Verificação de propriedades de gerenciamento de memória de programas em C baseado em transformações de código

---

Rafael Sá Menezes

Orientador: Prof. Dr. Herbert Oliveira Rocha

02 de Agosto de 2017

Universidade Federal de Roraima

Departamento de Ciência da Computação

1. **Introdução**
2. Conceitos e Definições
3. Método Proposto
4. Trabalhos Correlatos
5. Resultados Experimentais
6. Conclusões e Trabalhos Futuros

Aplicações do *Software*

Sistemas Críticos

São sistemas onde diversas restrições (tempo de resposta e precisão dos dados) devem ser atendidas e mensuradas de acordo com os requisitos do usuários (ROCHA et al., 2015b)

Figura: Ariane-5 explodiu após 35 segundos por erro de *cast*(BAIER; KATOEN, 2008)

- Uma propriedade em tempo linear que especifica um comportamento admissível de um sistema (BAIER; KATOEN, 2008).
- Validar o comportamento

Caixa eletrônico

Uma pessoa só deve poder sacar de um caixa eletrônico caso ela se identifique corretamente.

- Com o aumento da complexidade de *softwares* e a diminuição dos prazos tornam mais complexa a verificação e correção deste *software* (D'SILVA et al., 2008).
- Em 2009 mais de 100 erros de vazamentos de memória foram reportados no navegador Firefox (CLAUSE; ORSO, 2010).

- Falhas no gerenciamento de memória impactam na performance não só da aplicação como de todo o sistema (CLAUSE; ORSO, 2010).
- Erros de gerenciamento são difíceis de serem identificados (CLAUSE; ORSO, 2010).
- *Softwares* de alto risco necessitam de um alto grau de confiabilidade (D'SILVA et al., 2008).

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?

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.

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 escritos 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

Metodologia de desenvolvimento do trabalho

1. Análise e avaliação de toda a teoria necessária para entender os métodos, técnicas e ferramentas aplicados à verificação e teste de *software* com foco em segurança de memória;
2. Desenvolvimento do método proposto;
3. Avaliação experimental.

1. A implementação e avaliação de um método para a verificação e teste de programas em C.
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.
3. Este trabalho visa contribuir com método Map2Check (ROCHA et al., 2015), adicionando suporte a execução simbólica e geração de propriedades de segurança de memória.

1. Introdução
2. **Conceitos e Definições**
3. Método Proposto
4. Trabalhos Correlatos
5. Resultados Experimentais
6. Conclusões e Trabalhos Futuros

Teste de *Software*

Através de execução, garante que o *software* funcione para um conjunto de casos (DING et al., 2008).

Verificação de *Software*

Através de modelos matemáticos, prova a corretude de um *software* (CLARKE et al., 2009).

```
1  int main() {
2    int i, j;
3    int length = nondet_int();
4    if (length < 1) length = 1;
5    int *arr = alloca(length);
6    if (!arr) return 0;
7    int *a = arr;
8    while (*a != *(arr + length -↩
         1)) {
9      *a += *(arr + length - 1);
10     a++;
11   }
12   return 0;
13 }
```

- Programa adaptado do *benchmark* da SV-COMP'17.
- `alloca` é uma função para fazer uma alocação na memória *heap*.
- `nondet_int` é uma função que retorna um inteiro não determinístico.

```
1  int main() {
2    int i, j;
3    int length = nondet_int();
4    if (length < 1) length = 1;
5    int *arr = alloca(length);
6    if (!arr) return 0;
7    int *a = arr;
8    while (*a != *(arr + length -↩
         1)) {
9      *a += *(arr + length - 1);
10     a++;
11   }
12   return 0;
13 }
```

- Numa arquitetura de 32-bits o limite teórico de um *array* é de: $2^{31} - 1$ (que corresponde a 2147483647);
- A quantidade alocada de *bytes* deve ser um múltiplo do tamanho de um inteiro.

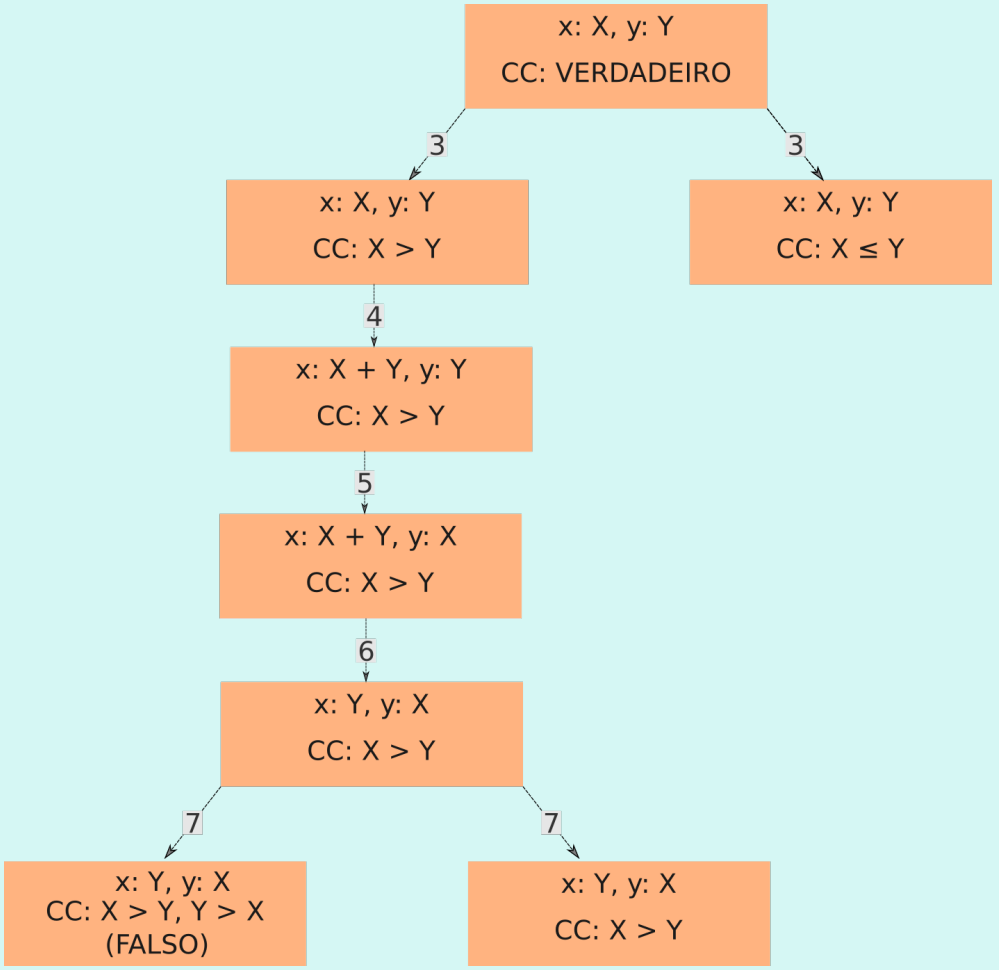
```
1  int main() {
2    int i, j;
3    int length = nondet_int();
4    if (length < 1 || length >= 2147483647 / sizeof(↩
         int)) length = 1;
5    int *arr = alloca(length*sizeof(int));
6    if (!arr) return 0;
7    int *a = arr;
8    while (*a != *(arr + length - 1)) {
9      *a += *(arr + length - 1);
10     a++;
11   }
12   return 0;
13 }
```

- Geração de valores utilizando valores simbólicos.

Exemplo

```
1:  int x
2:  int y
3:  se x > y então
4:      x ← x + y
5:      y ← x − y
6:      x ← y − x
7:      se y < x então
8:          error()
9:      fim se
10: fim se
```

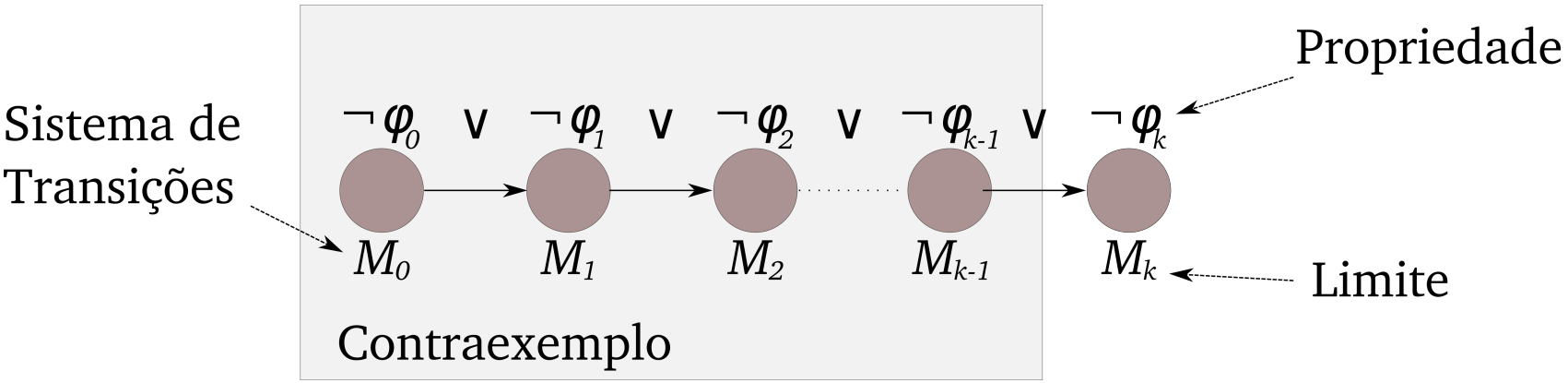
Árvore de Execução



- NP-Completo (CORMEM, 2009);
- Entrada é uma expressão lógica.

- Técnica de verificação que explora todos os estados possíveis do sistema (BAIER;KATOEN, 2008).
- O *model checking* pode demonstrar que um dado modelo do sistema satisfaz uma determinada propriedade (BAIER;KATOEN, 2008).
- O *model checker* gera contra-exemplos se um estado violar uma propriedade (BAIER;KATOEN, 2008).

- A ideia básica do BMC é verificar (a negação de) uma dada propriedade em uma dada profundidade do sistema (CORDEIRO et al., 2012).
- Sistema de transição M desenrolado k vezes (CORDEIRO et al., 2012).



- Desalocação inválida.

```c
int main() {
  int a;
  free(a);
  return 0;
}
```

- Vazamento de memória.

```c
int main() {
  int* a = malloc(sizeof(int)*32);
  return 0;
}
```

- Desreferência de ponteiro.

```c
int main() {
  int a[5];
  a[5] = 7;
  return 0;
}
```

Remoção de código morto

Remove instruções que computam valores não utilizados (AHO et al., 1986)

```
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 }
```

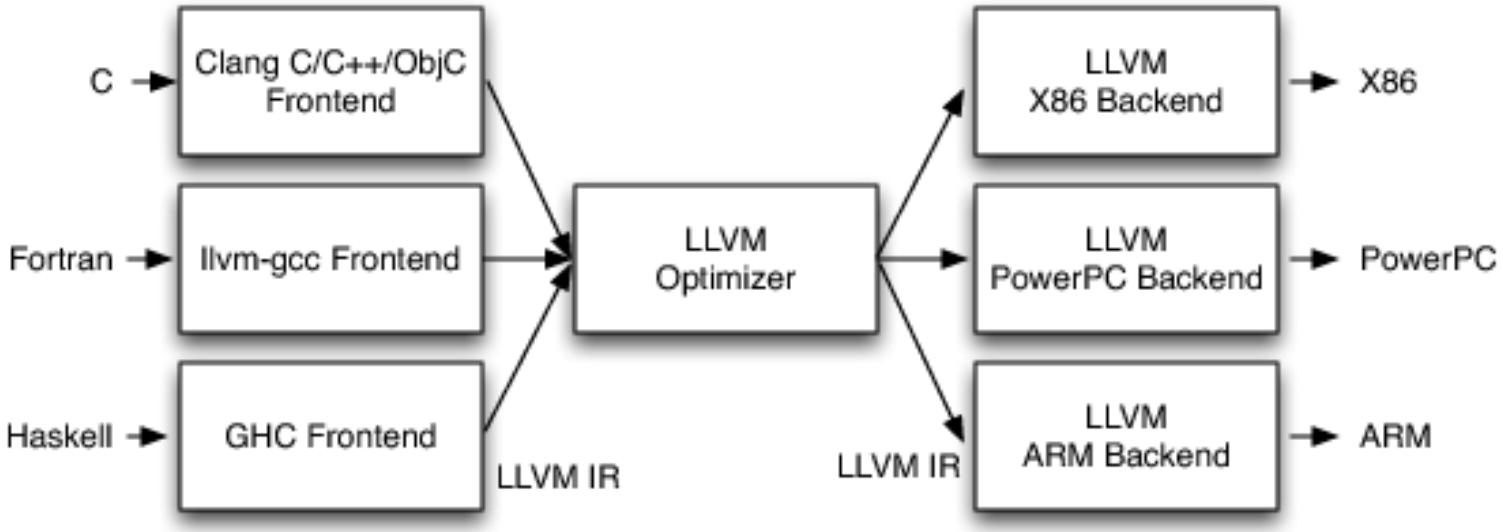
```
1 int global;
2
3 int foo() {
4   global = 10;
5   return 2;
6 }
```

Propagação de constantes

Consiste em realizar operações matemáticas ou lógicas durante o processo de compilação ao invés de durante a execução (AHO et al., 1986)

```
1 int a = 3 * 7;
2 int b = a + 2;
```

```
1 int a = 21;
2 int b = 23;
```



Características

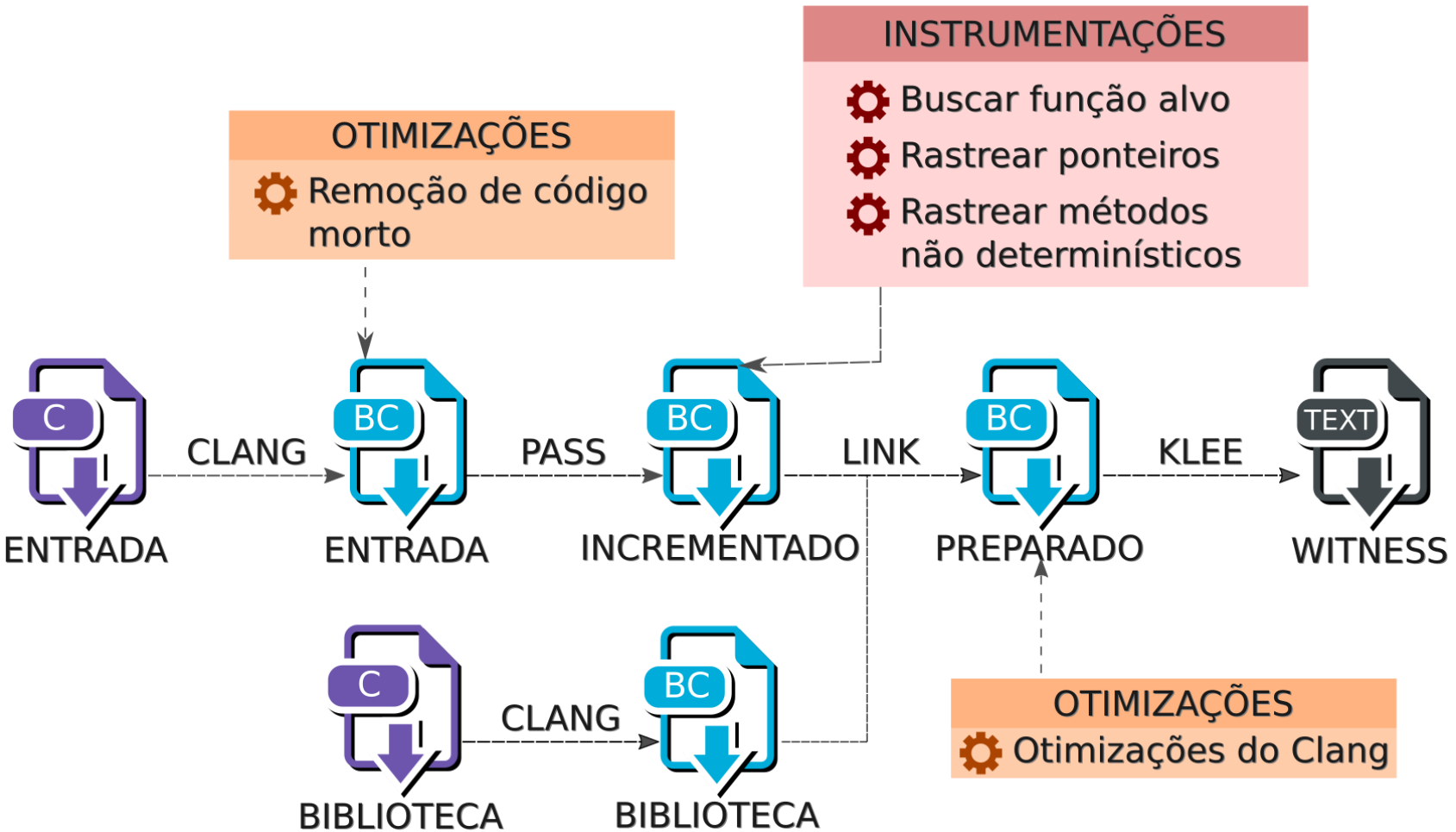
- *Framework* de compilação
- Código intermediário em SSA
- Simples de criar análises e transformações sobre programas

http://llvm.org/

```
1 int main() {
2   int a = 2;
3   int b = a ↩
        + 7;
4   a = a * 3 ↩
        + b;
5   return 0;
6 }
```

```
1  define i32 @main() #0 {
2    %1 = alloca i32, align 4
3    %a = alloca i32, align 4
4    %b = alloca i32, align 4
5    store i32 0, i32* %1, align 4
6    store i32 2, i32* %a, align 4
7    %2 = load i32, i32* %a, align 4
8    %3 = add nsw i32 %2, 7
9    store i32 %3, i32* %b, align 4
10   %4 = load i32, i32* %a, align 4
11   %5 = mul nsw i32 %4, 3
12   %6 = load i32, i32* %b, align 4
13   %7 = add nsw i32 %5, %6
14   store i32 %7, i32* %a, align 4
15   ret i32 0
16 }
```

1. Introdução
2. Conceitos e Definições
3. **Método Proposto**
4. Trabalhos Correlatos
5. Resultados Experimentais
6. Conclusões e Trabalhos Futuros



[1]

[1] https://github.com/hbgit/Map2Check/tree/map2checkllvm

```
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
```

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

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

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

HEAP_LOG e MALLOC_LOG

Representam o estado atual da memória *heap* e da memória dinâmica, sendo utilizados para validar propriedades de segurança de memória.

KLEE_LOG e LIST_LOG

Estruturas que contém os valores gerados pela execução simbólica e o histórico de operações que ocorreram nos ponteiros, essas estruturas utilizadas para geração do contraexemplo.

```c
int main(){
  int *A = malloc(10);
  free(A);
  return 0;
}
```

```c
int main(){
  void *temp;
  map2check_alloca(&temp, 4);
  int *A;
  void *temp_var_0 = malloc(10);
  map2check_malloc(temp_var_0, 10);
  map2check_alloca(&temp_var_0, 4);
  map2check_deref(&temp, sizeof(temp_var_0));
  temp = temp_var_0;
  ...
  map2check_free(A, "main");
  free(A);
  map2check_exit();
  return 0;
}
```

```
1  int main(){
2    int *A = malloc(10);
3    int *B = malloc(10);
4    int x = non_det_int();
5    int y = x * 7 - 15;
6
7    if (y > 14) {
8      B = A;
9    }
10
11   free(A);
12   free(B);
13   return 0;
14 }
```

Violação de Propriedade

Com a execução é possível checar quais casos falham. Por exemplo, para $x = 7$ a propriedade de segurança de desalocação de memória é violada.

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

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

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

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

```
1  int main() {
2    int i, j;
3    int length = ↩
         nondet_int();
4    if (length < 1) ↩
         length = 1;
5    int *arr = alloca(↩
         length);
6    if (!arr) return 0;
7    int *a = arr;
8    while (*a != *(arr + ↩
         length - 1)) {
9      ...
10   }
11   return 0;
12 }
```

```
State 2: file /home/rafaelsa/add_last_u
----------------------------------------
>>Memory list log

  Line content   :   int *a = arr;
  Address        : 0x55d40d95f2f0
  PointsTo       : 0x55d40d8e5050
  Is Free        : FALSE
  Is Dynamic     : FALSE
  Var Name       : a
  Line Number    : 7
  Function Scope : main

----------------------------------------
Violated property:
        file map2check_property line 8
        FALSE-DEREF: Reference to point

VERIFICATION FAILED
```

1. Introdução
2. Conceitos e Definições
3. Método Proposto
4. **Trabalhos Correlatos**
5. Resultados Experimentais
6. Conclusões e Trabalhos Futuros

- **Symbiotic 3**, LLVM para instrumentação de código (CHALUPA et al., 2016).
- **LEAKPOINT**, rastreio de métodos de alocação e desalocação em C/C++(CLAUSE; ORSO, 2010).

- **SMACK**, LLVM para tradução do código C para Boogie (WANG et al., 2016).
- **Map2Check v6**, união entre *model checking* com teste unitário (ROCHA et al., 2015a).

1. Introdução
2. Conceitos e Definições
3. Método Proposto
4. Trabalhos Correlatos
5. **Resultados Experimentais**
6. Conclusões e Trabalhos Futuros

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?

- 328 programas da categoria *MemorySafety* do *benchmark* da *Competition on Software Verification* (SV-COMP).
- Utilização da ferramenta *Benchexec* com *timeout* de 15 minutos.
- Linux Ubuntu 16.10 com 8GB de RAM e processador AMD 2.9 GHz com 8 núcleos.

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) com: PredatorHP, Ultimate Automizer, CBMC, SMACK, Map2Check v6 e Symbiotic 4

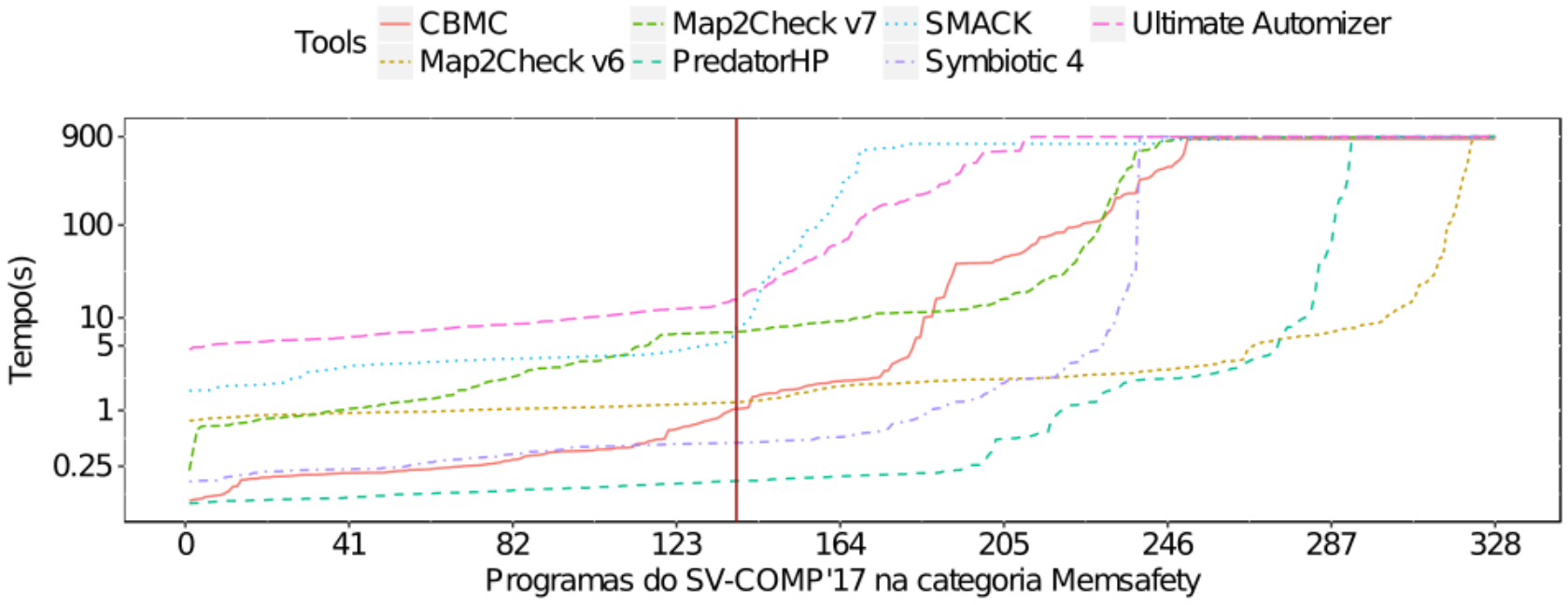
Ferramenta	Map2Check v7 LLVM	Ultimate Automizer	PredatorHP	Symbiotic 4	SMACK	Map2Check v6	CBMC
Verdadeiro Correto	106	94	103	104	142	138	66
Falso Correto	126	51	116	129	121	24	127
Verdadeiro Incorreto	0	0	0	0	1	88	0
Falso Incorreto	0	0	0	0	10	16	0
Desconhecidos	96	183	109	95	54	62	135
Total Corretos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

Regras

- +2 pontos por Verdadeiro Correto;
- +1 ponto por Falso Correto;
- -32 pontos por Verdadeiro Incorreto;
- -16 pontos por Falso Incorreto.

Pontuações

1. Map2Check v7: 338;
2. Symbiotic 4: 337;
3. PredatorHP: 322.


Tools
CBMC
Map2Check v7
SMACK
Ultimate Automizer
Map2Check v6
PredatorHP
Symbiotic 4
Tempo(s)
900
100
10
5
1
0.25
0
41
82
123
164
205
246
287
328
Programas do SV-COMP'17 na categoria Memsafety

1. Introdução
2. Conceitos e Definições
3. Método Proposto
4. Trabalhos Correlatos
5. Resultados Experimentais
6. **Conclusões e Trabalhos Futuros**

- Comparando a versão anterior do Map2Check, o novo método apresentou melhorias significativas sobre o *benchmark* da SV-COMP'17;
- 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;
- A nova versão do Map2Check não gerou nenhum falso positivo ou falso negativo, o que sugere que o método proposto é eficaz mesmo quando comparado a outras ferramentas.
- O Map2Check v7 está disponível em: `https://map2check.github.io/`.

- Suporte a programas com paralelismo;
- Geração de um *witness* de corretude;
- Geração de código fonte com assertivas suportadas por um *framework* de teste de unidade;
- Adoção de um *slicer* de código;
- Validação das *witness* geradas.

Obrigado e Dúvidas?