Detecção Automática de Overflow em Contratos Solidity: Uma Análise Detalhada do Pipeline do ESBMC

Matheus Junio
Universidade Federal de Viçosa (UFV) – Florestal, MG – Brasil
matheus.junio@ufv.br

17 de junho de 2025

Resumo

Este documento apresenta uma análise técnica do processo de detecção automática de vulnerabilidades de overflow em contratos Solidity utilizando o ESBMC (Efficient SMT-Based Bounded Model Checker). Através de uma abordagem empírica, exploramos o pipeline completo desde o parsing do contrato até a geração de contraexemplos, fundamentando nossa análise no manual oficial do ESBMC [1] e em evidências obtidas através de logs de debug. O estudo demonstra como o ESBMC implementa verificação formal através da fórmula $C \wedge \neg P$, onde C representa as constraints do programa e P as propriedades de segurança. Comparamos propriedades automáticas versus manuais, analisando diferenças na instrumentação, tempos de execução e geração de condições de verificação. Os resultados mostram que propriedades manuais são instrumentadas durante a geração do programa GOTO, enquanto propriedades automáticas são adicionadas em fase posterior de instrumentação, resultando em diferentes características de performance.

1 Introdução

A verificação formal de contratos inteligentes representa um desafio crítico na garantia de segurança de sistemas blockchain. Vulnerabilidades de overflow aritmético constituem uma das classes mais prevalentes de bugs em contratos Solidity, onde operações que excedem os limites dos tipos de dados podem resultar em comportamentos inesperados e falhas de segurança.

O ESBMC (Efficient SMT-Based Bounded Model Checker) é uma ferramenta de verificação formal que utiliza técnicas avançadas de model checking para detectar vulnerabilidades em código. Diferentemente de ferramentas de análise estática tradicionais, o ESBMC emprega verificação baseada em satisfibilidade para provar matematicamente a presença ou ausência de bugs através de técnicas de verificação formal [1].

Este trabalho apresenta uma análise técnica abrangente do processo interno de detecção de overflow no ESBMC, com foco particular na comparação entre propriedades automáticas e manuais. Baseamos nossa investigação tanto na documentação oficial quanto em evidências empíricas obtidas através da execução controlada do verificador.

2 Fundamentos Teóricos

2.1 A Fórmula: $C \wedge \neg P$

O coração da verificação formal no ESBMC é baseado na estratégia de **verificação por contradição**. Como explica o manual do ESBMC [1], o verificador utiliza a fórmula:

$$C \wedge \neg P$$

Onde:

- C: Representa todas as **constraints** (restrições) do programa
- P: Representa as **propriedades** de segurança que queremos verificar
- $\neg P$: A negação da propriedade (assumindo que ela pode ser violada)

A estratégia técnica desta abordagem baseia-se na lógica de que, para provar que uma propriedade é sempre verdadeira, busca-se um contraexemplo onde ela seja falsa. Se nenhum contraexemplo for encontrado, a propriedade está verificada.

2.2 Arquitetura Multi-Frontend e Conversão GOTO

Uma característica fundamental do ESBMC é sua arquitetura multi-frontend, como descrito no manual [1]. O ESBMC suporta múltiplas linguagens de programação (C, C++, Python, Solidity) através da conversão para uma representação intermediária unificada chamada programa GOTO.

Esta estratégia arquitetural oferece vantagens significativas:

- **Reutilização de código**: As fases de análise, instrumentação e verificação são implementadas uma única vez
- **Consistência**: Garante comportamento uniforme independentemente da linguagem fonte
- **Manutenibilidade**: Novos frontends podem ser adicionados sem modificar o núcleo de verificação

O programa GOTO consiste em instruções básicas como ASSIGN, ASSUME, ASSERT, GOTO, IF, facilitando a análise simbólica posterior. Como documentado no manual [1], esta representação preserva a semântica do programa original enquanto simplifica o processo de verificação.

2.3 Pipeline de Verificação

O ESBMC segue um pipeline bem definido, como documentado no manual [1]:

1. **Parsing**: Conversão do código fonte para AST (Abstract Syntax Tree)
2. **Conversão GOTO**: Transformação para linguagem intermediária
3. **Instrumentação**: Adição automática de verificações
4. **Execução Simbólica**: Geração de condições de verificação
5. **Encoding SMT**: Conversão para fórmulas matemáticas
6. **Solving**: Resolução através de solvers SMT

3 Caso de Estudo: Contrato com Overflow

Para demonstrar o funcionamento do ESBMC, utilizamos o seguinte contrato Solidity vulnerável:

Listing 1: Contrato IntegerOverflowAdd.sol

```
pragma solidity ^0.8.0;

contract IntegerOverflowAdd {
    uint public count = 1;

    function run(uint256 input) public {
        // Linha 17: Vulner vel a overflow
        count += input;
    }
}
```

Este contrato simples contém uma vulnerabilidade clássica: se `input` for muito grande, a operação `count += input` pode resultar em overflow.

3.1 Execução do ESBMC

Executamos o ESBMC com o comando:

```
./build/src/esbmc/esbmc --sol integer_overflow_add.sol \
  integer_overflow_add.solast --incremental-bmc --overflow-check
```

3.2 Análise dos Logs de Debug

Os logs de execução revelam informações cruciais sobre o funcionamento interno:

Listing 2: Log de execução do ESBMC

```
ESBMC version 7.9.0 64-bit x86_64 linux
Parsing integer_overflow_add.solast
Converting
Generating GOTO Program
GOTO program creation time: 0.502s
GOTO program processing time: 0.023s
Checking base case, k = 1
Starting Bounded Model Checking
Symex completed in: 0.024s (119 assignments)
Generated 21 VCC(s), 10 remaining after simplification
```

4 Detecção Automática de Propriedades

4.1 Instrumentação Automática

Uma das características mais impressionantes do ESBMC é sua capacidade de **gerar automaticamente** propriedades de verificação. Como descrito no manual [1], o arquivo `goto_check.cpp` é responsável por esta instrumentação.

Quando encontra uma operação aritmética como `count += input`, o ESBMC automaticamente adiciona verificações equivalentes a:

```
// Verifica  o autom tica gerada pelo ESBMC
assert(!overflow("+", count, input));
```

4.2 Tipos de Verificação de Overflow

O ESBMC implementa três tipos principais de verificação de overflow, conforme documentado no código fonte:

1. **overflow2tc()**: Para operações normais (+, -, *, /)
2. **overflow_neg2tc()**: Para operações de negação (-x)
3. **overflow_cast2tc()**: Para conversões de tipo

4.3 Função add_guarded_claim()

A função `add_guarded_claim()` no arquivo `goto_check.cpp` é responsável por adicionar as verificações ao programa de forma condicional:

Listing 3: Função add_guarded_claim simplificada

```
void goto_checkt::add_guarded_claim(
  const expr2tc &expr,            // Propriedade P
  const std::string &comment,     // Descri  o
  const std::string &property,    // Tipo (overflow, bounds, etc.)
  const locationt &location,      // Localiza  o no c digo
  const guardt &guard)           // Condi  o G: "s  verifica SE..."
{
  // Combina guard com propriedade (G     P)
  expr2tc new_expr = guard.is_true() ?
    e : implies2tc(guard.as_expr(), e);

  // Adiciona instru  o ASSERT no programa GOTO
  goto_programt::targett t = new_code.add_instruction(ASSERT);
  t->guard = new_expr;
  t->location = location;
}
```

5 Execução Simbólica e SSA

5.1 Static Single Assignment (SSA)

O ESBMC converte o programa para forma SSA, onde cada variável é atribuída apenas uma vez. Como explicado no manual [1], isso facilita a análise simbólica:

Listing 4: Exemplo de conversão SSA

```
// C digo original:
count = 1;
count = count + input;

// Forma SSA:
count_1 = 1;
count_2 = count_1 + input;
```

5.2 Geração de Condições de Verificação

Durante a execução simbólica, o ESBMC gera **Verification Conditions (VCs)**, que representam fórmulas lógicas que devem ser verificadas pelo solver SMT. Como documentado no manual [1], as VCs constituem a interface entre a execução simbólica e o processo de verificação formal.

5.2.1 Natureza das Condições de Verificação

As VCs são expressões booleanas que codificam:

- **Path conditions**: Condições que definem caminhos de execução específicos
- **Safety properties**: Propriedades que não devem ser violadas
- **Assumptions**: Restrições sobre valores de entrada e estado

5.2.2 Processo de Geração

No caso de estudo analisado:

- **Geração inicial**: 21 VCs para verificação automática, 12 para verificação manual
- **Após simplificação**: 10 VCs mantidas (automática), 1 VC mantida (manual)
- **Otimização**: O processo de slicing remove VCs irrelevantes para a propriedade alvo

A diferença no número de VCs reflete a abrangência da instrumentação: verificação automática monitora múltiplas operações, enquanto verificação manual focaliza propriedades específicas.

5.2.3 Conversão para SSA

As VCs são geradas a partir da forma SSA (Static Single Assignment) do programa, onde cada variável é atribuída exatamente uma vez. Esta representação facilita a análise simbólica e a geração de fórmulas SMT precisas.

6 Combinação de Constraints e Propriedades

6.1 Implementação em symex_target_equation.cpp

O arquivo `symex_target_equation.cpp` é onde acontece a "mágica"da combinação $C \wedge \neg P$, como documentado no manual [1]:

Listing 5: Função convert_internal_step

```
void symex_target_equationt::convert_internal_step(
  smt_convt &smt_conv,
  smt_astt &assumpt_ast,        // Constraints C
  smt_convt::ast_vec &assertions, // Propriedades  P
  SSA_stept &step)
{
  if (step.is_assert())  // Se    uma propriedade
  {
    // Combina C com P via implica  o
    step.cond_ast = smt_conv.imply_ast(assumpt_ast, step.cond_ast);

    // NEGA a propriedade ( P )
    assertions.push_back(smt_conv.invert_ast(step.cond_ast));
  }
  else if (step.is_assume())  // Se    uma constraint
  {
    // Acumula constraints via AND
    assumpt_ast = smt_conv.mk_and(assumpt_ast, step.cond_ast);
  }
}
```

6.2 Por que Negar a Propriedade?

A negação da propriedade ($\neg P$) é fundamental para a verificação por contradição. Se o solver encontrar uma solução para $C \wedge \neg P$, significa que existe um contraexemplo onde a propriedade falha.

7 Resultados e Contraexemplo

7.1 Detecção do Bug

O ESBMC detectou com sucesso a vulnerabilidade de overflow, conforme mostrado nos logs:

Listing 6: Saída do ESBMC - Bug detectado

```
[Counterexample]

State 6 file integer_overflow_add.sol line 17 function run thread 0
Violated property:
  file integer_overflow_add.sol line 17 function run
  arithmetic overflow on add
  !overflow("+", this->count, input)

VERIFICATION FAILED
Bug found (k = 1)
```

8 Matemática por Trás das Verificações

8.1 Limites de Tipos de Dados

Para inteiros sem sinal (uint256 no Solidity), o ESBMC implementa verificações baseadas nos limites matemáticos:

$$MAX\_UINT256 = 2^{256} - 1$$

A verificação de overflow para adição é:

$$\text{overflow}(a + b) \iff (a + b) > MAX\_UINT256$$

8.2 Verificação para Inteiros com Sinal

Para tipos com sinal (int8, int16, etc.), a verificação considera tanto overflow positivo quanto negativo:

$$MAX\_INT_n = 2^{n-1} - 1$$
$$MIN\_INT_n = -2^{n-1}$$

Para operação de adição `a + b` com sinal:

$$\text{overflow\_positivo} \iff (a > 0) \wedge (b > 0) \wedge (a > MAX\_INT_n - b) \quad (1)$$
$$\text{overflow\_negativo} \iff (a < 0) \wedge (b < 0) \wedge (a < MIN\_INT_n - b) \quad (2)$$

8.3 Verificação para Operações de Negação

A negação unária (`-x`) apresenta casos especiais críticos:

8.3.1 Inteiros com Sinal

O caso problemático ocorre com o menor valor representável:

$$\text{overflow}(-x) \iff x = MIN\_INT_n$$

Exemplo: para `int8`, o valor -128 não pode ser negado pois +128 excede o limite de 127.

8.3.2 Inteiros sem Sinal

Para tipos unsigned, qualquer negação de valor não-zero resulta em wrap-around:

$$\text{overflow}(-x) \iff x \neq 0$$

8.4 Verificação para Multiplicação

A multiplicação requer análise cuidadosa devido ao crescimento quadrático:

Para tipos unsigned:

$$\text{overflow}(a \times b) \iff a \neq 0 \wedge b > \frac{MAX\_UINT_n}{a}$$

Para tipos com sinal, a verificação considera quatro cenários baseados nos sinais dos operandos.

9 Solvers SMT e Encoding

9.1 Z3 Solver

Nos logs, vemos que o ESBMC utilizou o solver Z3 v4.8.12:

```
No solver specified; defaulting to z3
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Solving with solver Z3 v4.8.12
Runtime decision procedure: 0.079s
```

O Z3 resolve as fórmulas SMT em apenas 0.079 segundos, demonstrando a eficiência da abordagem.

9.2 Encoding de Bit-Vectors

O ESBMC utiliza aritmética de bit-vectors para representar inteiros, permitindo verificação precisa de overflow conforme os tipos de dados do Solidity.

10 Comparação: Propriedades Manuais vs Automáticas

10.1 Abordagem Manual

Na abordagem manual, o desenvolvedor precisa especificar explicitamente as propriedades:

```
function run(uint256 input) public {
    // Propriedade manual - verifica  o ANTES da opera  o
    assert(count <= type(uint256).max - input);
    count += input;
}
```

10.2 Abordagem Automática

Com `--overflow-check`, o ESBMC adiciona automaticamente todas as verificações necessárias, como demonstrado no nosso caso de estudo.

Vantagens da abordagem automática:

- Não requer conhecimento específico de segurança
- Cobertura completa de operações aritméticas
- Reduz erro humano
- Facilita manutenção do código

11 Performance e Otimizações

11.1 Análise dos Tempos

Os logs revelam os tempos de cada etapa do processo:

- **Parsing**: Parsing do AST Solidity
- **GOTO creation**: 0.502s
- **GOTO processing**: 0.023s
- **Symbolic execution**: 0.024s (119 assignments)
- **Slicing**: 0.000s (removeu 90 assignments)
- **Encoding**: 0.002s
- **Solving**: 0.079s

11.2 Otimização através de Slicing

O ESBMC otimiza a verificação removendo assignments irrelevantes (90 de 119), mantendo apenas os essenciais para verificação.

12 Limitações e Trabalhos Futuros

12.1 Bounded Model Checking

O ESBMC utiliza Bounded Model Checking (BMC), verificando propriedades até um limite k. No nosso caso, $k = 1$ foi suficiente para encontrar o bug.

12.2 Loops e Unwinding

Os logs mostram loops não desenrolados:

```
Not unwinding loop 47 iteration 1
Not unwinding loop 48 iteration 1
```

Estes loops fazem parte da biblioteca C e não afetam a verificação do contrato Solidity.

13 Conclusões

Este estudo apresentou uma análise técnica abrangente dos mecanismos de detecção de overflow no ESBMC, com foco particular na comparação entre propriedades automáticas e manuais. As principais contribuições incluem:

13.1 Descobertas sobre Instrumentação

1. **Timing de instrumentação**: Propriedades manuais são incorporadas durante a geração GOTO (0.547s), enquanto automáticas são instrumentadas posteriormente (0.491s de geração GOTO + 0.024s de processamento)
2. **Granularidade**: Verificação automática gera 21 VCs iniciais vs 12 para verificação manual, demonstrando cobertura mais ampla porém menos focada
3. **Eficiência de resolução**: Propriedades manuais resultam em 76.7% menos tempo de SMT solving (0.017s vs 0.073s)

13.2 Implicações para Verificação Formal

A análise empírica revela trade-offs fundamentais:

- **Cobertura vs Eficiência**: Verificação automática oferece cobertura completa mas com custo computacional maior
- **Especialização vs Generalização**: Propriedades manuais permitem otimizações específicas do domínio
- **Usabilidade vs Performance**: Flags automáticas reduzem barreira de entrada mas aumentam complexidade de verificação

13.3 Validação da Fórmula $C \wedge \neg P$

Os logs de debug confirmaram a implementação correta da verificação por contradição, onde o ESBMC:

1. Combina constraints do programa (C) via conjunção
2. Nega propriedades de segurança ($\neg P$)
3. Busca contraexemplos satisfazendo $C \wedge \neg P$
4. Conclui sobre a validade das propriedades baseado na satisfibilidade

O estudo comprova que o ESBMC implementa verificação formal rigorosa conforme documentado no manual [1], oferecendo tanto verificação automática para usuários iniciantes quanto controle fino para especialistas em verificação formal.

14 Trabalhos Relacionados

O ESBMC deriva significativamente do projeto CBMC [1], adaptando suas técnicas para verificação de contratos Solidity. A abordagem de instrumentação automática distingue o ESBMC de outras ferramentas que dependem de especificações manuais.

A arquitetura multi-frontend do ESBMC, documentada no manual [1], permite suporte uniforme a múltiplas linguagens através da conversão para programa GOTO, uma estratégia que demonstra eficácia tanto em termos de engenharia de software quanto de performance de verificação.

15 Análise Comparativa de Performance

15.1 Metodologia Experimental

Para investigar as diferenças de performance entre abordagens manual e automática, executamos o mesmo contrato vulnerável sob duas configurações:

1. **Configuração Automática**: Contrato original com flag `--overflow-check`
2. **Configuração Manual**: Contrato modificado com `assert()` explícito, sem flags automáticas

15.2 Resultados Empíricos

15.2.1 Propriedades Automáticas

Listing 7: Logs - Verificação Automática

```
GOTO program creation time: 0.491s
GOTO program processing time: 0.024s
Symex completed in: 0.023s (119 assignments)
Generated 21 VCC(s), 10 remaining after simplification (29 assignments)
Runtime decision procedure: 0.073s
```

15.2.2 Propriedades Manuais

Listing 8: Logs - Verificação Manual

```
GOTO program creation time: 0.547s
GOTO program processing time: 0.015s
Symex completed in: 0.022s (113 assignments)
Generated 12 VCC(s), 1 remaining after simplification (9 assignments)
Runtime decision procedure: 0.017s
```

15.3 Análise dos Resultados

Os dados empíricos revelam diferenças significativas entre as abordagens:

15.3.1 Tempo de Geração GOTO

- **Automática**: 0.491s (mais rápido)
- **Manual**: 0.547s (+11.4%)

A diferença confirma que propriedades manuais são processadas durante a geração GOTO, aumentando a complexidade desta fase.

15.3.2 Condições de Verificação

- **Automática**: 21 VCs iniciais → 10 após simplificação
- **Manual**: 12 VCs iniciais → 1 após simplificação

A abordagem automática gera mais VCs pois instrumenta **todas** as operações aritméticas, enquanto a manual foca apenas na propriedade específica.

15.3.3 Tempo de Resolução SMT

- **Automática**: 0.073s
- **Manual**: 0.017s (-76.7%)

A diferença substancial no `Runtime decision procedure` demonstra que propriedades focalizadas resultam em fórmulas SMT mais simples e resolução mais eficiente.

15.3.4 Número de Assignments

- **Automática**: 119 assignments → 29 pós-slicing
- **Manual**: 113 assignments → 9 pós-slicing

A instrumentação automática gera mais assignments devido às múltiplas verificações de overflow inseridas.

Referências

[1] Jeremy Morse. *A manual on some internals of ESBMC*. Manual interno do ESBMC. Disponível no código fonte em docs/manual.tex.

[2] ESBMC Team. *ESBMC-Solidity: An SMT-Based Model Checker for Solidity Smart Contracts*. Artigo científico sobre ESBMC-Solidity. `https://arxiv.org/pdf/2111.13117`

[3] Ethereum Foundation. *Solidity Documentation*. `https://docs.soliditylang.org/`

[4] Daniel Kroening, Edmund Clarke. *Computing Science Technical Report: Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking*. Carnegie Mellon University, 2003.

[5] Leonardo de Moura, Nikolaj Bjørner. *Z3: An Efficient SMT Solver*. Microsoft Research, 2008.