teste e identificação de localizações de erro em programas escritos em C.
4. Validar a aplicação dos métodos sobre *benchmarks* públicos de programas em C, a fim de examinar a sua eficácia e aplicabilidade.

1.4 Metodologia Proposta

Esta seção descreve as principais etapas que foram identificadas para alcançar os objetivos desta proposta de projeto. Estas etapas fornecem os passos necessários e direções para desenvolver o método proposto e podem ser descritas em três diferentes fases como segue: análise do domínio, metodologia proposta e validação da metodologia. Na etapa de análise de domínio, toda a teoria necessária para entender os métodos, técnicas e ferramentas aplicadas à verificação e teste de software são analisadas e avaliadas. Na fase da metodologia proposta, uma versão inicial da metodologia para o desenvolvimento do método proposto, tem como foco uma ou mais restrições, e seu escopo serão precisamente definidos e propostos. Depois disso, esta

metodologia é mais adiante refinada na fase de validação aplicando-a na verificação dos estudos de caso, visando uma avaliação experimental da solução proposta.

Com o intuito de realizar as atividades deste trabalho, uma abordagem iterativa e incremental é usada com o propósito de reduzir riscos e incertezas. Sendo assim, para cada incremento do projeto do trabalho, as três etapas nomeadas neste trabalho como análise do domínio, metodologia proposta e validação da metodologia podem ser tratadas com diferentes ênfases em cada fase do trabalho.

Por exemplo, no início do trabalho, a análise de domínio provavelmente terá maior ênfase do que as outras fases metodologia proposta e validação da metodologia. Na metade do trabalho, a fase de metodologia proposta provavelmente terá mais ênfase do que as outras duas fases. Finalmente, a fase de validação da metodologia provavelmente terá mais ênfase no fim do projeto de pesquisa. A principal razão para adotar uma abordagem iterativa e incremental é desenvolver o projeto incrementalmente, permitindo assim tirar vantagem do que foi aprendido durante cada incremento do projeto (SCHWABER; BEEDLE, 2002).

1.5 Contribuições Propostas

As contribuições proposta por esse trabalho são:

1. A implementação e avaliação de um método para a verificação e teste de programas em C. O método proposto gera automaticamente casos de teste baseada em propriedades de segurança geradas através da instrumentação de código. Essas propriedades estão relacionadas à gerenciamento de memória e alcançabilidade de estados no programa.

2. Este trabalho apresenta para o método proposto, o desenvolvimento e implementação de um rastreador de endereços de memória baseado em representação intermediária (LLVM-IR) para auxiliar na verificação das propriedades do programa. Assim, esta contribuição auxilia na integração de técnicas formais de forma automática.

3. Este trabalho visa contribuir com método Map2Check (ROCHA et al., 2015), consiste em adicionar: 1) a ferramenta Clang como *front-end* para programas em C (LLVM Foundation, 2017); 2) o *framework* LLVM como base para aplicações de transformações de código utilizando a representação intermediária LLVM *bitcode* (LLVM Foundation, 2017); e 3) o Klee para execução simbólica de código baseado em LLVM (CADAR et al., 2008).

1.6 Organização do Trabalho

A introdução deste trabalho apresentou: o contexto, definição do problema, motivação, objetivos, metodologia e contribuições dessa pesquisa. Os capítulos restantes são organizados da seguinte forma:

No Capítulo 2 **Conceitos e Definições**, são apresentados os conceitos abordados neste trabalho, especificamente: Verificação e Teste de *Software*, Propriedades de Segurança, *Bounded Model Checking*, Execução Simbólica, Propriedade de Segurança, Análise de Contra-Exemplos e Técnicas de Compiladores.

No Capítulo 3 **Trabalhos Correlatos**, são analisados os trabalhos correlatos ao método Map2Check.

No Capítulo 4 **Método Proposto**, é descrito as etapas de execução do novo método proposto ao Map2Check. Em especial, são descritos: Geração da representação intermediária; Biblioteca para rastreamento de endereços de memória e assertivas; Instrumentação de funções; e Verificação do programa analisado.

No Capítulo 5 **Resultados Experimentais**, descreve-se a execução de uma avaliação experimental sobre o método proposto, bem como, uma análise dos resultados obtidos, onde comparamos o novo Map2Check com outras ferramentas.

E por fim no Capítulo 6 **Conclusões e trabalhos futuros**, apresenta-se as considerações parciais e os trabalhos futuros a serem desenvolvidos.

2 CONCEITOS E DEFINIÇÕES

Este capítulo tem como objetivo apresentar os principais conceitos e definições abordados neste trabalho, tais como: Verificação e Validação de *Software*, Propriedades de Segurança, *Bounded Model Checking*, Execução Simbólica, Propriedade de Segurança, Análise de Contra-Exemplos e Técnicas de Compiladores.

2.1 Verificação e Teste de *Software*

Os computadores modernos consistem de componentes complexos de *hardware* e *software*, garantir a corretude do *software* geralmente é muito mais complexo do que o *hardware* (D'SILVA et al., 2008), principalmente que controla nosso transporte ou infraestrutura (Neumann (2015) documenta centenas de casos onde falhas de *software* causaram problemas). Verificação e Validação de Software (V&V) são áreas da Engenharia de *Software* que auxiliam no desenvolvimento de *software* de qualidade. V&V verificam e testam *software* para determinar se ele funciona adequadamente, de acordo com os seus respectivos requisitos (WALLACE; FUJII, 1989). A parte de testar o *software* constitui uma parte significativa do projeto de engenharia. Entre 30% e 50% dos custos do projeto são voltados a teste. Testar é pegar uma parte do *software* em consideração e prover o código compilado com entradas, chamadas de testes. A principal diferença entre teste e verificação é o fato de teste apenas garantir a corretude para o conjunto de casos testados enquanto a verificação utiliza modelagem matemática para garantir a corretude do *software*.

2.1.1 Verificação formal

Métodos formais podem ser considerados como: Matemática aplicada para modelagem e análise de sistemas de informação (BAIER; KATOEN, 2008). Segundo Clarke et al. (2009), verificação formal busca validar a corretude de um programa utilizando lógica matemática. Neste contexto, um programa é um objeto matemático (usualmente composto por uma série de conjuntos) com um comportamento definido, assim a lógica matemática pode ser utilizada para descrever precisamente o que é um comportamento correto do programa. Isso torna possível utilizar modelos matemáticos de um programa e assim provar a corretude de seu comportamento.

A criação de métodos automatizados para validar a corretude de programas tem um impacto significante, visto que construção manual de uma prova de corretude pode ser complexa e essa prova manual pode não funcionar corretamente para programas grandes, devido enumeração exaustiva de estados em sistemas complexos. Um sistema de transições pode ser considerado um método formal como será explicado na Seção 2.5.

2.1.2 Testes de *Software*

Teste de *software* são métodos para checar a corretude da implementação de um sistema o experimentando, basicamente é o processo de executar um *software* procurando comportamentos incorretos. Testes não garantem observar todos os comportamentos de um programa, porém, se o programa contiver apenas um caminho, é possível (DING et al., 2008). A corretude então é determinado por forçar o programa a percorrer um conjunto de caminhos de execução que abragem a maior cobertura de execução do programa sem resultar na violação de propriedades (BAIER; KATOEN, 2008).

Um tipo de teste seria o teste unitário, que é o processo de testar componentes de um programa, como métodos e classes de objetos. Funções e métodos são o tipo mais simples de componentes. Os testes devem ser chamadas dessas rotinas com diferentes parâmetros de entrada (PRESSMAN, 2001). Para exemplificar esse teste, tomemos a situação em uma loja que vende batatas ao quilo, o preço normal é de R$1,5/kg, porém para compras acima de 50kg, o preço fica R$1,25/kg, ao desenvolver uma função que calculasse o custo de uma compra. Os casos de teste deveriam ser entradas da função, nesse exemplo, a entrada seria a quantidade de quilogramas, como: $0, 49$ e 51. A Figura 1 exibe um exemplo de implementação de uma função de custo, caso alguma entrada inválida seja feita, é retornado -1, os casos de teste também foram implementados.

Figura 1 – Exemplo de teste

```
1  double custo(double entrada)
2  {
3    if (entrada < 0)
4    {
5      return -1;
6    }
7    if ((entrada * 1.5) <= 0))
8    {
9      return -1;
10   }
11   if(entrada > 50)
12   {
13     return 1.25*entrada;
14   }
15   return 1.5*entrada;
16 }
```

(a) Código C

```
1  TEST_CASE_1()
2  {
3    double result;
4    result = custo(0.49)
5    assert(result == 0.735);
6  }
7  TEST_CASE_2()
8  {
9    double result;
10   result = custo(51)
11   assert(result == 63.75);
12 }
```

(b) Código de Teste

Fonte: Própria.

Segundo Ding et al. (2008) métodos formais podem ser utilizados em teste de *software*, os casos de teste podem ser gerados com eficiência e eficácia quando derivados de especificações formais. Na prática, qualidade nem sempre se refere a um *software* totalmente correto em projeto, pois isso aumenta o custo de produção consideravelmente. Porém, garantir que falhas específicas

estarão ausentes é uma tarefa realística e é considerada uma boa métrica para garantir qualidade (D'SILVA et al., 2008).

2.2 Execução Simbólica

A ideia da execução simbólica é a utilização de valores simbólicos que são valores expressos em fórmulas que possam gerar os valores (como $0 \leq x \leq 10$, podendo ser utilizado qualquer valor real que resolva essa inequação), ao invés de valores reais, para representação das variáveis dos programa. Como resultado, os valores de saída de um programa podem ser representado como uma função cujo domínio são os valores simbólicos de entrada (KHURSHID et al., 2003).

A execução simbólica, em verificação e teste de *software*, é uma técnica para gerar dados de entrada de programas utilizando valores simbólicos ao invés de valores concretos e assim representar os valores das variáveis com expressões simbólicas. Em teste de *software*, a execução simbólica é utilizada para gerar entradas para cada caminho de execução viável de um programa, gerando casos de teste para valores simbólicos que gerem um erro (CADAR et al., 2011). Um caminho de execução viável é uma sequência de valores booleanos, todos esses caminhos podem ser representados utilizando uma árvore de execução do programa (CADAR; SEN, 2013).

O estado de um programa executado de forma simbólica inclui: (a) os valores simbólicos das variáveis que são valores representativos das variáveis, exemplo, um programa contendo duas variáveis x e y, e havendo uma atribuição em x com o valor de $y + 1$ geraria a seguinte informação $x : Y + 1, y : Y$; (b) Uma condição de caminho (CC) que é uma condição lógica para o caminho ser executado, um exemplo seria uma estrutura de condição, onde a condição é que x seja maior que y; e (c) um *program counter* (PC) que aponta para o próximo estado a ser executado. A CC é uma fórmula booleana sem quantificadores que utiliza as entradas simbólicas, a fórmula aglomera todas as restrições que as entradas devem satisfazer para que a execução siga o caminho associado. O PC define a próxima instrução a ser executada. Uma árvore de execução simbólica exibe todos os caminhos seguidos durante a execução simbólica de um programa. Segundo Cadar e Sen (2013) o objetivo da execução simbólica pode ser descrito como obter um conjunto de entradas tais que todos os caminhos de execução viáveis possam ser explorados uma única vez ao executar o programa com essas entradas, na Seção 2.2.1 será apresentado um exemplo da utilização.

2.2.1 Árvore de execução simbólica

A árvore é um grafo onde seus nós representam os estados do programa e as arestas representam as transições entre os estados (KHURSHID et al., 2003), a árvore é uma forma de demonstrar os caminhos possíveis dentro de um programa, a Figura 3 mostra um exemplo de uma árvore para o programa da Figura 2. O programa contém duas variáveis que para este

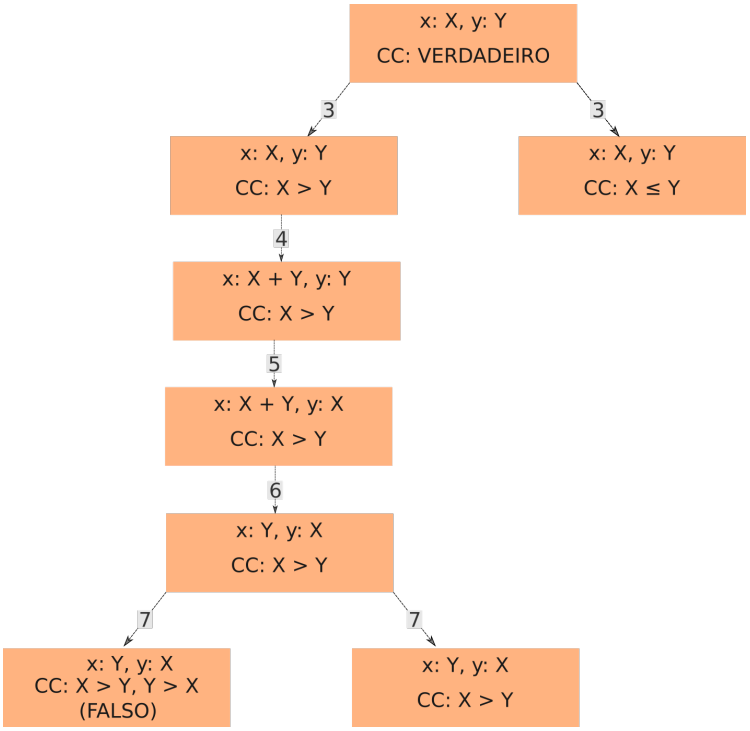
exemplo, vamos assumir que foram inicializadas com valores aleatórios, na linha 3 há uma condição de $x > y$, e logo em seguida os valores de x e y são trocados e caso $y > x$ chegamos a um estado de erro. Com a árvore de execução simbólica da Figura 3 podemos perceber que a seguinte condição: $x : Y$, $y : X$, e a CC $X > Y$. Para a linha 7 dar falso a condição de caminho será $(X > Y) \wedge (Y > X)$, o que é uma contradição.

Figura 2 – Exemplo de programa para execução simbólica

```
1  int x,y;
2
3  if(x > y){
4      x = x + y;
5      y = x - y;
6      x = y - x;
7      if(y > x) {
8        error();
9      }
10 }
```

Fonte: Adaptado de (KHURSHID et al., 2003)

Figura 3 – Exemplo de árvore de execução simbólica



Fonte: Adaptado de (KHURSHID et al., 2003)

2.3 Lógica Proposicional

Uma fórmula proposicional ϕ pode ser uma variável proposicional p, uma negação $\neg\phi_0$, uma conjunção $\phi_0 \wedge \phi_1$, uma disjunção $\phi_0 \vee \phi_1$ ou uma implicação $\phi_0 \Rightarrow \phi_1$ de fórmulas menores ϕ_0,ϕ_1. Uma atribuição de verdade M para uma fórmula ϕ, mapeia as variáveis proposicionais em ϕ para $\{\top, \bot\}$. Uma fórmula ϕ é satisfatível se existe uma atribuição de verdade M tal que $M \models \phi$. Se $M \models \phi$ para qualquer atribuição de verdade M, então ϕ é válido. Uma fórmula proposicional pode ser apenas válida ou sua negação é satisfatível (MOURA et al., 2007).

Um literal pode ser uma variável proposicional p ou sua negação $\neg p$. A negação de um literal p é $\neg p$ e a negação de $\neg p$ é somente p. Uma fórmula é uma cláusula se ela é a disjunção iterada de literais na forma $l_1 \vee ... \vee l_n$ para todos os literais l_i, onde $1 \leq i \leq n$. Uma fórmula esta na Forma Normal Conjuntiva (FNC) se está é uma conjunção iterada de cláusulas $\Gamma_1 \wedge ... \wedge \Gamma_m$ para todas as cláusulas Γ_i, onde $1 \leq i \leq m$ (MOURA et al., 2007).

2.3.1 Lógica de Primeira Ordem

Ao definir a assinatura de primeira ordem, assume-se um conjunto contável de variáveis X, símbolos de função F e predicados P. Uma assinatura de lógica de primeira ordem Σ é um mapa parcial de $F \cup P$ para os números naturais correspondentes à aridade do símbolo. Um Termo-Σ τ contém a forma $\tau := x|f(\tau_1, ..., \tau_n)$, onde $f \in F$ e $\Sigma(f) = n$. Por exemplo, se $\Sigma(f) = 2$ e $\Sigma(g) = 1$, então $f(x, g(x))$ é um temo Σ. Uma Fórmula-Σ ψ tem a forma

$$\psi := p(\tau_1, ..., \tau_n)|\tau_0 = \tau_1|\neg\psi_0|\psi_0 \vee \psi_1|\psi_0 \wedge \psi_1|\exists x > \psi_0|\forall x : \psi_0,$$

Onde $p \in P$, $\Sigma(p) = n$ e cada τ_i está no intervalo $1 \leq i \leq n$. Por exemplo, se $\Sigma(<) = 2$ para o símbolo predicativo $<$, então $(\forall x : (\exists y > x < y))$ é uma Fórmula-Σ. O conjuntos de variáveis livres em uma fórmula ψ é representado por $vars(\psi)$. Uma sentença é uma fórmula sem variáveis livres (MOURA et al., 2007).

Uma Estrutura-Σ, simbolizado por M, consiste em um domínio não vazio $|M|$ onde para cada $f \in F$ cujo $\Sigma(f) = n$, $M(f)$ é um mapa em $|M|$ para cada $p \in P$ tal que $\Sigma(p) = n$, $M(p)$ é um subconjunto de $|M|^n$ e para cada $x \in X$, $M(x) \in |M|$. A interpretação de um termo a em M é dada por $M[\![x]\!] = M(x)$ e $M[\![f(a_1, ..., a_n)]\!] = M(f)(M[\![a_1]\!], ..., M[\![a_n]\!])$. Para uma

Fórmula-Σ ψ e uma Estrutura-Σ M a satisfação $M \models \psi$ pode ser definida como:

$$M \models a = b \iff M[\![a]\!] = M[\![b]\!] \tag{2.1}$$

$$M \models p(a_1, ..., a_n) \iff (M[\![a_1]\!], ..., M[\![a_n]\!]) \in M(p) \tag{2.2}$$

$$M \models \neg\psi \iff M \not\models \psi \tag{2.3}$$

$$M \models \psi_0 \vee \psi_1 \iff M \models \psi_0 \ \texttt{ou} \ M \models \psi_1 \tag{2.4}$$

$$M \models \psi_0 \wedge \psi_1 \iff M \models \psi_0 \ \texttt{e} \ M \models \psi_1 \tag{2.5}$$

$$M \models (\forall x : \psi) \iff M\{x \to a\} \models \psi, \forall a \in |M| \tag{2.6}$$

$$M \models (\exists x : \psi) \iff M\{x \to a\} \models \psi, \ \texttt{para alguns} \ a \in |M| \tag{2.7}$$

Uma Fórmula-Σ de primeira ordem ψ pode ser satisfeita se existe uma Estrutura-Σ M tal que $M \models \psi$ e é válida se em todas as Estruturas-Σ M, $M \models \psi$. Uma Sentença-Σ é ou satisfatível ou sua negação é válida (MOURA et al., 2007).

2.4 Lógicas Temporais

Segundo Baier e Katoen (2008) lógicas temporais expandem a lógica proposicional ou a lógica de predicados em modalidades que permitem a referência de um comportamento infinito de um sistema reativo. Elas provém uma notação precisa para expressar propriedades sobre a relação entre dois estados na execução. Os operadores presentes na maioria das lógicas temporais são:

$$\Diamond \ \texttt{"eventualmente"(eventualmente no futuro)} \tag{2.8}$$

$$\Box \ \texttt{"sempre"(agora e sempre no futuro)} \tag{2.9}$$

A natureza do tempo em lógicas temporais podem ser linear (Lógica Temporal Linear - LTL) ou de ramificação (Lógica de Árvore de Computação - LAC ou CTL). Em uma visão linear, em cada momento no tempo existe um momento sucessor, já na visão de ramificação o tempo pode ser dividido em cursos alternativos (BAIER; KATOEN, 2008).

2.4.1 Lógica Temporal Linear (LTL)

Uma fórmula LTL é composta por proposições atômicas, conectores booleanos de conjunção, disjunção e negação, e dois operadores temporais: $\bigcirc$ (próximo) $\bigcup$ (enquanto). Um exemplo seria um semáforo com as fases "verde", "amarelo"e "vermelho", a propriedade $\Box\Diamond\texttt{verde}$ significa que a luz do semáforo será infinitamente frequentemente verde. Uma especificação dos ciclos de luz do semáforo e sua ordem cronológica pode ser definido através de uma conjunção de fórmulas LTL. O requerimento **uma vez vermelho, o sinal não pode ficar verde imediatamente** pode ser expresso em LTL como (BAIER; KATOEN, 2008):

$$\Box(\texttt{vermelho} \to \neg \bigcirc \texttt{verde})$$

O requerimento **uma vez vermelho, o sinal sempre fica verde eventualmente após ficar amarelo por algum tempo** é expressado como (BAIER; KATOEN, 2008):

$$\Box(\texttt{vermelho} \rightarrow \bigcirc(\texttt{vermelho} \bigcup (\texttt{amarelo} \wedge \bigcirc(\texttt{amarelo} \bigcup \texttt{verde}))))$$

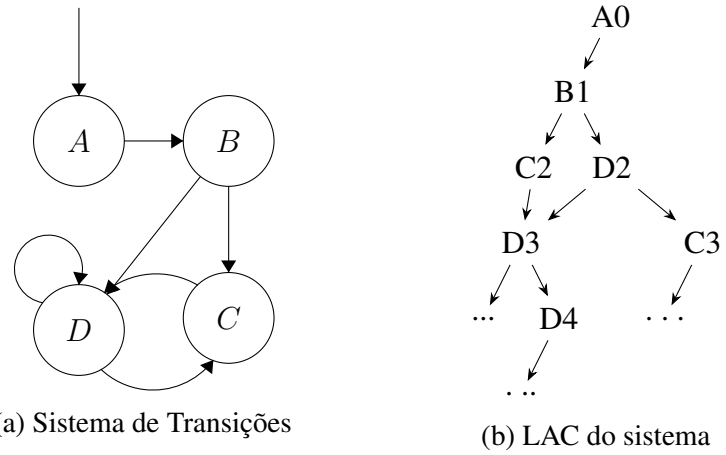
2.4.2 Lógica de Árvore de Computação (LAC)

A LAC é uma lógica onde o tempo pode evoluir além de um possível futuro, usando um modelo temporal discreto, as fórmulas LAC são compostas por: proposições atômicas, conectivos booleanos e finalmente, operadores temporais que consistem em (ROCHA et al., 2015):

- **G**: especifica que uma propriedade é válida em todos os estados do caminho;
- **F**: especifica que uma propriedade será válida em algum estado do caminho;
- **X**: especifica que uma propriedade deve ser válida no próximo estado de um caminho;
- **U**: especifica que existe um estado ao longo do caminho onde a segunda propriedade será válida e em todos os estados antes dela, a primeira será válida;
- **R**: especifica que a segunda propriedade é válida ao longo do caminho até o primeiro estado onde a primeira propriedade é válida.

A Figura 4 mostra um exemplo de um sistema de transições infinito no formato LAC. O sistema contém 4 estados que ficam em transição, do lado é mostrado um LAC contendo as computações.

Figura 4 – Exemplo de LAC



(a) Sistema de Transições

(b) LAC do sistema

Fonte: Adaptado de (BAIER; KATOEN, 2008).

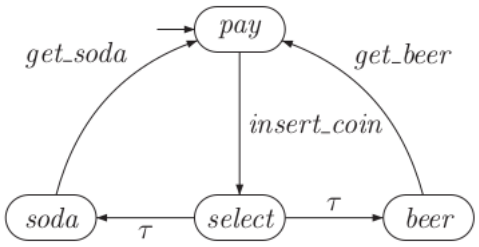
2.5 Propriedades de Segurança

Propriedades de segurança são imposições sobre os requerimentos de um conjunto finito de caminhos, e que, não podem ser verificados considerando apenas o estado. Um exemplo disso, seria um caixa eletrônico, um requerimento seria de que o dinheiro só pode ser retirado somente se a pessoa forneceu uma identificação válida (BAIER; KATOEN, 2008). Segundo Rocha et al. (2015) podemos, informalmente, dizer que uma propriedade em tempo linear especifica um comportamento admissível de um sistema. Caso o sistema falhe ao satisfazer alguma propriedade de segurança (o usuário possa retirar dinheiro sem se identificar), então existe uma execução finita que revele isso.

Formalmente podemos definir uma propriedade de segurança como: *dado um sistema de transições* $ST = (S, S_0, E)$*, seja um conjunto* $B \subset S$ *que especifica um conjunto de maus estados tais que* $S_0 \cap B = \emptyset$*, pode-se dizer que* ST *é seguro com relação a* B*, denotado por* $ST \models AG\neg B$ *se não existe um caminho no sistema de transição do estado inicial* S_0 *até o estado* B*, de outro modo é dito que* ST *não é seguro* (ROCHA et al., 2015).

Baier e Katoen (2008) exemplifica descrevendo uma máquina onde o usuário entra com uma moeda e o sistema retorna uma bebida de forma não determinística. A Figura 5 mostra o sistema de transições desssa máquina, um exemplo de propriedade de segurança seria: A máquina só entrega bebidas após ser colocada uma moeda (BAIER; KATOEN, 2008).

Figura 5 – Sistema de transições de uma máquina de bebidas



Fonte: (BAIER; KATOEN, 2008)

2.5.1 Propriedades de segurança de memória

Beyer (2017) classifica as propriedades de segurança de memória em três categorias, definidas em LAC:

G valid-free Todas as desalocações são válidas (contraexemplo: desalocação de um endereço inválido). A definição LAC traduz para: Não existe nenhum caminho finito de execução

onde uma desalocação inválida ocorre. A Figura 6 contém um programa em que um valor é desalocado de forma incorreta na linha 3, pois ele já havia sido desalocado.

Figura 6 – Programa com erro de desalocação

.

```
1 int *pointer = (int*) malloc(4);
2 free(pointer);
3 free(pointer);
```

Fonte: Própria

G valid-deref Todas as desreferências de ponteiros são válidas (contraexemplo: operação de I/O em um endereço de memória inválido). A definição LAC traduz para: Não existe nenhum caminho finito de execução onde uma desreferência inválida de ponteiro acontece. O programa da Figura 7 contém um erro de desreferência, pois ao acessar uma posição além do limite do `array` é acessado um endereço de memória inválido.

Figura 7 – Programa com erro de desreferência

.

```
1 int array[4];
2 array[4] = 7;
```

Fonte: Própria

G valid-memtrack Todas as referências de endereços alocados não são perdidos (contraexemplo: vazamento de memória). A definição de LAC traduz para: Não existe nenhum caminho finito de execução onde um endereço alocado é perdido. A Figura 8 contém um programa com um erro de vazamento, pois um recurso alocado não é liberado ao final da execução do programa.

Figura 8 – Programa com erro de vazamento

.

```
1 int main(){
2   int *pointer = malloc(4);
3   return 0;
4 }
```

Fonte: Própria

2.6 Análise de contraexemplos

Ferramentas de verificação de *software* podem produzir resultados incorretos pois o problema da verificação é indecidível, ou seja, para a análise haverão aproximações o que pode gerar erros (COUSOT, 2001) tanto para corretude como para violação. Sendo necessário muito esforço manual para analisar se um *bug* encontrado realmente representa uma violação (ROCHA et al., 2012). A verificação de contra-exemplos ou prova de testemunho (em inglês *witness checker*) é o processo de dar evidências para uma declaração de que um programa satisfaz ou viola sua especificação. A evidência da ausência ou presença de uma violação é dada por uma ou mais *witness* (BEYER et al., 2015a).

Segundo Beyer et al. (2015a) um *witness* deve poder ser lida por humanos e um validador de *witness*. Um autômato de *witness* permite diferentes níveis de abstração, onde se pode definir todos ou apenas um caminho que leve a um estado final de erro (ou nenhum erro). Além disso, Beyer et al. (2015a) comenta sobre o problema de alguns verificadores fazerem suas *witness* em formatos proprietários, que não são lidas nem por humanos nem por máquinas, e uma troca de *witness* entre verificadores era impossível. Essa troca de *witness* é essencial para eliminar alarmes falsos e para aumentar o potencial de combinação entre diferentes ferramentas de verificação, pois uma vez que um verificador imprime seu *witness* em um formato padronizado de troca, nenhuma implementação extra é necessária para a validação do *witness* (BEYER et al., 2015a).

Na Figura 9 temos um exemplo de um programa e ao lado seu respectivo *witness* em formato de autômato, o programa contém erros para alguns valores de `a`, assumindo que `foo` seja é função não-determinística, o autômato gerado contém todas as transições necessárias para alcançar um estado de erro no programa, uma *witness* de violação precisa de pelo menos um caminho que alcance um caminho de erro (BEYER et al., 2015a), o exemplo contém os valores que caso sejam gerados pela função `foo` levaria o programa a um estado de erro (que seria a execução da função `error`. Já na Figura 10 temos um *witness* corretude, diferentemente do *witness* de violação as transições funcionam nas ramificações dos programas e os estados armazenam invariantes sobre as variáveis que são condições que devem ser verdadeiras para a execução do programa alcançar aquela localização (ROCHA et al., 2015), no exemplo de corretude, o estado `q3` é inalcançável pois a seguinte restrição deveria ser obedecida $(0 \leq a < 100) \wedge a > 200$ que é uma contradição.

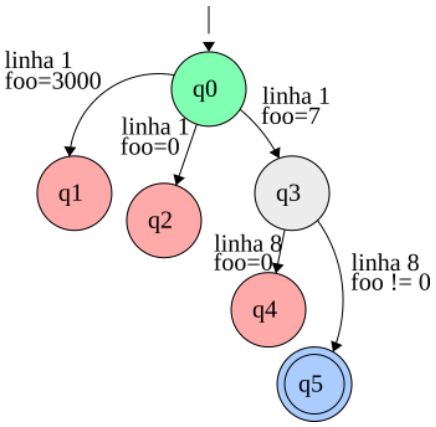
2.7 *Bounded model checking*

Técnicas de verificação baseadas em modelo são baseadas em descrever o possível comportamento de um sistema de uma forma matemática, precisa e não ambígua. Verificou-se que uma modelagem precisa de um sistema muitas vezes leva a descoberta de incompletudes, ambiguidades e inconsistências em especificações informais. Esse problemas geralmente só são descobertos muito depois durante o processo de desenvolvimento. Os modelos de sistemas são

Figura 9 – Exemplo de programa e *witness* de violação

```
1  int a = foo();
2  if(a >= 2000) {
3    return error();
4  }
5  if (a <= 0) {
6    return error();
7  }
8  a = foo();
9  a = a*a;
10 if (a == 0) {
11   return error();
12 }
13 return 0;
```

(a) Código C



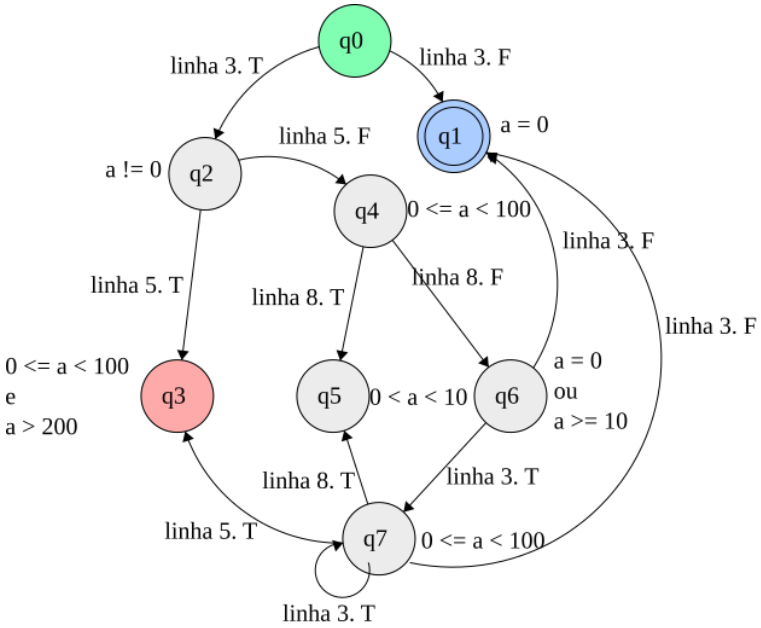
(b) *Witness* de violação

Fonte: Adaptado de (BEYER et al., 2015a).

Figura 10 – Exemplo de programa e *witness* de corretude

```
1  int a = foo();
2  int *ptr = malloc(4);
3  while(a) {
4    a = foo() % 100;
5    if(a > 200) {
6      return error();
7    }
8    if((a<10) && a) {
9      free(ptr);
10     return 0;
11   }
12 }
13 if (a == 0) {
14   free(ptr);
15 }
16 return 0;
```

(a) Código C



(b) *Witness* de corretude

Fonte: Adaptado de (BEYER et al., 2015a).

acompanhados por algoritmos que sistematicamente exploram todos os estados de um modelo de sistema. Isso provêm a base para uma gama de técnicas de verificação indo desde exploração exaustiva (*model checking*) até experimentos com um conjunto restrito de cenários no modelo (simulação), ou na realidade (testes) (BAIER; KATOEN, 2008).

Segundo Baier e Katoen (2008) *model checking* é uma técnica de verificação automatizada que: dado um modelo finito de estados de um sistema e uma propriedade formal, sistematicamente checa se essa propriedade é válida para um estado que pertence ao modelo. Dessa maneira, o *model checking* pode demonstrar que um dado modelo de sistema realmente satisfaz uma determinada propriedade. A dificuldade do *model checking* é lidar com espaços de estados muito grandes, pois dependendo do sistema eles podem crescer de forma exponencial (BAIER; KATOEN, 2008).

O modelo sistema é geralmente gerado de forma automática a partir de uma descrição do modelo que é especificado em algum dialeto de linguagens de programação como C ou Java. A especificação de uma propriedade descreve o que sistema deve fazer, e o que não deve fazer, enquanto a descrição do modelo descreve como o sistema se comporta. O *model checker* examina todos os estados relevantes e checa se eles satisfazem a propriedade desejada. Se um estado viola essa propriedade, o *model checker* gera um contra-exemplo que indica como o modelo pode chegar em um estado indesejado (BAIER; KATOEN, 2008).

Segundo Rocha et al. (2015) o *Bounded Model Checking* (BMC) é a verificação de uma dada propriedade em uma determinada profundidade: dado um sistema de transições M, uma propriedade ϕ, e um limite (*bound*) k, o BMC desenrola o sistema k vezes e traduz o sistema em uma condição de verificação(CV) ψ tal que ψ é satisfeito se e somente se ϕ tem um contra-exemplo de profundidade menor ou igual a k. Um exemplo de ferramenta BMC é o ESBMC (CORDEIRO et al., 2012).

2.8 Técnicas de Compiladores

Esta seção irá descrever os seguintes tópicos: Análise Estática, Análise Dinâmica, Otimizações de Código e *framework* de compiladores com LLVM.

2.8.1 Análise Estática

A análise estática é a análise de um programa sem a execução , através de análises no código-fonte e analisando a semântica da linguagem (como tipagem ou acesso de elemento fora do índice). Compiladores utilizam esse tipo de análise durante o processo de compilação para verificar erros (como de tipagem) e otimização de código. As ferramentas que fazem essa análise só precisam ler o código para efetuar a análise (ROCHA et al., 2015). O programa da Figura 11 contém dois erros que podem ser detectados através de uma análise estática: (a) na linha 2, tenta-se carregar o conteúdo de `a` no índice 7 sendo que na linha 1 ele foi definido como

de tamanho 3; (b) na linha 5, tenta-se atribuir um valor de uma variável do tipo flutuante para um inteiro, o que pode gerar erros futuros.

Figura 11 – Programa para análise estática

.

```
1 int a[3];
2 int b = a[7];
3
4 long c = 5.5432;
5 a[0] = c;
```

Fonte: Própria

Segundo (COUSOT, 2001) A vantagem dessa abordagem é que ela pode ser usada para programas grandes (com mais de 220 mil linhas de código C) sem interação do usuário e as abstrações escolhidas podem ser implementadas em bibliotecas que podem ser reutilizadas em diferentes linguagens. Já a desvantagem dessa abordagem é que as especificações e a maioria das propriedades são simples, geralmente propriedades de segurança elementares como erros de execução.

Um exemplo de aplicação que utiliza esse tipo de análise seria o ASTRÉE (COUSOT et al., 2005) que é um analisador estático que prova automaticamente a falha de erros em tempo de execução para programas escritos em C, e foi aplicado com sucesso em vários sistemas embarcados críticos. Astrée baseia-se na teoria da interpretação abstrata (COUSOT; COUSOT, 1977) e prossegue calculando uma aproximação das propriedades semânticas de traços de execução do programa analisado e provando que essas propriedades abstratas implicam na ausência de erros em tempo de execução.

2.8.2 Análise Dinâmica

A análise dinâmica infere conclusões a partir da execução do programa, assim derivando propriedades de segurança para uma ou mais execuções, podendo detectar a violação das mesmas. Para a análise dinâmica, as ferramentas devem instrumentar o programa com análises (ROCHA et al., 2015). Segundo Ernst et al. (2007), algumas utilizações dessa abordagem são: evitar *bugs*, depuração, teste e verificação. Com a análise dinâmica é possível executar testes e verificar se suas assertivas não são violadas.

O VALGRIND (NETHERCOTE; SEWARD, 2007) é um *framework* que faz uso de instrumentação binária para análise dinâmica. O módulo *memcheck* VALGRIND verificar por erros de memória em tempo de execução, por exemplo, vazamentos de memória (CHENG et al., 2006). A verificação efetuada pelo VALGRIND envolve sombrear/marcar cada bit de dados em registos e memória com um segundo *bit* que indica se o bit tem um valor definido. Cada operação de determinação de valor é instrumentada com uma operação de marcação que propaga

os *bits* adequadamente. O Memcheck usa esses *bits* de marcação para detectar usos de valores indefinidos que poderiam afetar adversamente o comportamento de um programa.

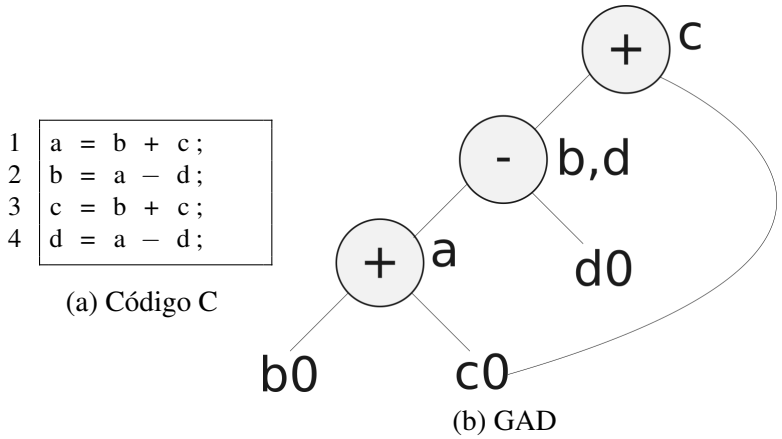
2.8.3 Otimizações de Código

Podemos obter uma melhora significante no tempo de execução ou verificação de um código, executando otimizações locais, dentro de um bloco básico, ou através de uma otimização global, através do fluxo entre os blocos básicos (AHO et al., 1986). Por ser um assunto complexo e com muitas técnicas a serem consideradas, abordaremos apenas as seguintes técnicas que são abordadas na solução proposta neste trabalho: Representação com Grafo Acíclico Direcionado (GAD), Remoção de código morto, sub-expressões comuns, propagação de constantes.

2.8.3.1 Representação com Grafo Acíclico Direcionado

Diversas técnicas de otimização local começam por transformar um bloco básico de código em um GAD. A Figura 12 exibe um exemplo de GAD utilizando a partir de um programa. Podemos observar que os nós contém os operandos utilizados e quais variáveis se refere e já podemos vislumbrar algumas otimizações possíveis (como sub-expressões comuns), visto que b e d recebem os mesmos valores.

Figura 12 – Exemplo de GAD de um programa



(a) Código C

(b) GAD

Fonte: Adaptado de (AHO et al., 1986).

2.8.3.2 Sub-expressões comuns

Sub-expressões comuns de um programa são detectadas ao perceber que um novo nó M, de um GAD, esta para ser adicionado mesmo já existindo um nó N com os mesmos filhos, na mesma ordem e com os mesmos operadores. Logo, se isso ocorre, N computa o mesmo valor de M e pode ser usado no seu lugar (AHO et al., 1986). No exemplo da Figura 13 podemos perceber uma otimização de sub-expressões comuns no nó atrelado a b e d, sendo uma operação que só precisa ser realizada uma vez. A Figura 13 exibe um exemplo dessa otimização.

Figura 13 – Exemplo de aplicação da técnica de sub-expressões comuns

```
1 a = b + c;
2 b = a - d;
3 c = b + c;
4 d = a - d;
```

(a) Código C antes da técnica

```
1 a = b + c;
2 b = a - d;
3 c = b + c;
4 d = b;
```

(b) Código C após a técnica

Fonte: Adaptado de (AHO et al., 1986).

2.8.3.3 Remoção de código morto

Código morto são instruções que computam um valor que nunca será utilizado, essa técnica basicamente remove do GAD qualquer raiz que não contenha variáveis que são utilizadas em outros nós. A reaplicação dessa transformação irá remover todos os nódulos do GAD que correspondem a um código morto (AHO et al., 1986). A Figura 14 mostra dois programas: o primeiro contendo algumas computações desnecessárias, como atribuições seguidas em uma variável, código inacessível, e computações cujos resultados não são utilizados; o segundo programa, está simplificado, não havendo operações desnecessárias.

Figura 14 – Exemplo de aplicação da técnica de remoção de código morto

```
1  int global;
2
3  int foo() {
4      int a = 3 * 7;
5      int b = a / 9;
6      global = 1;
7      global = 10;
8      return 2;
9      global = 5;
10 }
```

(a) Código C antes da técnica

```
1 int global;
2
3 int foo() {
4     global = 10;
5     return 2;
6 }
```

(b) Código C após a técnica

Fonte: Própria.

2.8.3.4 Propagação de constantes

A propagação de constantes consiste em realizar operações matemáticas ou lógicas, durante o processo de compilação ao invés de durante a execução. A técnica consiste em buscar variáveis ou funções cujo valor seja constante, ou seja, se todos os valores que sejam necessários para calcular seus valores são conhecidos durante o tempo de compilação, a computação já é executada durante a compilação (AHO et al., 1986). A Figura 15 mostram como seria um programa antes e depois da aplicação dessa técnica.

Essas técnicas geram algumas vantagens para a verificação de *software* como:

Figura 15 – Exemplo de aplicação da técnica de propagação de constantes

```
1 int x = 2;
2 int b = a + 3;
```

(a) Código C antes da técnica

```
1 int a = 2;
2 int b = 5;
```

(b) Código C após a técnica

Fonte: Própria.

- Expressões avaliadas durante a compilação não precisam ser avaliadas durante a execução. Se as expressões estiverem dentro de estruturas de repetição, a avaliação durante a compilação pode reduzir o tempo de execução (WEGMAN; ZADECK, 1991);
- Propagação de constantes pode ser utilizada sobre vários domínios como por exemplo a tipagem (WEGMAN; ZADECK, 1991).

2.8.4 LLVM

O LLVM é um *framework* de compilação desenvolvido pra suportar de forma transparente e duradoura, análises e transformações de programas. Para isso, o LLVM utiliza uma linguagem intermediária, o LLVM IR e uma API em alto-nível para construção de *passes* capazes de analisar ou transformar um programa em LLVM IR (LATTNER; ADVE, 2004).

O LLVM IR consiste em um conjunto infinto de registradores virtuais que podem guardar valores de tipos primitivos (inteiro, ponto flutuante, ponteiro e booleano). A arquitetura do LLVM é baseada em *load e store*, os programas transferem valores entre registradores e memória somente através de operações *loads* e *stores* usando ponteiros tipados (LATTNER; ADVE, 2004). O LLVM IR utiliza instruções em um formato de *Static Single Assignment* (SSA) que é um formato para instruções onde para as atribuições de variáveis sempre é gerado uma nova variável para armazenar seus resultados, o que facilita otimizações e análises sobre o código (CYTRON et al., 1991).

O formato SSA está no paradigma funcional (APPEL, 1998), ou seja, um código que originalmente poderia estar em um paradigma imperativo é convertido para um paradigma funcional. A vantagem do paradigma funcional é a facilidade de realizar otimizações e análises como as descritas na subseção 2.8.3 Otimizações de Código, pois: variáveis não mudam de valor, entre outras vantagens (APPEL, 1998).

As análise e transformações que o LLVM proporciona podem ser usadas antes ou depois do processo de linkedição, ou durante o processo de montagem (podendo gerar otimizações específicas de uma arquitetura). A Figura 16 exibe um exemplo de um programa em C compilado para LLVM IR com a opção de aplicar otimizações (*flag* `-O3`).

Figura 16 – Código C compilado e otimizado para LLVM IR

```c
int foo(int n, int
    total) {
  if(!n)
    return total;
  return foo(n-1,
      total += 2);
}

int main() {
  int a = 3;
  int i = 0;

  for(;i<3; i++)
    a += 5;

  return foo(a+2, 0);
}
```

(a) Código C

```
define i32 @foo(i32, i32) {
  %3 = shl i32 %0, 1
  %4 = add i32 %3, %1
  ret i32 %4
}

define i32 @main() {
  ret i32 40
}
```

(b) Código LLVM após compilação com a *flag* `-O3`

Fonte: Própria.

3 TRABALHOS CORRELATOS

Neste capítulo serão descritos e apresentados os principais trabalhos que estão relacionados com a solução proposta deste trabalho, e de que forma eles contribuem ou diferenciam deste trabalho. Em específico, serão abordados as seguintes trabalhos que utilizam ferramentas de verificação e/ou teste de *software*: Valgrind Memcheck (NETHERCOTE; SEWARD, 2007), Symbtiotic 3 (CHALUPA et al., 2016), LEAKPOINT (CLAUSE; ORSO, 2010), SMACK (CARTER et al., 2016), Ceagle (WANG et al., 2016), Map2Check v6 (ROCHA et al., 2015), o PredatorHP (KOTOUN et al., 2016), o Ultimate Automizer (HEIZMANN et al., 2013) e o ESBMC (CORDEIRO et al., 2012). Todas essas feramentas verificam programas em C e são capazes de detectar violações em propriedades de segurança de memória.

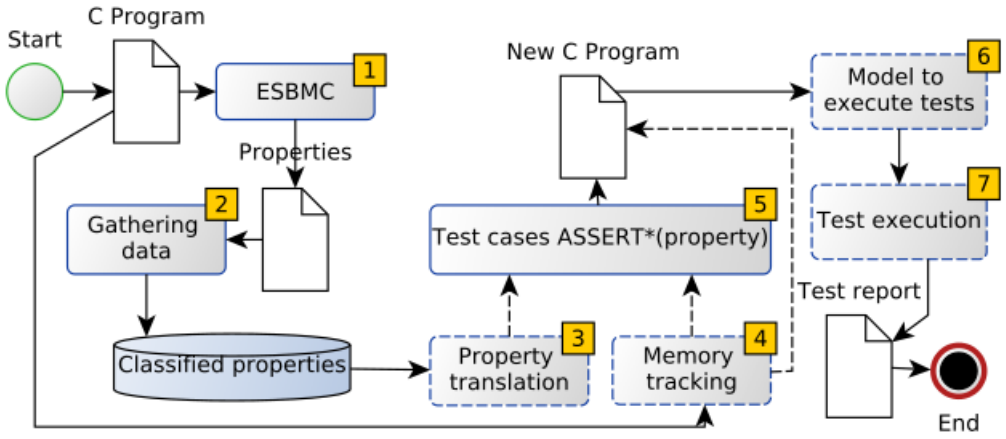
O *Map2Check* (ROCHA et al., 2015) é uma ferramenta criada para unir a técnica de verificação formal de *software model checking* com teste de *software*, neste caso teste unitário. Esta ferramenta gera assertivas utilizando ferramentas de *bounded model checking* e depois as verifica utilizando CUnit por meio de uma biblioteca auxiliar para as assertivas. O foco do Map2Check é a linguagem C com o objetivo de observar e validar propriedades de segurança da memória. Para geração de valores não determinísticos adotou-se o uso de uma função que gera um número aleatório (0 ou 1) a partir de um vetor de acordo uma distribuição de 30% para 0 e 65% para 1 (ROCHA et al., 2015). O Map2Check antigo estava na versão 6, a Figura 17 demonstra o fluxo do Map2Check v6[1]. O fluxo consiste em (ROCHA et al., 2015):

- **Identificação das propriedades de segurança (1):** O Map2Check v6 usa o ESBMC para identificar possíveis violações de propriedades de segurança, dando como entrada um programa C;
- **Coleta de informações das propriedades de segurança (2):** Na segunda etapa, é feito um tratamento sobre os resultados da etapa anterior para coletar informações necessárias para as próximas etapas;
- **Tradução das propriedades de segurança (3):** Ao ESBMC produzir as propriedades violadas, são utilizadas funções próprias do ESBMC, é feita então uma tradução para assertivas, para a verificação ser realizada sem a intervenção do ESBMC;
- **Rastreamento de memória (4):** O rastreamento de memória consiste em localizar e anotar todas as operações sobre ponteiros, diretamente no código fonte analisado, afim de estender a verificação das propriedades geradas pelo ESBMC. Com ele é possível validar propriedades de desalocação inválida e de vazamentos de memória;

[1] <https://github.com/hbgit/Map2Check/releases/tag/map2check_v6>

- **Instrumentação do código com assertivas (5)** Nessa etapa são adicionadas as assertivas contendo as prorpriedades de segurança identificadas na etapa 2 e 4. Essas assertivas podem ser `assert` da linguagem C ou uma assertiva do *framework* CUnit;
- **Implementação dos testes (6)** A implementação dos testes pode ocorrer de duas formas: (a) se for uma assertiva C, então é feito um `include` de uma biblioteca do Map2Check v6; e (b) se for CUnit é aplicado um *template* fornecido pelo método ao o programa analisado;
- **Execução dos testes (7)** Finalmente se executa os casos de teste gerados e é verificado os casos que falharam ou deram sucesso.

Figura 17 – Fluxo do Map2Check v6



Fonte: (ROCHA et al., 2015)

Neste trabalho é proposto melhorias no Map2Check que resultam na versão 7. A nova versão conta com diferenças significativas como mostrado pela Figura 18[2]. O novo método será descrito com mais detalhes na próxima seção, mas de maneira geral as principais diferenças são:

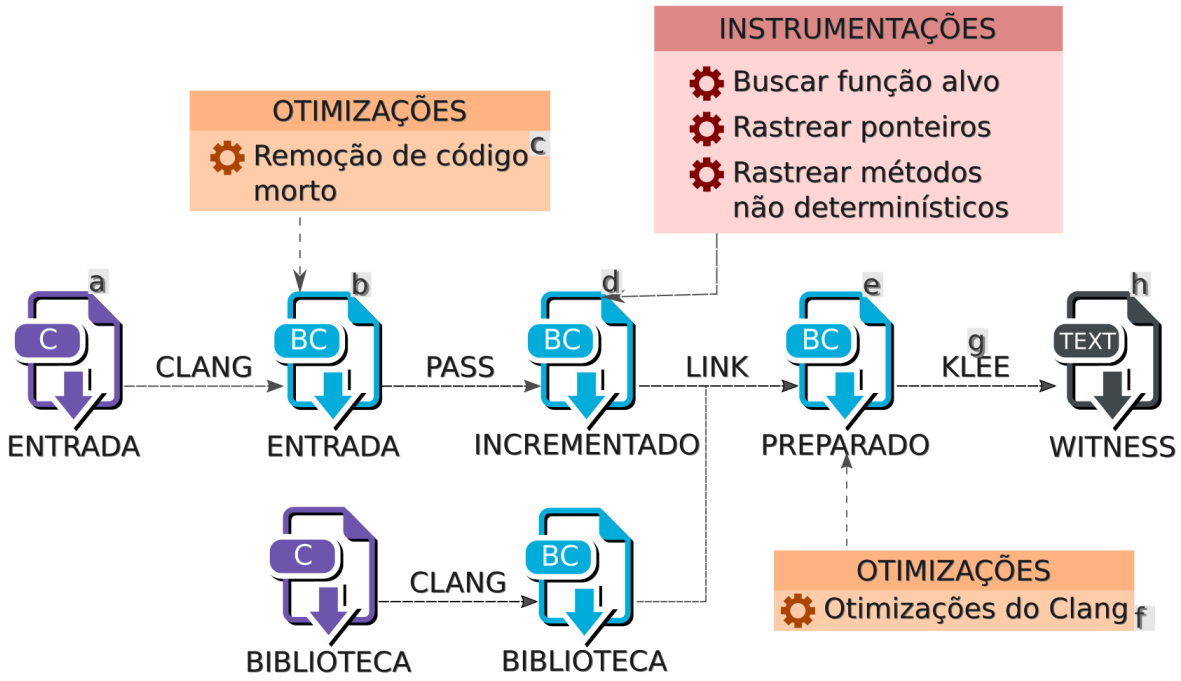
- O processamento do código é feito em nível intermediário, usando LLVM, o que possibilita desenvolvimento de métodos, análises e transformações sem serem engessadas na linguagem de programação C;
- As propriedades de segurança são geradas pela própria ferramenta, por meio de análise estática baseada em instruções e implementadas em assertivas suportadas por bibliotecas em C, sendo mais simples adicionar suporte a novas ferramentas, portanto não é mais necessário a utilização do ESBMC para propriedades de segurança de memória;

[2] Ícones dos vetores obtidos de <br.freepik.com>

- O *framework* do LLVM proporciona aos desenvolvedores uma API poderosa para transformações em diversos níveis de código (original, intermediário e máquina), possibilitando uma ferramenta fazer uma análise em diversos níveis. Com isso em mente, a nova versão do Map2Check permite contribuidores ou novas funcionalidades sejam implementadas sem alterar o fluxo geral do programa;
- Utilização de execução simbólica ao invés de execução probabilística. O Map2Check v6 usava probabilidade para tentar encontrar as situações de violação, já na versão 7, é utilizado execução simbólica (utilizando o Klee (CADAR et al., 2008)).
- Nesta nova versão foi aprimorado a geração de contra-exemplo, com o foco no formato witness (BEYER et al., 2015a).

.

Figura 18 – Fluxo do Map2Check v7



Fonte: Própria

Clause e Orso (2010) proporam uma método para detectar *memory leaks* em programas em C e C++, denominado de LEAKPOINT, de forma que obtenha informações para geração de contra-exemplos. O LEAKPOINT faz uma análise buscando funções de alocação (`malloc` e `new`) e liberação de memória (`free` e `delete`), além de operações com ponteiros. Após isso, é feita uma análise se todas as instruções de alocação são desalocadas e se a aritmética de ponteiros está correta, caso seja verificado um problema, é possível informar com precisão onde ele ocorreu.

A ideia do LEAKPOINT é marcar valores interessantes (como variáveis e localidades de memória), após isso, propagar essas marcar com seus dados associados, conforme o programa executa e finalmente checar quais marcas estão associadas com qual dado de programa a um ponto específico da execução. Por exemplo, quando uma área m da memória é alocada, a técnica cria uma marca tm e a usa para marcar o ponteiro retornado, ao esse valor marcado for utilizado em alguma operação aritmética ou lógica, é executado um algoritmo para determinar se outros valores que foram utilizados nessa operação devem ser marcados. A técnica armazena mais cinco informações para cada marca: (a) Local da alocação; (b) Tamanho alocado; (c) Indicador de desalocação; (d) Número de ponteiros que foram marcados; e (e) Local de último uso (CLAUSE; ORSO, 2010).

Similarmente ao LEAKPOINT o Map2Check v7 LLVM, marca valores interessantes e anota valores relacionados a eles. Essas marcas ocorrem durante uma análise estática, onde é buscado funções e operações sobre ponteiros, porém, além de memória alocada, também é feito o rastreio de memória *heap* e não é feita a propagação dessas marcas, pois todas as operações envolvendo memória são analisadas.

Valgrind é um *framework* para instrumentação dinâmica em código binário (DBI) que são utilizados por ferramentas de *Dynamic Binary Analysis* (DBA), onde o código de análise é instrumentado durante a execução, o que é conveniente pois o usuário não necessita recompilar o código. O módulo *memcheck* é capaz de detectar violação de propriedades de segurança de memória utilizando *shadow values*, *bit* extra de rastreamento de endereços (NETHERCOTE; SEWARD, 2007), como descrito na subseção 2.8.2. Similarmente o Map2Check faz uso de instrumentação para verificar e validar propriedades de segurança de memória, porém o Map2Check não utiliza *shadow values* para ocultar os registradores, a análise é feita anotando todas as operações de memória em uma estrutura. Essas diferenças proporcionam maior controle sobre a semântica original do código, facilitando a geração do contraexemplo.

O *Extended SMT-Based Bounded Model Checker* (ESBMC) é um BMC baseado em SMT (Satisfiability Modulo Theories) (CORDEIRO et al., 2012) que decide a satisfatibilidade de fórmula de primeira ordem usando uma combinação de diferentes teorias de bases e generaliza a satisfatibilidade proposicional provendo suporte a teorias decidíveis de primeira ordem (ROCHA et al., 2015). O ESBMC é capaz de verificar programas em C e C++ (RAMALHO et al., 2013)com uma mais *threads* e código com variáveis e *locks* compartilhadas (MORSE et al., 2013). O ESBMC é capaz de ser utilizado para gerar propriedades de segurança de programas em C que podem ser verificadas afim de garantir a corretude do código (ROCHA et al., 2015). O ESBMC usa uma análise estática para aproximar para cada variável de ponteiro o conjunto de objetos de dados (isto é, blocos de memória) no qual este objeto poderia apontar para algum estágio na execução do programa. Os objetos de dados são numerados e um alvo de ponteiro é representado por um par de inteiros que identificam o objeto de dados e o deslocamento dentro do objeto. O valor de uma variável de ponteiro é então o conjunto de objetos (objeto, deslocamento)

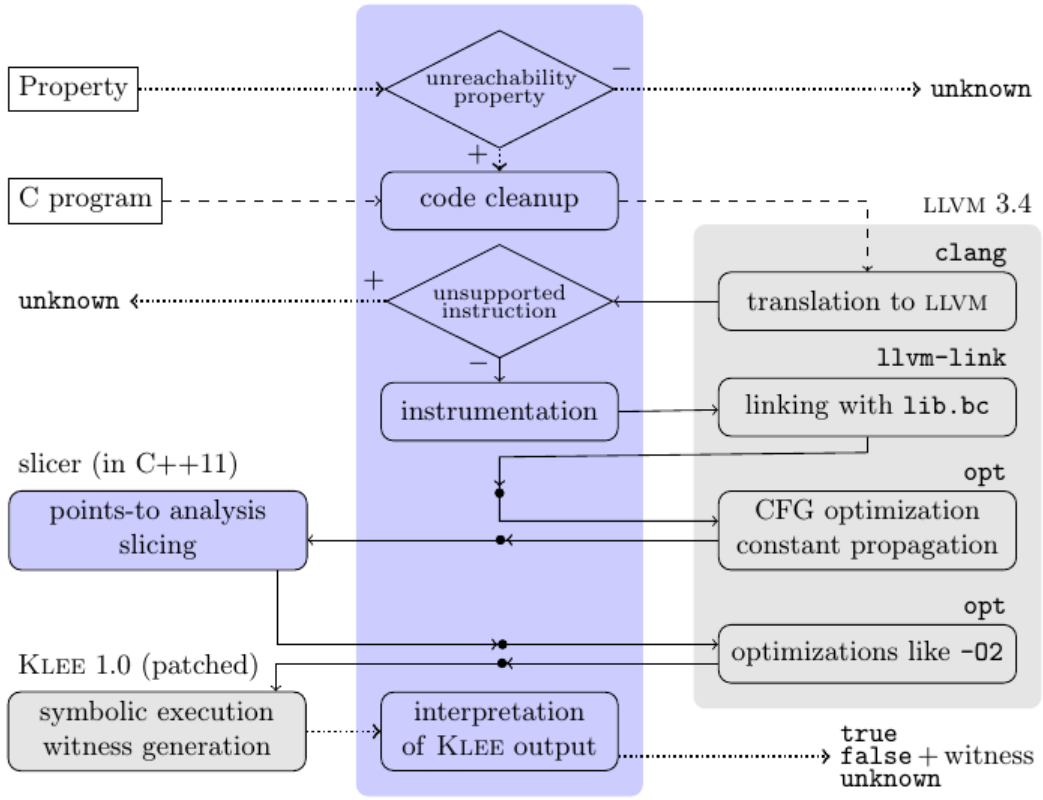
para o qual o ponteiro pode apontar na etapa de execução atual. O resultado de uma desreferência é a união dos conjuntos de valores associados a cada um dos objetos (objeto, `offset`).

O ESBMC é capaz de inferir propriedades de segurança de programas em C como: *overflow*, *underflow*, segurança de memória, limites de vetos, atomicidade, violações de ordem, *deadlock* e *data race*. É possível utilizar o ESBMC para gerar uma Condição de Verificação (CV) e verificar segurança de memória (ROCHA et al., 2015). Para segurança de memória as seguintes CV's podem ser usadas:

- **SAME_OBJECT:** Verifica se o ponteiro referencia o objeto correto;
- **INVALID_POINTER:** Verifica se o ponteiro é nulo ou um objeto inválido;
- **IS_DYNAMIC_OBJECT:** CV utilizado para checar se os argumentos de um command `malloc`, `free`, ou operações de desreferência são um objeto dinâmico;
- **VALID_OBJECT:** CV utilizado para checar se os argumentos de um command `malloc`, `free`, ou operações de desreferência são um objeto dinâmico.

O Symbiotic 3 (CHALUPA et al., 2016) é uma ferramenta de verificação de *software* criada utilizando instrumentação do código para rastreio de uma máquina de estado finito, e execução simbólica com o Klee para encontrar violações de propriedades de segurança. O Symbiotic 3 não utiliza um *model checker* para gerar as propriedades de segurança, ele utiliza o Klee e um *slicer*, pois a execução do Klee gera um espaço de estados muito grande. A Figura 19 exibe o fluxo do Symbiotic 3 que consiste em: (1) Verifica se a propriedade a ser verificada é suportada; (2) Traduzir um código C pré-processado para LLVM; (3) Verifica se alguma função não suportada é utilizada no código; (4) É feito um processo de instrumentação e linkedição no código LLVM; (5) Aplicam-se otimizações de código; (6) Executa-se o Klee . Similarmente ao trabalho do Chalupa et al. (2016), a nova versão do Map2Check utliliza o Klee para execução simbólica.

Figura 19 – Fluxo do Symbiotic 3



Fonte: (CHALUPA et al., 2016)

SMACK é uma ferramenta que como a Figura 21 mostra, recebe um programa C, converte para LLVM (CARTER et al., 2016), aplica otimizações e traduz para um código Boogie onde são aplicadas análises. O Boogie é uma linguagem feita para aliviar a complexidade de modelar novas linguagens e implementar novos algoritmos de verificação (LEINO, 2008). O site Rise4Fun[3] contém um compilado de programas Boogie, a Figura 20 contém um exemplo de um programa que é uma adaptação do programa McCarthy-91[4]. No exemplo, é criado um procedimento `F` que recebe um valor n do tipo inteiro e retorna um valor r do tipo inteiro (linha 1), logo em seguida nas linhas 2 e 3 são feitas as seguintes assertivas: Se $100 < n$ então $r = n - 10$ (linha 2) senão $r = 91$ (linha 3), vamos considerar essas assertivas como assertivas A e B respectivamente. O procedimento contém dois caminhos básicos baseado na condição de se $100 < n$ (linha 5), se ocorrer então r recebe o valor de $n - 10$ (linha 6) senão r vai receber o resultado da operação $F(n + 11)$ e logo em seguida vai receber o valor de $F(r)$. A assertiva A é facilmente vista pelas linhas 5 e 6, porém a segunda assertiva não é facilmente verificada.

O Map2Check v7 não utiliza a linguagem Boogie, como representação intermediária

[3] <http://rise4fun.com/>
[4] <http://rise4fun.com/Boogie/McCarthy-91>

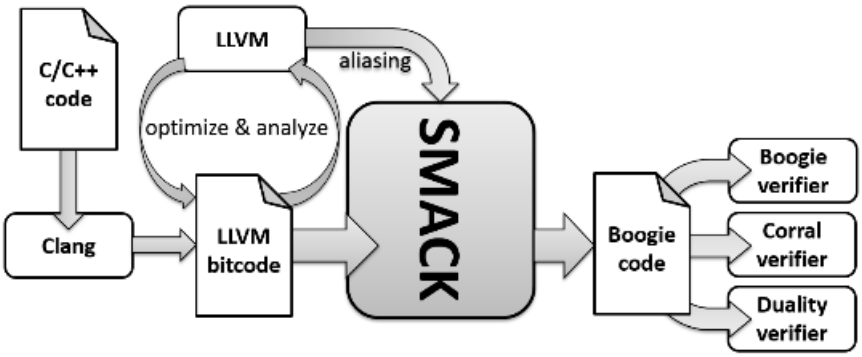
Figura 20 – Programa com desalocação inválida

```
1  procedure F(n: int) returns (r: int)
2    ensures 100 < n ==> r == n - 10;
3    ensures n <= 100 ==> r == 91;
4  {
5    if (100 < n) {
6      r := n - 10;
7    } else {
8      call r := F(n + 11);
9      call r := F(r);
10   }
11 }
```

Fonte: Adaptado de (Microsoft Corporation, 2017)

para verificação dos programas analisados, os algoritmos de verificação são implementados em C e compilados para LLVM IR, sendo virtualmente compatível com quaisquer linguagens que compilem pra LLVM IR e com métodos como o do Symbiotic 3.

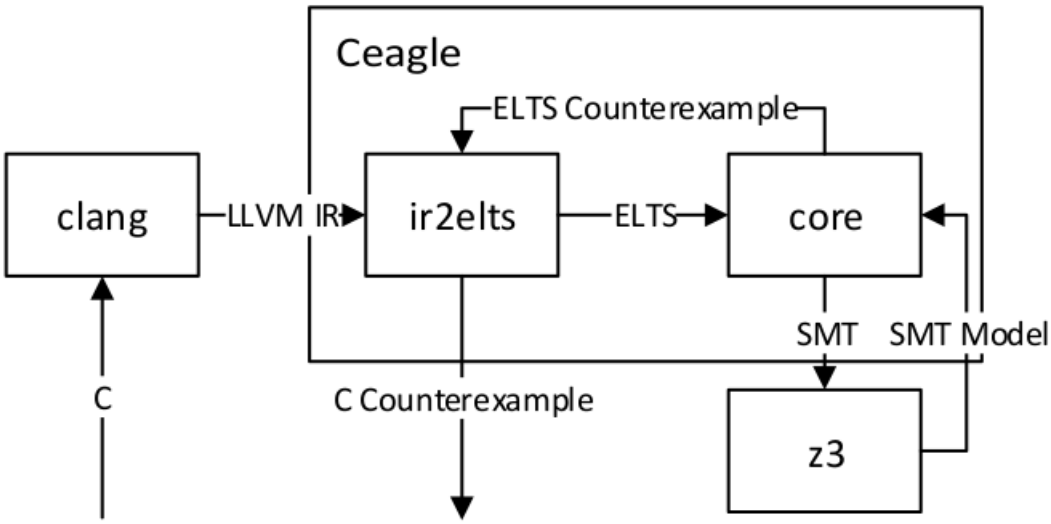
Figura 21 – Fluxo do SMACK



Fonte: (CARTER et al., 2016)

O Ceagle verifica o código C convertendo-o para LLVM IR onde ele passa por duas etapas, o módulo `ir2elts` converte de LLVM IR para o modelo de programa ELTS que é uma lingaguem de modelagem de programas com semântica equivalente a linguagem C. No módulo `core` são gerados as condições e após a resolução delas é verificado se o programa viola alguma propriedade de segurança. A Figura 22 mostra o fluxo descrito do Ceagle (WANG et al., 2016). O Map2Check v7 faz toda sua análise em LLVM IR, instrumentando as assertivas no próprio código intermediário, sendo necessário o uso da execução simbólica.

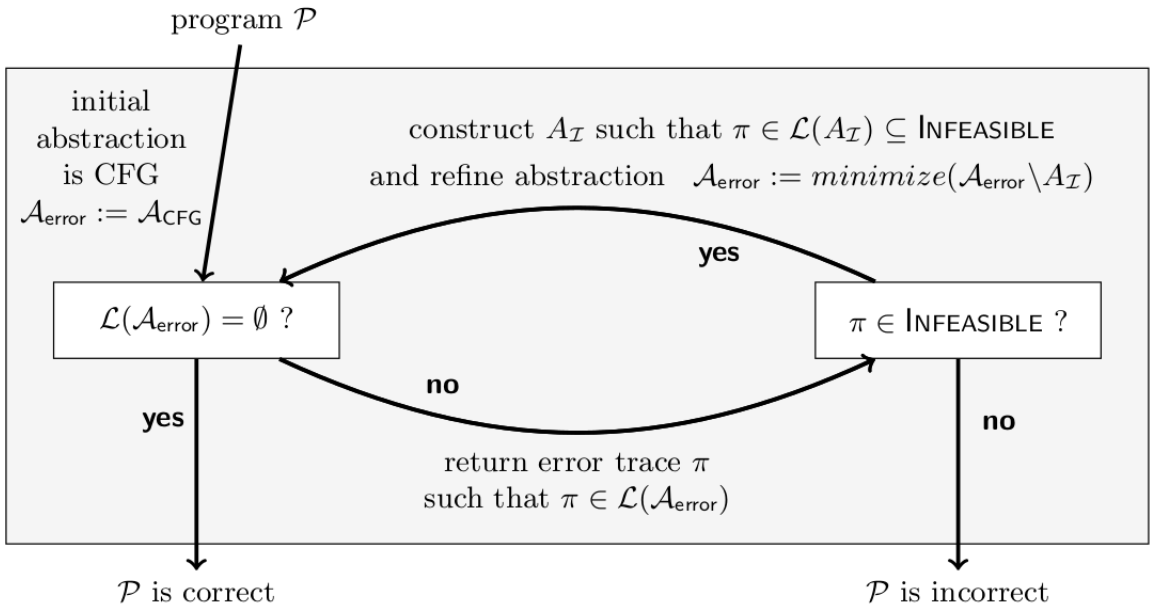
Figura 22 – Fluxo do Ceagle



Fonte: (WANG et al., 2016)

O UltimateAutomizer é um verificador de *software* baseado em teoria dos autômatos, primeiramente traduzindo o programa para Boogie e depois traduzindo para um grafo de controle de fluxo interprocedural que representa todo o programa, não somente métodos. No algoritmo da ferramenta (ver Figura 23), o programa é representado como um autômato que aceita todos os rastros de erro do programa. Um rastro de erro é um caminho no grafo de controle de fluxo que leva a uma localização de erro. Se todos os rastros de erro são impossíveis em relação as semânticas do programa, o programa está correto (HEIZMANN et al., 2013). O UltimateAutomizer diferentemente do Map2Check é uma ferramenta com foco em provar a corretude de um programa, já o Map2Check é uma ferramenta de *bug-hunting* ou seja, as técnicas implementadas focam em identificar a violação de propriedades.

Figura 23 – Verificação do Ultimate Automizer



Fonte: (HEIZMANN et al., 2013)

O PredatorHP faz sua análise usando grafos de memória simbólica (SMGs) que são grafos orientados com dois tipos principais de vértices e dois tipos principais de arestas. Os vértices podem ser divididos em objetos e valores. Objetos podem ser divididos ainda em regiões - representando blocos concretos de memórias de memória alocada na pilha, no *heap*, ou estaticamente - e listas simples ou duplamente encadeadas, que representam de uma forma abstrata sequências interruptas de regiões. As arestas podem ser divididas em: contém valor e aponta para. O primeiro representa valores armazenados na memória alocada, o outro representa alvos de valores de ponteiro. Tanto os vértices e arestas são anotados com informações como tamanho dos objetos, valores armazenados . Através da execução simbólica é possível determinar valores não explícitos, e assim fazer a verificação das propriedades (KOTOUN et al., 2016). O Map2Check v7 similarmente ao método proposto por Kotoun et al. (2016) registra todas as operações de memória, porém utilizando listas para modelar o programa.

4 MÉTODO PROPOSTO

Este Capítulo descreve o método proposto neste trabalho, baseado em LLVM, para aprimoramento do método Map2Check para geração de casos de teste de gerenciamento de memória de programas em C.

4.1 Método Map2check LLVM

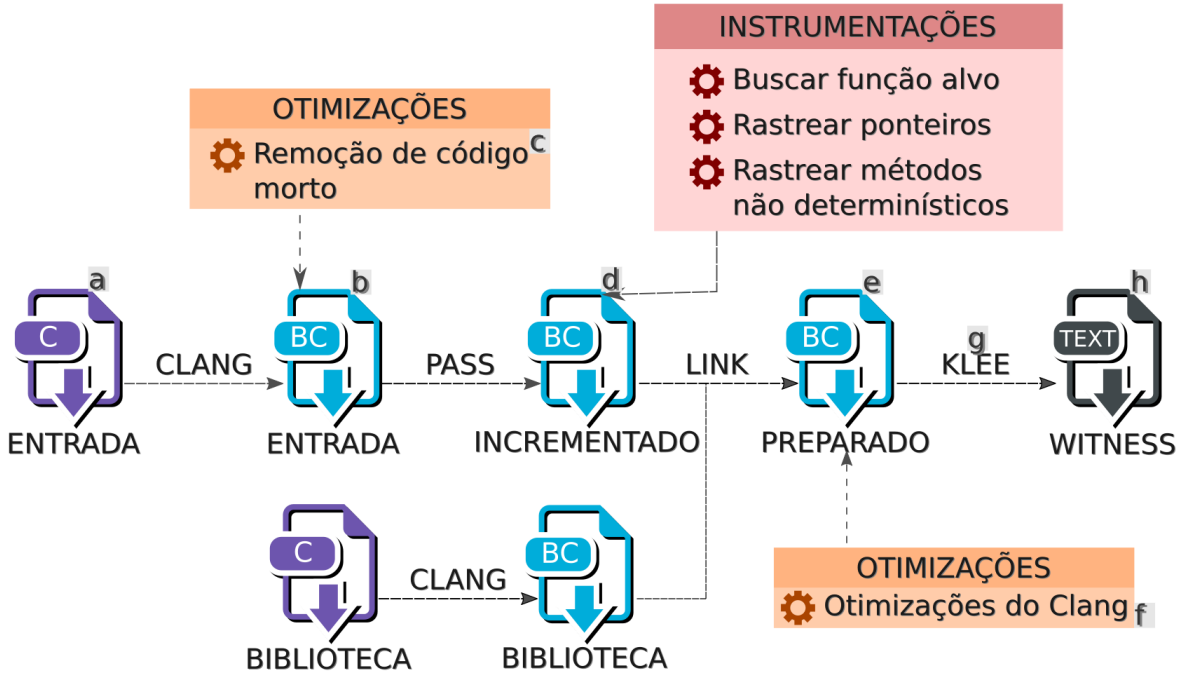
A Figura 24[1] demonstra o fluxo do método proposto que consiste em: (a) Receber como entrada o código fonte de um programa escrito em C; (b) Converter o programa em C para um representação intermediária, neste caso LLVM IR (LATTNER; ADVE, 2004); (c) Aplicação de otimizações de código que não removam informações úteis para geração de um contraexemplo, por exemplo, remoção de código morto (LATTNER; ADVE, 2004); (d) Instrumentação do código analisado com funções para o rastreamento de ponteiros (endereços de memória), assertivas e funções para execução simbólica com o *framework* Klee; (e) Conexão entre o código instrumentado com uma biblioteca desenvolvida para dar suporte a execução das funções instrumentadas; (f) Otimizações no código após a instrumentação do código analisado; (g) Execução simbólica do código analisado instrumentado em LLVM IR utilizando o Klee para geração de dados entradas para as funções instrumentadas e validação das assertivas com as propriedades de segurança; e (h) Geração de um contraexemplo no formato de *witness* caso uma propriedade seja violada, ou seja, uma assertiva falhe durante a verificação do código analisado.

A nova versão do Map2Check (chamada de Map2Check v7) utiliza execução simbólica para percorrer o espaço de estados do programa e verifica propriedades de segurança mapeando e validando todas as operações que ocorrem na memória durante a execução do programa. A nova versão do Map2Check está disponível em: <https://github.com/hbgit/Map2Check/tree/map2checkllvm>. A versão já tem suporte experimental a geração de um *witness* de corretude, porém ainda não há suporte para geração de invariantes.

As propriedades já suportadas pelo novo Map2Check consistem em: erros de desalocação inválida, vazamentos de memória, desreferência de ponteiros e de alcançabilidade de funções, além de gerar um rastreamento do erro reportado, contendo as informações necessárias sobre como o erro aconteceu. Esse rastreamento de memória guia os desenvolvedores diretamente para os locais onde os erros de gerenciamento de memória são identificados.

[1] Ícones dos vetores obtidos de <br.freepik.com>

Figura 24 – Método Map2Check



Fonte: Própria

A implementação da ferramenta do Map2Check LLVM pode ser dividida em múltiplos módulos, como mostrado pela Figura 25, os principais módulos são:

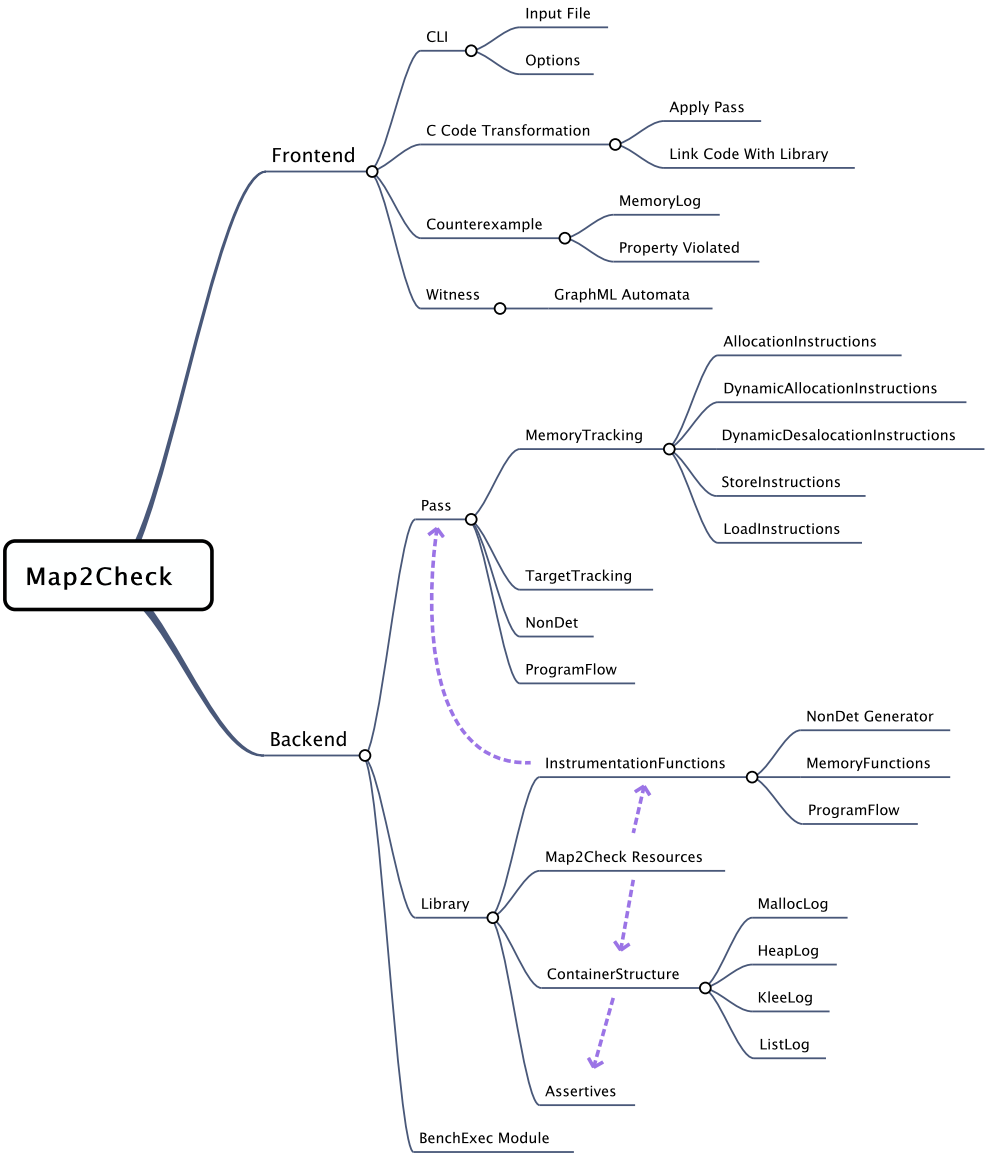
- ***Command-Line Interface* (CLI)**: Interface para o usuário executar o Map2Check LLVM com parâmetros específicos, entre as opções de utilização estão: (a) `target-function`: verifica se a função especificada pelo parâmetro é alcançável (existe algum caminho finito de execução); (b) `generate-witness`: Após o Map2Check verificar que alguma propriedade foi violada, um arquivo contendo o *witness* de violçao é gerado no formato especificado por Beyer et al. (2015a); e (c) `assume-malloc-true`: Assume que todas as funções de alocação dinâmica têm que funcionar, se alguma função gerar erro, o Map2Check para e é incapaz de determinar se o programa viola alguma propriedade.

- **Transformação de código C**: Nesse módulo é feita toda a conversão e adaptação do código original (dado como entrada para o programa, o método será descrito com mais detalhes nas Seções 4.1.1 e 4.1.2.

- **Contraexemplo**: A geração do contraexemplo é feita em um formato próprio apresentando ao usuário em ordem de execução os seguintes estados do programa: Geração de valores não determinísticos, Operações sobre ponteiros e Operações sobre memória dinâmica.

- ***Witness***: A geração do *Witness* é feita no formato da SV-COMP (BEYER et al., 2015a).

Figura 25 – Estrutura Map2Check



Fonte: Própria

- ***Pass***: Módulo responsável pela instrumentação do código utilizando o *framework* do LLVM que é descrito com mais detalhes na subseção 4.1.2.
- ***Benchexec Module***: Módulo responsável pela execução do Map2Check junto à ferramenta Benchexec (BEYER et al., 2015b), essa ferramenta será explicada com mais detalhes no próximo capítulo.

4.1.1 Geração de representação intermediária de código

Para geração do código intermediário (para LLVM-IR) a partir de um código em C o Map2Check utiliza o compilador CLANG como *front-end*. Assim, utilizamos o CLANG com as seguintes opções `-O0 -c -emit-llvm -g`, que significam respectivamente: (a) Não otimizar o código; (b) Não *linkar* com outras bibliotecas; (c) Gerar o LLVM IR; e (d) Adicionar informações de *debug*. Desta forma, a IR do código gerado tem os dados necessários (exemplo, referência ao número da linha do código analisado) para aplicar as análises de forma abrangente e com uma corretude maior.

Figura 26 – Programa com desalocação inválida

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6    int *A = malloc(10);
7    int *B = malloc(10);
8    int x = non_det_int();
9    int y = x * 7 - 15;
10
11   if(y > 14){
12     B = A;
13   }
14
15   free(A);
16   free(B);
17
18   return 0;
19 }
```

Fonte: Própria

Finalizado a geração do código intermediário, o método aplica otimizações na IR (usando o *framework* LLVM), tais como, a remoção de código morto, ou seja, remoção de todas as partes do código inalcançáveis. Assim, o método elimina partes de código que seria desperdiçada na verificação das propriedades, pois o código nunca seria executado, diminuindo o tempo de execução necessário de execução do método. Após isso, o código intermediário está pronto para ser analisado e instrumentado. A Figura 27 mostra o grafo de controle de fluxo (CFG) do código analisado da Figura 26 em LLVM IR.

O programa do CFG da Figura 27, exibe um código em LLVM na versão 3.8.1, o código LLVM IR contém diversos comando e operadores, em especial devemos nos atentar as seguintes estruturas:

Variáveis: Variáveis podem ser nomeadas ou não, caso não tenham nome, um número será utilizado para a representar. Se uma variável nomeada aparecer novamente em outro escopo (porém diferente) o nome será modificado automaticamente (a Figura 28 mostra um exemplo disso). Exemplos de variáveis seriam `%1` e `%x`.

Atribuições: As instruções com operador `store` representam atribuições, onde o primeiro argumento é o valor a ser atribuído e o segundo argumento é a variável com endereço de destino. Essas instruções são importantes para realizarmos o rastreio da memória pois se o segundo argumento for um ponteiro (a API do LLVM nos permite diferenciar diretamente) significa que devemos acompanhar seus valores.

Chamadas de função: O operador `call` define chamadas de funções, esse operador é útil para nossa análise por ser um jeito simples de rastreio dos métodos de diferentes linguagens (desde seja adicionado o suporte).

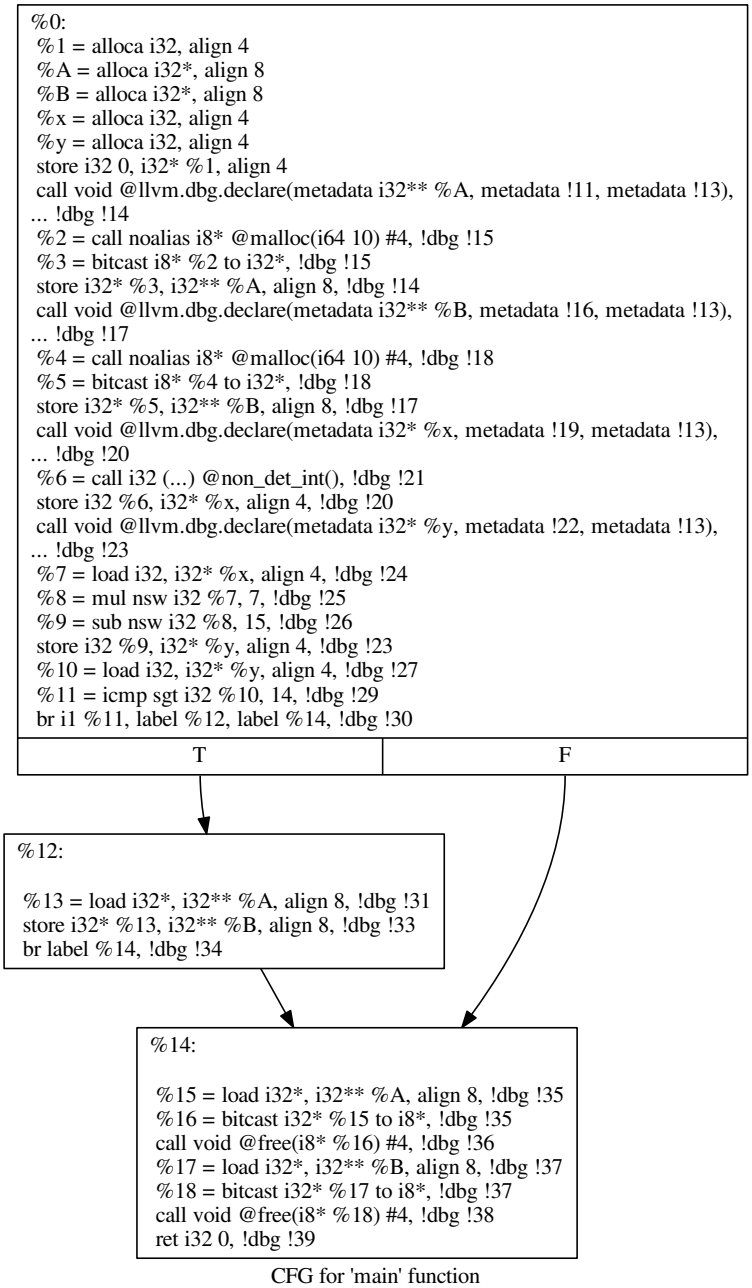
Metadados: Os metadados contém informações úteis referentes a instrução atual antes da compilação, por exemplo, número da linha e escopo. Essas informações são úteis para geração do contra-exemplo.

4.1.2 Transformação de código

As transformações no código analisado ocorrem na representação intermediária gerado pelo método, no formato LLVM IR (seguindo o padrão em *Static Single Assignment* (SSA)). As transformações aplicadas consistem em adicionar funções da biblioteca para rastreamento de memória e assertivas (ver Seção 4.1.3), sendo que os argumentos para as funções e assertivas são obtidos analisando as instruções no código intermediário.

A representação intermediária do código analisado é lido para identificar cada tipo de instrução como declarações, atribuições e chamadas de funções para executar uma instrumentação de código baseada em funções suportadas por uma biblioteca do método proposto. Essas

Figura 27 – CFG do programa original



CFG for 'main' function

Fonte: Própria

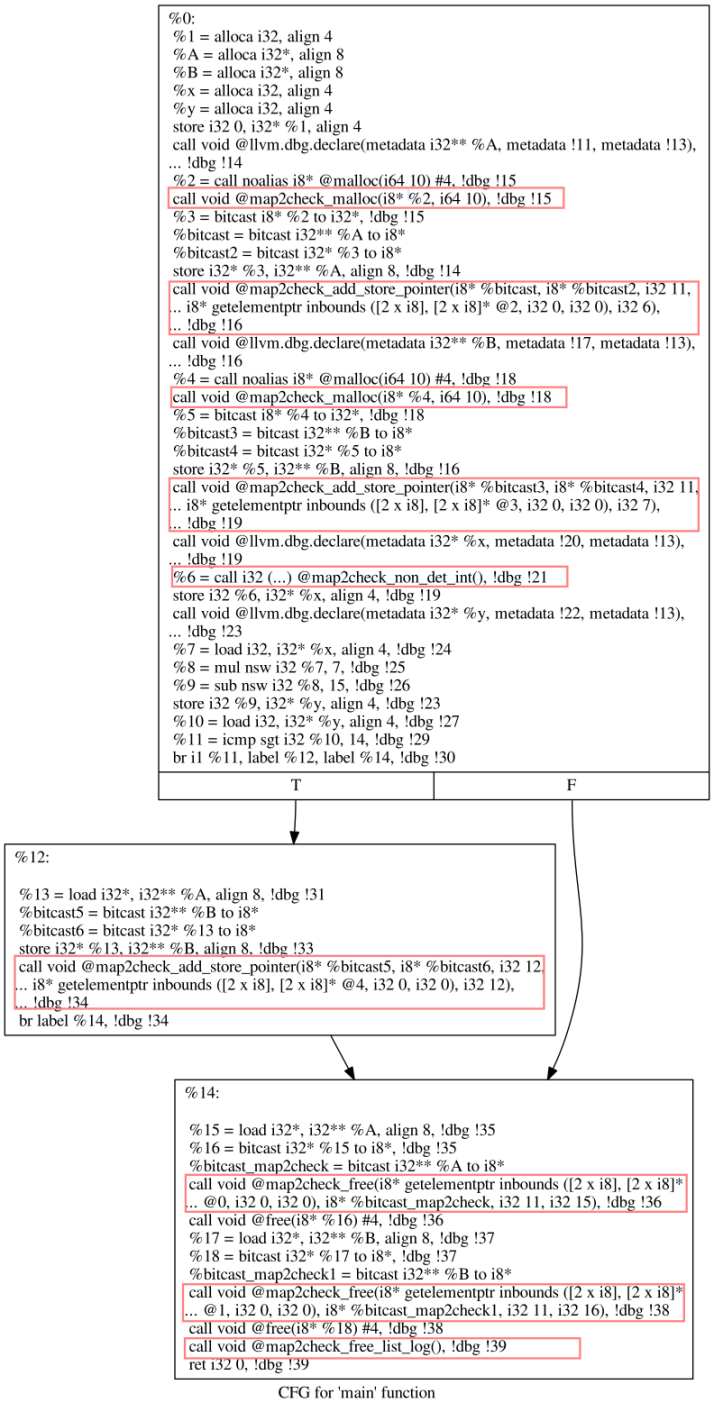
Figura 28 – Exemplo de programa com variável com mesmo nome em escopos diferentes

```
1 int x;
2 {
3     int x;
4 }
```

Fonte: Própria

instrumentações tem os seguintes objetivos: (a) Buscar uma função alvo, ou seja, uma chamada de uma função específica ou mesmo uma localização de erro no código; (b) Rastrear operações de I/O sobre a memória; (c) Rastrear funções não determinísticas; (d) Validar operações sobre ponteiros; e (e) Liberar todos os recursos alocados para a validação. A Figura 29 exibe o CFG do programa da Figura 26 após a instrumentação.

Figura 29 – CFG do programa após instrumentação



Fonte: Própria

Visando automatizar a instrumentação de código é utilizado o algoritmo da Figura 30. Como um dos objetivos do LLVM é ser uma suíte para análises e transformações de código (LATTNER; ADVE, 2004), o LLVM dispõe de um mecanismo chamado *pass* que é uma forma modularizada de implementar as análises ou transformações em um código LLVM IR,

não precisando ser integrado ao LLVM, pois basta compilar o *pass* separadamente. Para a implantação do algoritmo foram utilizados três *FunctionPass* onde o algoritmo é executado em cada função do programa. O algoritmo consiste em checar todas as instruções de todos os blocos básicos (linha 2), então analisa a instrução e baseado na análise (linhas 4,9,10,12,14,16 e 19), instrumenta a função de rastreamento ou assertiva com a sua respectiva propriedade de segurança (linhas 7,11,13,15,18,21). Caso seja a última instrução da função `main` é adicionado uma instrução de liberação de recursos (linha 24).

A complexidade do tempo de execução do algoritmo, na Figura 30 é $O(n \times m)$ onde n é o número de instruções do código intermediário e m é a quantidade total de funções do programa analisado a ser comparada com uma função alvo a ser rastreada.

Figura 30 – Algoritmo utilizado para instrumentação

```
1:  FUNÇÃO INSTRUMENTAÇÃO(Função)                                  ▷ O(n × m)
2:      PARA TODO Instrução ∈ Função FAÇA                          ▷ O(n)
3:          ESCOLHA Instrução.tipo FAÇA
4:              CASO CallInst
5:                  i ← (CallInst) Instrução
6:                  SE i.funcao.nome ∈ ListaFunçõesAlvo ENTÃO      ▷ O(m)
7:                      InstrumentarFunçãoAlvo()
8:                  FIM SE
9:                  ESCOLHA i.funcao.nome FAÇA
10:                     CASO matchRegex("__VERIFIER_NONDET_")
11:                         InstrumentarKlee()
12:                     CASO "free"
13:                         InstrumentarDesalocação()
14:                     CASO "malloc"
15:                         InstrumentarAlocação()
16:             CASO StoreInst
17:                 i ← (StoreInst) Instrucao
18:                 InstrumentarOperacaoEscritaMemoria()
19:             CASO LoadInst
20:                 i ← (LoadInst) Instrucao
21:                 InstrumentarOperacaoLeituraMemoria()
22:     FIM PARA
23:     SE Função.nome = "main" ENTÃO
24:         InstrumentarLiberaracaoRecursos()
25:     FIM SE
26: FIM FUNÇÃO
```

Fonte: Própria

4.1.3 Biblioteca para rastreamento de memória e assertivas

Visando prover suporte a verificação das propriedades abordadas pelo Map2Check v7, que utiliza instrumentação de funções foi necessário complementar o método do Map2Check v6 (ROCHA et al., 2015), implementando uma biblioteca com as seguintes capacidades: (a) Criar um rastreamento das operações de ponteiros; (b) Execução de assertivas baseada em operações com listas que contém informações sobre os programas analisados como: endereço de memória, estado da memória (estático ou dinâmico); (c) Criar rastreamento das operações dinâmicas da memória, alocação, desalocação, leitura e escrita; (d) Gerar informações de depuração; e (e) Manter o fluxo original do programa inalterado.

A biblioteca do método Map2Check é escrita em C, e somente então compilada para LLVM IR, por C ser uma linguagem de alto nível isso trás vantagens durante o processo de desenvolvimento, como manutenabilidade e modularização. Logo, após a compilação da biblioteca de C para LLVM IR é feito o processo de linkedição com o código analisado ser instrumentado.

4.1.3.1 Rastreamento de memória

Para rastreio da memória são utilizadas as seguintes estruturas para analisar o estado atual da memória:

Heap Log: São anotados todos os endereços alocados e seus tamanhos na memória *heap*.

Malloc Log: São anotados todos os endereços alocados/desalocados dinamicamente, seus tamanhos e se o endereço está liberado ou não.

List Log: São anotadas todas as operações sobre ponteiros, contendo linha, escopo, nome da variável, endereço apontado, endereço do ponteiro e estado do endereço apontado.

A utilização dessas estruturas para a verificação do código será descrita nas próximas seções.

4.1.3.1.1 Memória *Heap*

Os endereços anotados da memória heap são: argumentos de funções, funções e variáveis. O programa da Figura 31 exibe um programa contendo funções e variáveis, a instrumentação dele ficaria como no programa da Figura 32, onde os parâmetros da função `map2check_alloca` são: nome da variável, endereço, tamanho, tamanho da primitiva, linha onde a operação ocorreu e escopo (nos programas das figuras foram escolhidos de forma aleatória).

Figura 31 – Programa para demonstrar instrumentação de memória Heap

```
1  int foo(int a, int b) {
2    return 2;
3  }
4
5  int main() {
6    int a;
7    return 0;
8  }
```

Fonte: Própria

Figura 32 – Programa para demonstrar instrumentações de Heap

```
1  int foo(int a, int b) {
2    map2check_alloca("a", &a, 4, 4, 1, 8);
3    map2check_alloca("b", &b, 4, 4, 1, 8);
4    return 2;
5  }
6
7  int main() {
8    map2check_function("foo", &foo);
9    int a[5];
10   map2check_alloca("a", a, 20, 4, 6, 1);
11   return 0;
12 }
```

Fonte: Própria

4.1.3.1.2 Memória Dinâmica

Para memória dinâmica, são buscadas instruções de alocação/desalocação dinâmica e são instrumentados os métodos para adicionar ao `Malloc Log`. Os métodos além de anotarem a operação realizam algumas assertivas. A Figura 33 e a Figura 34 mostram um exemplo de como a instrumentação ocorre.

Figura 33 – Programa para demonstrar instrumentações para anotar endereços dinâmicos

```
1 int main() {
2   int *pointer = malloc(4);
3   int *pointer2 = malloc(8);
4   free(pointer);
5 }
```

Fonte: Própria

Figura 34 – Programa após instrumentações para anotar endereços dinâmicos

```
1 int main() {
2   int *pointer = malloc(4);
3   map2check_malloc(pointer, 4);
4   int *pointer2 = malloc(8);
5   map2check_malloc(pointer2, 8);
6   free(pointer);
7   map2check_free("pointer", pointer, 1, 4, "main");
8 }
```

Fonte: Própria

4.1.3.2 Rastreio de funções não determinísticas

O rastreio de funções não determinísticas (funções que retornam um valor que pode variar durante a execução do programa) ocorre ao verificar se a instrução é uma chamada de um método não determinístico previamente definido que retorna um valor arbitrário de um tipo indicado (BEYER, 2016). Identificada a função não determinística partir é feito uma instrumentação de uma função para o Klee que irá gerar um valor baseado em execuções simbólicas, e o valor gerado será anotado na estrutura `KleeLog` que é uma estrutura utilizada para armazenar todas os valores gerados para operações não determinísticas ocorridas.

A instrumentação de função para o Klee consiste em criar uma nova variável no programa analisado com o tipo igual ao tipo do não-determinismo identificado (instrumenta-se um inteiro caso seja pedido um inteiro não-determinístico), dessa forma é solucionado um dos possíveis problemas que seria a chamada de um método com um valor não determinístico (apenas com utilização de variáveis temporárias) como: `if(non_det_int())`, durante a execução de código o valor de dentro do `if` será armazenado em uma variável temporária. A Figura 35 exibe um programa com uma chamada de método não-determinístico e a Figura 36 mostra como ficaria após instrumentado. Para utilização do Klee é necessário informar quais as variáveis devem assumir valores simbólicos, na Figura 37 é exemplificado como fazer isso para uma variável, bastando usar a função `klee_make_symbolic` e passando como argumentos o endereço da

variável, o tamanho em *bytes* e um *string* para representar o nome dela (não precisa ser igual ao nome da variável e pode ser repetida), portanto a implementação do método de não determinismo foi basicamente criar uma nova variável e a transformar em um valor simbólico, a Figura 38 exibe a implementação do método.

Figura 35 – Programa com método não-determinístico

```
1 int main() {
2   if (non_det_int()) {
3     return 1;
4   }
5   return 0;
6 }
```

Fonte: Própria

Figura 36 – Programa instrumentado com Klee

```
1 int main() {
2   if (map2check_non_det_int()) {
3     return 1;
4   }
5   return 0;
6 }
```

Fonte: Própria

Figura 37 – Exemplo de utilização do Klee

```
1 int main() {
2   int a;
3   klee_make_symbolic(&a, 4, "a");
4   if(a) {
5     return -1;
6   }
7   return 0;
8 }
```

Fonte: Própria

Figura 38 – Implementação do método que instrumenta o Klee

```
1 int map2check_non_det_int() {
2   int non_det;
3   klee_make_symbolic(&non_det,
4                      sizeof(non_det),
5                      "non_det_int");
6   return non_det;
7 }
```

Fonte: Própria

4.1.3.3 Rastreio de funções alvo

Segundo Beyer (2016) apartir de um conjunto de estados iniciais de um programa na chamada da função `main`. A definição $LTL(f)$ especifica que a fórmula f é verdadeira para cada um dos estados iniciais. Assim, a fase de instrumentação consiste em localizar a função alvo e logo antes dela, adicionar uma função da biblioteca, assim verificando se ao longo da execução essa função foi chamada.

Para exemplificar isso temos a Figura 39 onde temos a função não determinística `non_det_int()` e a função alvo `TARGET()`, observemos que: (a) a linha 2 contém uma chamada não determinística; e (b) a função alvo se encontra na linha 4. A Figura 40 exemplifica a instrumentação, nesse exemplo, é assumido que o escopo é 1, esse escopo é determinado durante a compilação. O método instrumentado `map2check_target_function` ao ser executado, demostra que a localização seria alcançada, injetando uma assertiva de erro na localização onde ocorre a chamada da função alvo. No exemplo da Figura 39, poderia ser gerado o valor 11, o que faria o programar alcançar a função de erro.

Figura 39 – Exemplo de função alvo

```
1 int main() {
2     int x = non_det_int();
3     if (x > 10) {
4       TARGET();
5     }
6 }
```

Fonte: Própria

Figura 40 – Exemplo de função alvo após instrumentação (assumindo escopo 1)

```
1 int main() {
2     int x =  map2check_non_det_int();
3     if (x > 10) {
4       map2check_target_function("main", 1, 4);
5       TARGET();
6     }
7 }
```

Fonte: Própria

4.1.4 Verificação das propriedades de segurança

A verificação do programa analisado consiste em determinar se o programa satisfaz ou viola suas propriedades de segurança, no caso do Map2Check as assertivas instrumentadas. Assim, caso uma propriedade seja violada então é gerado um contraexemplo e uma *witness*. No Map2Check v7, o contraexemplo contém informações referentes aos: valores não determinísticos gerados e o estado da memória após cada operação. Já o *witness* anota apenas o valores não determinísticos gerados para a violação ser validada através de um *witness validator*(BEYER et al., 2015a).

4.1.4.1 Desalocação Inválida

Uma desalocação inválida ocorre quando o programador tenta liberar um endereço que não está alocado na memória dinâmica. Para validar esse tipo de operação, o Map2Check insere uma assertiva antes da chamada de um comando`free` da linguagem C.

A assertiva é descrita no algoritmo da Figura 41, o objetivo é verificar se o endereço a ser liberado está contido no `Malloc Log`, caso ele esteja contido, verificar se ele já esta liberado. Assim a assertiva determina se houve uma desalocação inválida. O algoritmo a partir da linha 3 percorre todo o `Malloc Log` buscando algum elemento que aponte para o mesmo lugar do argumento de entrada, caso não encontre a linha 14 retorna falso, indicando um erro. Já caso o endereço seja encontrado, é feita uma checagem simples verificando se o endereço já esta liberado ou não (linha 7) e retornando se é válido ou não (linhas 8 e 10). A complexidade do algoritmo é $O(m)$ onde m é o número de elementos do `Malloc Log`.

Figura 41 – Algoritmo para validar desalocação de endereço

```
1:  FUNÇÃO IsValidFree(addr, mallocLog)
2:      i ← tamanhoDoLog(mallocLog) − 1
3:      ENQUANTO i ≥ 0 FAÇA                                  ▷ O(m)
4:          row ← mallocLog[i]
5:          i ← i − 1
6:          SE (row.addr = addr) ENTÃO                       ▷ O(1)
7:              SE row.isFree ENTÃO                          ▷ O(1)
8:                  return FALSE
9:              SENÃO
10:                 return TRUE
11:             FIM SE
12:         FIM SE
13:     FIM ENQUANTO
14:     return FALSE
15: FIM FUNÇÃO
```

Fonte: Própria

4.1.4.2 Vazamento de memória

Durante a programação o desenvolvedor aloca recursos de memória do computador, devendo após terminar sua utilização o desalocar, um vazamento de memória ocorre quando o programador por algum motivo esquece de liberar esse recurso previamente alocado na memória (CLAUSE; ORSO, 2010). Para validar o programa, a assertiva percorre o `Malloc Log` buscando referências não liberadas. Essa assertiva só é validada ao final do programa.

A Figura 42 descreve o algoritmo utilizado para a verificação de vazamentos de memória. São percorridos todos os elementos do `Malloc Log` (linha 3), verifica-se se o endereço do elemento atual esta liberado (linha 6), caso não esteja houve um vazamento de memória (linha 7), caso não tenha havido nenhum erro e todos os elementos foram percorridos, o programa não teve vazamentos (linha 10). A complexidade do algoritmo é $O(m)$, onde m é o número de elementos do `Malloc Log`.

Figura 42 – Algoritmo para verificar vazamentos de memória

```
1:  FUNÇÃO IsFalseMemtrack(heapLog)
2:      i ← tamanhoDoLog(mallocLog) − 1
3:      ENQUANTO i ≥ 0 FAÇA                                   ▷ O(m)
4:          row ← mallocLog[i]
5:          i ← i − 1
6:          SE ¬(row.isFree) ENTÃO                            ▷ O(1)
7:              return TRUE
8:          FIM SE
9:      FIM ENQUANTO
10:     return FALSE
11: FIM FUNÇÃO
```

Fonte: Própria

4.1.4.3 Desreferência de ponteiro

Uma falha de desreferência ocorre quando o programador faz alguma operação que envolve a utilização de um espaço de memória que não foi previamente alocado nem dinamicamente nem pela memória *heap*. O Map2Check verifica as seguintes situações:

- Quando carrega-se um valor de um endereço de memória não alocado, ou o extrapolando. O programa da Figura 43 contém esse erro, pois a variável b tenta carregar um valor de um endereço inválido. Exemplo, um inteiro de 4 bytes sendo carregado de um endereço que só tem 2 bytes disponíveis.
- Quando armazena-se um valor em um endereço de memória não alocado, ou o extrapolando. O programa da Figura 44 contém esse erro, pois a variável $a[5]$ não é um endereço válido para salvar. Exemplo, um inteiro de 4 bytes sendo armazenado de um endereço que só tem 2 bytes disponíveis

Figura 43 – Exemplo de programa com erro de Deref ao carregar

```
1 int a[5];
2 int b = a[5];
```

Fonte: Própria

Figura 44 – Exemplo de programa com erro de Deref ao salvar

```
1 int a[5];
2 a[5] = 7;
```

Fonte: Própria

O Algoritmo da Figura 45 verifica endereços no `Malloc Log` e o Algoritmo da Figura 46 verifica endereços no `Heap Log`. Os algoritmos funcionam de maneira semelhante, com a diferença que ao checar no `Malloc Log` é verificado se o endereço está alocado ou não. Contudo, no algoritmo do `Malloc Log` primeiro são percorridos todos os elementos (linha 3), na linha 6 é verificado qual o endereço máximo que a variável a ser lida ou escrita pode ter, exemplo, uma variável de 4 bytes sendo escrita em um endereço de 8 bytes no endereço 0x0 só pode ser escrita no máximo no endereço 0x4. Após isso, checa-se se a variável está entre o endereço inicial e esse endereço limite de leitura ou escrita (linha 7), caso esteja, é verificado se o endereço esta liberado ou não (linha 8), se estiver, houve um erro de desreferência (linha 9), se não, não houve erro (lina 11). Caso todos os elementos sejam percorridos e o algoritmo não encontrou o endereço, houve um erro (linha 15).

Figura 45 – Algoritmo para verificar endereço no MallocLog

```
1:  FUNÇÃO IsDerefAddrMalloc(addr, size, mallocLog)
2:      i ← tamanhoDoLog(mallocLog) − 1
3:      ENQUANTO i ≥ 0 FAÇA                                   ▷ O(m)
4:          row ← mallocLog[i]
5:          i ← i − 1
6:          addrTop ← row.addr + row.size − size + 1
7:          SE (row.addr ≤ addr) ∧(addr < addrTop) ENTÃO      ▷ O(1)
8:              SE row.isFree ENTÃO                           ▷ O(1)
9:                  return TRUE
10:             SENÃO
11:                 return FALSE
12:             FIM SE
13:         FIM SE
14:     FIM ENQUANTO
15:     return TRUE
16: FIM FUNÇÃO
```

Fonte: Própria

Figura 46 – Algoritmo para validar endereço no HeapLog

```
1: FUNÇÃO IsDerefAddrHeap(addr, size, heapLog)
2:     i ← tamanhoDoLog(heapLog) − 1
3:     ENQUANTO i ≥ 0 FAÇA                                   ▷ O(n)
4:         row ← mallocLog[i]
5:         i ← i − 1
6:         addrTop ← row.addr + row.size − size + 1
7:         SE (row.addr ≤ addr) ∧(addr < addrTop) ENTÃO      ▷ O(1)
8:             return FALSE
9:         FIM SE
10:    FIM ENQUANTO
11:    return TRUE
12: FIM FUNÇÃO
```

Fonte: Própria

5 RESULTADOS EXPERIMENTAIS

Essa seção descreve o planejamento, projeto, execução e análise dos resultados de um estudo experimental para avaliar o método proposto neste trabalho, para verificar de forma automática propriedades de segurança de memória, quando aplicado em *benchmark* públicos de programas escritos em GNU-C ou ANSI-C.

5.1 Planejamento e projeto dos experimentos

Esta avaliação experimental visa avaliar a habilidade do Map2Check com suporte a LLVM em gerar e verificar casos de testes (assertivas) relacionados a gerência de memória, especificamente desalocações inválidas, desreferência de ponteiros e vazamentos de memória. Neste sentido, foi definido as seguintes questões de pesquisa:

QP1: Os casos de teste gerados pelo Map2Check são suficiente para detectar uma falha de gerência de memória do programa analisado?

QP2: Qual a eficácia do Map2Check para detectar falhas de gerência de memória quando comparado com outras ferramentas?

QP3: O método proposto aprimora a detecção de falhas de gerência de memória quando comparado a versão anterior do Map2Check?

Visando responder essas questões, foram utilizados 328 programas escritos em C da categoria *MemSafety*[1] do *benchmark* da Competição Internacional de Verificação de Software (*Competition on Software Verification*: SV-COMP) (BEYER, 2017). Estes programas são classificados como: `TRUE`, `FALSE(valid-free)`, `FALSE(valid-deref)` ou `FALSE(valid-memtrack)`, significando respectivamente: o programa não contém erros de memória, o programa contém desalocações inválidas, o programa possui falhas de desreferência e o programa contém vazamentos de memória. O SV-COMP é uma competição internacional onde diversas ferramentas são submetidas a *benchmark* públicos (obtidos de diferentes domínios como *drivers* e *kernel* do Linux) e depois avaliadas e pontuadas. Assim, o SV-COMP tem como objetivo avaliar a transferência de tecnologia e comparar verificadores de software no estado da arte em relação à eficácia e eficiência.

Nos *benchmarks* da SV-COMP, alguns programas adotam funções específicas, por exemplo, a categoria de *MemSafety* contém a função `__VERIFIER_nondet_int` que modela valores não determinísticos de inteiros. No Map2Check, foi implementado uma instrumentação

[1] Maiores detalhes em https://sv-comp.sosy-lab.org/2017/results/results-verified/META_MemSafety.table.html

para que o Klee (utilizando execução simbólica) gere os valores necessários para explorar os caminhos de execução do programa e então efetuar a verificação das assertivas nos programas analisados. Adicionalmente, neste experimento avaliamos os resultados do método proposto utilizando o esquema de pontuação da SV-COMP, para os resultados das ferramentas de verificação de software, que conta com um sistema de pontuações baseado em acerto e erros com as seguintes regras para as classificações dos programas:

- **Verdadeiro Correto:** O programa foi analisado e o resultado da verificação determina que nenhuma propriedade foi violada, pontua-se $+2$ pontos.
- **Falso Correto:** Um erro (violação de uma propriedade) foi identificado no programa analisado, pontua-se $+1$ ponto.
- **Verdadeiro Incorreto:** O programa contém um erro, mas a ferramenta não identificou o erro, perde-se -32 pontos.
- **Falso Incorreto:** A ferramenta reporta um erro que não existe no programa analisado, perde-se -16 pontos.
- **Desconhecido:** Quando a ferramenta é incapaz de chegar a uma conclusão sobre o programa, não se pontua.

A execução da ferramenta Map2Check para avaliar os *benchmarks* foi feita utilizando a ferramenta Benchexec (BEYER et al., 2015b) que gerencia, monitora e faz medidas de recursos (memória e tempo de execução pela CPU) utilizados por ferramentas. No Benchexec foram adotas as seguintes configurações para execução do experimento: um tempo máximo de execução (*timeout*) de 15 minutos por programa a ser analisado, ou seja, se o Map2Check não chegar a uma conclusão dentro desse tempo, a execução será interrompida; 5 GB de memória RAM; e 5 núcleos de processador. A máquina utilizada para execução do experimento continha um sistemas operacional Linux Ubuntu 16.10 com 8GB de memória, processador AMD 2.9 GHz com 8 cores de CPU.

A avaliação foi conduzida da seguinte forma: (1) Aplicação do Map2Check v7 sobre os programas do *benchmark* do SV-COMP; (2) Aplicação do Map2Check v6 sobre os programas do *benchmark* do SV-COMP; (3) Comparação dos resultados obtidos (diretamento da página oficial do SV-COMP (BEYER, 2017)) com: PredatorHP, Ultimate Automizer, CBMC, SMACK, Map2Check v6 e Symbiotic 4[2].

5.2 Execução dos experimentos e análise dos resultados

Após a execução dos *benchmarks*, foram obtidos os resultados mostrados na Tabela 1, onde cada linha da tabela contém: (1) nome da ferramenta (Ferramenta); (2) número de pro-

[2] https://sv-comp.sosy-lab.org/2017/results/results-verified/META_MemSafety.table.html

gramas que atendem as especificações e a ferramenta identificou corretamente (Verdadeiro Corretos); (3) número de programas que não atendem as especificações e a ferramenta identificou corretamente (Falso Corretos); (4) número de programas que a ferramenta não identificou um erro para um programa que não atende todas as especificações (Verdadeiro Incorreto); (5) número de programas que a ferramenta identificou um erro para um programa que atende todas as especificações (Falso Incorreto); e (6) número de programas em que a ferramenta falhou para computar a verificação (Desconhecidos);

Tabela 1 – Resultado da experimentação

Ferramenta	Map2Check v7 LLVM	Ultimate Automizer	PredatorHP	Symbiotic 4	SMACK	Map2Check v6	CBMC
Verdadeiro Correto	106	94	103	104	142	138	66
Falso Correto	126	51	116	129	121	24	127
Verdadeiro Incorreto	0	0	0	0	1	88	0
Falso Incorreto	0	0	0	0	10	16	0
Desconhecidos	96	183	109	95	54	62	135
Total Corretos	232	145	219	233	263	162	193
Total Incorretos	0	0	0	0	11	104	0

Fonte: Própria

O Map2Check v7 mesmo sendo uma ferramenta de *bug-hunting* demonstrou bons resultados em relação as demais ferramentas de verificação quando comparado em programas que foram provados corretamente, efetuando um *ranking* das ferramentas, o Map2Check v7 estaria em primeiro lugar no quesito classificação de programas, na Tabela 1 linha Verdadeiro Correto. Fazendo um *ranking* das ferramentas avaliadas, o Map2Check v7 ficaria em terceiro lugar. Este resultado, em parte se explica, pois como a verificação de programas é indecidível, ou seja, os métodos computacionais de verificação de programas são todos parciais ou incompletos, a indecidibilidade sempre é resolvida através de alguma aproximação resultando em que as ferramentas podem sofrer limitações como analisar apenas certas especificações de programas (COUSOT, 2001). O Map2Check v7 não aplica técnicas para aproximar o comportamento de execução do programa, o método proposto basea-se em realizar a execução concreta do programa. Desta forma, o Map2Check v7 tem como vantagem a corretude dos resultados, porém tem-se como desvantagem o seu tempo de verificação.

Em relação a detecção de erros e fazendo um *ranking* das ferramentas avaliadas, utilizando o esquema de pontuação da SV-COMP[3] (ver Seção 5.1), o Map2Check v7 estaria em primeiro lugar (com 338 pontos), em segundo estaria o Symbiotic 4 (com 337 pontos) e em

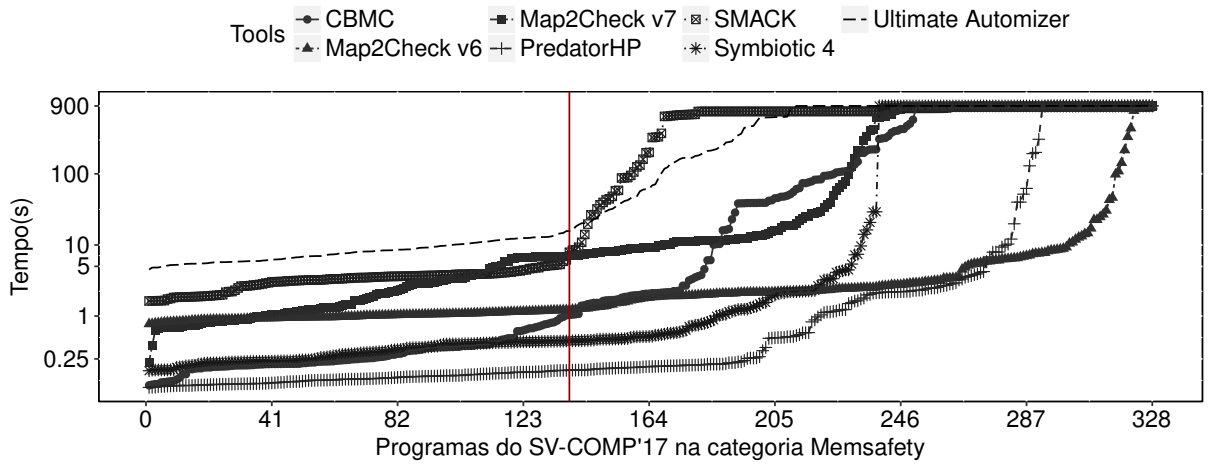
[3] Maiores informações em https://sv-comp.sosy-lab.org/2017/rules.php

terceiro o PredatorHP (com 322 pontos). Entre essas ferramentas nos primeiros lugares, o Symbiotic 4 obteve um melhor resultado no quesito Falso Correto, um motivo seria pela utilização de um *slicer* o que diminui os estados necessários na verificação dos programas analisados, logo reduzindo a exploração de caminhos não necessários para a verificação de dadas propriedades.

O Map2Check v7 não gerou resultados incorretos, como pode ser observado na Tabela 1 linhas Verdadeiro Incorreto e Falso Incorreto, diferentemente das ferramentas SMACK e Map2Check v6. Comparado a versão 7 com 6 do Map2Check, a v7 obteve melhores resultados nos *benchmarks* (v7 com 232 resultados corretos e a v6 com 162) devido a substituição da geração de valores não determinísticos de forma probabilística para execução simbólica, bem como, a utilização da representação intermediária usando LLVM para a intrumentação de código que permitiu um melhor mapeamento da memória. Logo, os casos de teste gerados foram mais eficazes e se pode verificar mais caminhos de execução do programa.

A Figura 47 apresenta um gráfico contendo o tempo de execução das ferramentas da Tabela 1, no gráfico é possível verificar que a performance do Map2Check v7 foi média quando comparada a algumas ferramentas, em parte este fato se explica devido ao método não gera invariantes de programas e nem utiliza um *slicer* de código com isso muitos caminhos e verificações são feitas de forma desnecessária, além de não lidar muito bem com estruturas de repetição de maneira geral. Pode-se perceber também que por volta do programa 138 houve um considerável aumento no tempo de execução de maneira geral.

Figura 47 – Tempo de execução das ferramentas em programas



Fonte: Própria

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi apresentado a importância da verificação de *software* para validar propriedades de segurança no gerenciamento de memória. Foi descrito um método proposto para aprimorar o método Map2Check (ROCHA et al., 2015) além da implementação do mesmo, que é capaz de gerar, verificar e validar propriedades de segurança no gerenciamento de memória em programas em C. Adicionalmente, foi apresentado um avaliação experimental comparado a eficácia da ferramenta em relação a outras ferramentas com propostas similares, utilizando *benchmark* públicos de programa em C de uma competição internacional de verificação de software - SV-COMP'17(BEYER, 2017).

O método proposto apresentado neste trabalho, demonstrou melhorias para o método Map2Check que consistiu em adicionar: 1) a ferramenta Clang como *front-end* para programas em C (LLVM Foundation, 2017); 2) o *framework* LLVM como base para aplicações de transformações de código utilizando a representação intermediária LLVM *bitcode* (LLVM Foundation, 2017); e 3) o Klee para execução simbólica de código baseado em LLVM (CADAR et al., 2008). Assim, o método proposto é capaz verificar e validar propriedades de segurança para validar desalocações inválidas, desreferências de ponteiro e vazamentos de memória. Com base nas melhorias, a nova versão do Map2Check é mais eficiente em relação a ser capaz de lidar com diversas estruturas na limguagen de programação C e gerar contra-exemplos no formato *witness*, descrito em Beyer (2016), e para cada violação de propriedade apresentar um *log* de rastreamento com dados das variaveis envolvidas do programa analisado.

Comparando a versão anterior do Map2Check, no quesito geração de casos de teste e verificação de propriedades de segurança, o novo método apresentou melhorias significativas quanto a eficácia, resultado analisados sobre o *benchmark* da SV-COMP'17 (ver Capítulo 5), isso pode ser justificado pela utilização da execução simbólica em conjunto com a utilização da representação intermediária em LLVM, ao invés da probabilística sem representação intermediária de código, o que fez com que mais casos de teste fossem verificados. E quando comparado com as ferramentas atuais (como SYMBIOTIC 4 (CHALUPA et al., 2016) e PredatorHP (KOTOUN et al., 2016)), foi verificado que o Map2Check foi mais eficaz em apresentar resultados corretos em validar as propriedades de segurança, e por um programa inferior ao SYMBIOTIC 4 no quesito total de programas corretos e sem resultados incorretos. Além disso, o Map2Check não gerou nenhum falso positivo ou falso negativo, quando comparado com as outras ferramentas avaliadas e utilizando o esquema de pontuação do SV-COMP'17 (ver Seção 5.1), o Map2Check obteve a maior pontuação com 338 pontos. Desta forma, os resultados apresentados sugerem que o método proposto é eficaz mesmo quando comparado a outras ferramentas.

Como propostas de trabalhos futuros para a expansão do método proposto, as seguintes funcionalidade poderão ser investigadas e adicionadas: suporte para geração de código fonte com assertivas suportadas por um *framework* de teste de unidade como o CUnit[1]; suporte a programas com paralelismo (INVERSO et al., 2014); geração de invariantes de programas (HENRY et al., 2012); geração de um *witness* de corretude (BEYER, 2016); adoção de um *slicer* de código (CHALUPA et al., 2016); e validação das *witness* geradas (BEYER et al., 2015a). Desta forma, espera-se que o Map2Check seja capaz de verificar e validar programas em C com maior eficácia, assim apresentados melhores resultados em relação a sua precisão e tempo de verificação.

[1] Disponível em http://cunit.sourceforge.net

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-10088-6.

APPEL, A. W. Ssa is functional programming. *SIGPLAN notices*, v. 33, n. 4, p. 17–20, 1998.

BAIER, C.; KATOEN, J.-P. *Principles of Model Checking (Representation and Mind Series)*. [S.l.]: The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BENSALEM, S.; LAKHNECH, Y. Automatic generation of invariants. *Formal Methods in System Design*, v. 15, n. 1, p. 75–92, 1999. ISSN 1572-8102. Disponível em: <http://dx.doi.org/10.1023/A:1008744030390>. Acesso em: 25 jul. 2016.

BEYER, D. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2016. p. 887–904.

BEYER, D. Software verification with validation of results. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2017. p. 331–349.

BEYER, D. et al. Witness validation and stepwise testification across software verifiers. In: ACM. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2015. p. 721–733.

BEYER, D.; LÖWE, S.; WENDLER, P. Benchmarking and resource measurement. In: *Model Checking Software*. [S.l.]: Springer, 2015. p. 160–178.

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 (OSDI)*. USENIX Association, 2008. p. 209–224. Disponível em: <http://dl.acm.org/citation.cfm?id=1855741.1855756>.

CADAR, C. et al. Symbolic execution for software testing in practice: Preliminary assessment. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011. (ICSE '11), p. 1066–1071. ISBN 978-1-4503-0445-0. Disponível em: <http://doi.acm.org/10.1145/1985793.1985995>.

CADAR, C.; SEN, K. Symbolic execution for software testing: Three decades later. *Commun. ACM*, ACM, New York, NY, USA, v. 56, n. 2, p. 82–90, fev. 2013. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/2408776.2408795>.

CARTER, M. et al. SMACK software verification toolchain. In: VISSER, W.; WILLIAMS, L. (Ed.). *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion*. [S.l.]: ACM, 2016. p. 589–592.

CHALUPA, M. et al. Symbiotic 3: New slicer and error-witness generation. In: ____. *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software,*

ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. p. 946–949. ISBN 978-3-662-49674-9. Disponível em: <http://dx.doi.org/10.1007/978-3-662-49674-9_67>.

CHENG, W. et al. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In: *11th IEEE Symposium on Computers and Communications (ISCC'06)*. [S.l.: s.n.], 2006. p. 749–754. ISSN 1530-1346.

CLARK, E. M. *25 Years of Model Checking: History, Achievements, Perspective*. [S.l.]: Springer-Verlag Berlin Heidelberg, 2008. ISBN 978-3-540-69850-0.

CLARKE, E. M.; EMERSON, E. A.; SIFAKIS, J. Model checking: Algorithmic verification and debugging. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 11, p. 74–84, nov. 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1592761.1592781>.

CLAUSE, J.; ORSO, A. Leakpoint: Pinpointing the causes of memory leaks. In: PROCEEDINGS OF THE 32ND ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - VOLUME 1, 2010, Cape Town, South Africa. *Proceedings...* New York, NY, USA: ACM, 2010. (ICSE '10), p. 515–524. ISBN 978-1-60558-719-6. Disponível em: <http://doi.acm.org/10.1145/1806799.1806874>. Acesso em: 25 jul. 2016.

CORDEIRO, L.; FISCHER, B.; MARQUES-SILVA, J. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 38, n. 4, p. 957–974, 2012. ISSN 0098-5589. Acesso em: 25 jul. 2016.

COUSOT, P. Abstract interpretation based formal methods and future challenges. In: SPRINGER. *Informatics*. [S.l.], 2001. p. 138–156.

COUSOT, P.; COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. [S.l.], 1977. p. 238–252.

COUSOT, P. et al. The astreé analyzer. In: ____. *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 21–30. ISBN 978-3-540-31987-0. Disponível em: <http://dx.doi.org/10.1007/978-3-540-31987-0_3>.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 13, n. 4, p. 451–490, 1991.

DING, Z.; ZHANG, K.; HU, J. A rigorous approach towards test case generation. *Inf. Sci.*, Elsevier Science Inc., New York, NY, USA, v. 178, n. 21, p. 4057–4079, nov. 2008. ISSN 0020-0255. Disponível em: <http://dx.doi.org/10.1016/j.ins.2008.06.020>.

D'SILVA, V.; KROENING, D.; WEISSENBACHER, G. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 27, n. 7, p. 1165–1178, 2008.

ERNST, M. D. et al. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, Elsevier, v. 69, n. 1, p. 35–45, 2007.

GUPTA, A.; RYBALCHENKO, A. Invgen: An efficient invariant generator. In: COMPUTER AIDED VERIFICATION: 21ST INTERNATIONAL CONFERENCE, 2009, Grenoble, France. *Proceedings...* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. (CAV 2009), p. 634–640. ISBN 978-3-642-02658-4. Disponível em: <http://dx.doi.org/10.1007/978-3-642-02658-4_48>. Acesso em: 25 jul. 2016.

HEIZMANN, M. et al. Ultimate automizer with smtinterpol. In: ____. *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 641–643. ISBN 978-3-642-36742-7. Disponível em: <http://dx.doi.org/10.1007/978-3-642-36742-7_53>.

HENRY, J.; MONNIAUX, D.; MOY, M. Pagai: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 289, p. 15–25, 2012.

HODER, K.; KOVÁCS, L.; VORONKOV, A. Case studies on invariant generation using a saturation theorem prover. In: ADVANCES IN ARTIFICIAL INTELLIGENCE: 10TH MEXICAN INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 2011, Puebla, Mexico. *Proceedings...* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. (MICAI 2011), p. 1–15. ISBN 978-3-642-25324-9. Disponível em: <http://dx.doi.org/10.1007/978-3-642-25324-9_1>. Acesso em: 25 jul. 2016.

INVERSO, O. et al. Lazy-cseq: a lazy sequentialization tool for c. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2014. p. 398–401.

KHURSHID, S.; PĂSĂREANU, C. S.; VISSER, W. Generalized symbolic execution for model checking and testing. In: ____. *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 553–568. ISBN 978-3-540-36577-8. Disponível em: <http://dx.doi.org/10.1007/3-540-36577-X_40>.

KIM, J. et al. Static dalvik bytecode optimization for android applications. *ETRI Journal*, v. 37, n. 5, p. 1001–1011, Oct 2015. ISSN 1225-6463.

KOTOUN, M. et al. Optimized predatorhp and the sv-comp heap and memory safety benchmark. In: ____. *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. p. 942–945. ISBN 978-3-662-49674-9. Disponível em: <http://dx.doi.org/10.1007/978-3-662-49674-9_66>.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. Washington, DC, USA: IEEE Computer Society, 2004. (CGO '04), p. 75–. ISBN 0-7695-2102-9. Disponível em: <http://dl.acm.org/citation.cfm?id=977395.977673>.

LEINO, K. R. M. *This is boogie 2*. [S.l.], 2008.

LLVM Foundation. *LLVM Users*. 2009. Disponível em: <http://llvm.org/Users.html>.

LLVM Foundation. *C++ at Google: Here Be Dragons*. 2011. Disponível em: <http://blog.llvm.org/2011/05/c-at-google-here-be-dragons.html>.

LLVM Foundation. *The LLVM Compiler Infrastructure*. 2017. Disponível em: <http://llvm.org>.

MERZ, F.; FALKE, S.; SINZ, C. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In: JOSHI, R.; MÜLLER, P.; PODELSKI, A. (Ed.). *Proceedings...* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 146–161. ISBN 978-3-642-27705-4. Disponível em: <http://dx.doi.org/10.1007/978-3-642-27705-4_12>. Acesso em: 25 jul. 2016.

Microsoft Corporation. *Is this property always true?* 2017. Disponível em: <http://rise4fun.com/Boogie/McCarthy-91>.

MORSE, J. et al. Handling unbounded loops with esbmc 1.20. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2013. p. 619–622.

MOURA, L.; DUTERTRE, B.; SHANKAR, N. A tutorial on satisfiability modulo theories. In: SPRINGER. *Computer Aided Verification*. [S.l.], 2007. p. 20–36.

NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2007. (PLDI '07), p. 89–100. ISBN 978-1-59593-633-2. Disponível em: <http://doi.acm.org/10.1145/1250734.1250746>.

NEUMANN, P. G. *Illustrative Risks to the Public in the Use of Computer Systems and Related Technology*. 2015. Disponível em: <http://www.csl.sri.com/users/neumann/illustrativerisks.html>.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 5th. ed. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0072496681.

RAMALHO, M. et al. Smt-based bounded model checking of c++ programs. In: IEEE. *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*. [S.l.], 2013. p. 147–156.

ROCHA, H.; BARRETO, R.; CORDEIRO, L. Memory management test-case generation of c programs using bounded model checking. In: ____. *Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*. Cham: Springer International Publishing, 2015. p. 251–267. ISBN 978-3-319-22969-0. Disponível em: <http://dx.doi.org/10.1007/978-3-319-22969-0_18>.

ROCHA, H. et al. Understanding programming bugs in ansi-c software using bounded model checking counter-examples. In: SPRINGER. *Integrated Formal Methods*. [S.l.], 2012. p. 128–142.

ROCHA, H. O. et al. Verificação de sistemas de software baseada em transformações de código usando bounded model checking. Universidade Federal do Amazonas, 2015. Disponível em: <http://tede.ufam.edu.br/handle/tede/4752>.

SCHWABER, K.; BEEDLE, M. *Agile software development with Scrum*. [S.l.]: Prentice Hall Upper Saddle River, 2002. v. 1.

WALLACE, D. R.; FUJII, R. U. Software verification and validation: an overview. *Ieee Software*, IEEE Computer Society, v. 6, n. 3, p. 10, 1989.

WANG, D. et al. C code verification based on the extended labeled transition system model. In: *D&P@MoDELS*. [S.l.: s.n.], 2016.

WEGMAN, M. N.; ZADECK, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 13, n. 2, p. 181–210, 1991.

WONG, W. E. et al. Recent catastrophic accidents: Investigating how software was responsible. In: SECURE SOFTWARE INTEGRATION AND RELIABILITY IMPROVEMENT , 2010 FOURTH INTERNATIONAL CONFERENCE ON, 2010. *Proceedings...* [S.l.], 2010. (SSIRI 2010), p. 14–22.