Verificação de Contratos Solidity com o ESBMC 7.9

Matheus Júnio da Silva

13/05/2025

ESBMC 7.9 – Melhoria no suporte a contratos Solidity

A versão 7.9 do ESBMC traz várias melhorias específicas para verificação de contratos Solidity[1]. Entre os principais avanços estão:

- **Verificação cross-contract limitada:** introdução de um algoritmo de bounded cross-contract verification" que permite modelar chamadas externas de forma precisa dentro de um limite pré-definido (opção `--bound`). Isso melhora a análise de interações entre múltiplos contratos.
- **Suporte a chamadas com opções e unidades:** passa a dar suporte a palavras-chave de unidades (wei", gwei", ether", year", second" etc.) e inspeciona automaticamente verificações de *balance* insuficiente antes de transferências, detectando potenciais falhas ao enviar fundos sem verificar saldo.
- **Cobertura de código Solidity:** implementa métricas de *code coverage* para contratos Solidity, permitindo medir estatisticamente quais trechos foram verificados (instrumentação de cobertura mais fina).
- **Mesclagem de AST e multi-contratos:** passou a fundir ASTs de vários arquivos Solidity e corrigiu bugs de ordenação de funções, melhorando a verificação de projetos com múltiplos contratos.
- **Modelagem aprimorada:** refinou a modelagem de endereços e garantiu criação única de arquivos temporários para evitar condições de corrida durante a análise.

Vulnerabilidades detectáveis em contratos Solidity

O ESBMC-Solidity apresenta cobertura relevante sobre algumas das principais vulnerabilidades descritas no Top 10 da OWASP.

Especificamente, ele detecta com precisão a vulnerabilidade de Integer Overflow and Underflow (SWC-101), que corresponde diretamente à categoria SC08:2025 - Integer Overflow and Underflow, considerada uma das falhas críticas mais comuns em contratos inteligentes.

[1] `https://github.com/esbmc/esbmc/releases/tag/v7.9`

A ferramenta também identifica o uso inseguro de tx.origin (SWC-115), que pode ser explorado por contratos maliciosos para burlar verificações de permissão. Esse padrão específico pode ser associado à categoria SC01:2025 - Access Control Vulnerabilities da OWASP, embora seja importante destacar que o ESBMC não cobre todas as formas de falhas de controle de acesso, apenas aquelas diretamente relacionadas ao uso indevido de tx.origin.

Além disso, o ESBMC é capaz de detectar erros de lógica como asserts falsos e divisões por zero, que podem ser caracterizados como pertencentes à categoria SC03:2025 - Logic Errors. No entanto, essa associação não é completamente precisa e deve ser interpretada com cautela.

Por fim, outras vulnerabilidades, como acesso fora dos limites em arrays (SWC-110) e transferência de Ether sem verificação de saldo, também podem ser consideradas falhas de lógica ou de validação de entrada, aproximando-se das categorias SC03 e SC04:2025 - Lack of Input Validation.

Dessa forma, ainda que não cubra todas as vulnerabilidades da OWASP, o ESBMC-Solidity demonstra capacidade de identificar várias classes críticas, com destaque para SC08 e, de forma mais parcial, SC01 e SC03.

ESBMC-Solidity consegue detectar as principais classes de vulnerabilidades listadas no SWC (Smart Contract Weakness Classification)[2]. Estudos com *benchmarks* de contratos vulneráveis mostram que ESBMC identifica com sucesso vulnerabilidade como, por exemplo:

- **Overflow/Underflow de inteiros (SWC-101):** aritmética inteira que ultrapassa o limite do tipo causa reversão. Exemplo: um `uint8 x = 250; x += 10;` desbordaria para 4. ESBMC verifica tais operações aritméticas e gera contraprovas em caso de overflow [3].

- **Acesso fora de limites em arrays (SWC-110):** escrita ou leitura de índice inválido em array estático ou dinâmico. Por exemplo, um laço que atribui valor em `a[2]` num array `uint8[2] a;` é capturado como violação de *array bounds* [4] [fonte].

- **Uso inseguro de `tx.origin` (SWC-115):** contratos que usam `if (tx.origin == owner)` podem ser enganados por contratos maliciosos. ESBMC detecta condicionais que dependem de `tx.origin` ao invés de `msg.sender` e sinaliza esse padrão como problema de controle de acesso[5].

- **Verificação de saldo insuficiente:** se um contrato realiza transferência sem checar seu próprio saldo, ESBMC pode simular a chamada e disparar a assertiva de falha de saldo. A versão 7.9 detecta erros em transferências de Ether acima do que o contrato possui.

- **Outros erros de segurança comuns:** por meio do modelo simbólico, ESBMC pode encontrar `asserts` falsos, divisão por zero, índices inválidos, uso indevido de variáveis

[2] `https://swcregistry.io`
[3] `https://ssvlab.github.io/lucasccordeiro/papers/icse2022.pdf`
[4] `https://ssvlab.github.io/lucasccordeiro/papers/icse2022.pdf`
[5] `https://ssvlab.github.io/lucasccordeiro/papers/icse2022.pdf`

não inicializadas etc., que representam vulnerabilidades ou falhas de lógica típicas em contratos smart[6]..

Em uma avaliação comparativa publicada, ESBMC-Solidity detectou *todas* as vulnerabilidades dos casos de teste e foi a ferramenta mais rápida, sempre fornecendo contraprova completa para reproduzir o bug[7]. Em particular, Song et al. (2022) mostraram que ESBMC encontrou bugs em todos os testes confirmados (mesmo casos que outras ferramentas falharam), evidenciando sua eficácia e precisão no contexto Solidity.

Limitações conhecidas na verificação de Solidity

Apesar dos avanços, ESBMC-Solidity ainda não cobre toda a complexidade da linguagem. São reportadas limitações como:

- **Herança e polimorfismo:** o front-end Solidity traduz contratos para uma IR simplificada, mas não suporta plenamente herança múltipla ou sobrescrita complexa de funções. Isso faz com que contratos que usam extensivamente `contract A is B` possam falhar na verificação.

- `mapping` **e tipos avançados:** até a versão 7.9, não há suporte completo para o tipo `mapping`; questões envolvendo mapeamentos (hash tables do Solidity) ainda estão em desenvolvimento[8]. Outros recursos faltantes são estruturas aninhadas complexas, bibliotecas externas e *fallback functions* complexas.

- **Exemplos não analisáveis:** há registros de que ESBMC-Solidity não conseguiu analisar certos exemplos simples presentes em sua documentação oficial, indicando casos em que a análise front-end se rompe com a sintaxe Solidity esperada.

- **Modelagem de ambiente:** como ESBMC traduz Solidity para C intermediatário, discrepâncias semânticas podem ocorrer (por exemplo, na modelagem do ambiente Ethereum real). Além disso, o esquema de *bounded model checking* implica que casos de chamadas recursivas profundas ou loops muito longos podem não ser totalmente explorados devido a limites de *unroll*.

Avaliações recentes e benchmarks (desde 2022)

A literatura recente tem avaliado ESBMC em comparativos de ferramentas de análise de contratos. Song et al. (ICSE 2022) apresentaram um *benchmark* com contratos vulneráveis e mostraram que ESBMC-Solidity detectou todos os bugs de forma mais rápida que outras ferramentas populares (Mythril, Slither etc.)[9].

[6] `https://ssvlab.github.io/esbmc/documentation.html`
[7] `https://doi.org/10.1145/3510003.3510077`
[8] `https://github.com/esbmc/esbmc/issues/1769`
[9] `https://doi.org/10.1145/3510003.3510077`

Em estudos de 2023, grandes conjuntos de testes foram usados para comparar múltiplas ferramentas de segurança Ethereum; ESBMC costuma aparecer nesses levantamentos como uma das abordagens formais avaliadas (por exemplo, Wei et al. 2023 incluiu ESBMC entre 13 ferramentas comparadas)[10]. Em geral, esses benchmarks indicam que ESBMC apresenta alta acurácia na detecção de bugs de contratos e costuma ter desempenho competitivo, embora ainda precise superar limitações de linguagem para se igualar completamente aos *linters* especializados.

TC	SmartCheck		Slither		Oyente		Mythril		ESBMC-Solidity	
	Found	CE	Found	CE	Found	CE	Found	CE	Found	CE
TC1	No	-	No	-	No	-	Yes	No	Yes	Yes
TC2	No	-	No	-	No	-	Yes	No	Yes	Yes
TC3	No	-	No	-	No	-	Yes	No	Yes	Yes
TC4	No	-	No	-	No	-	Yes	No	Yes	Yes
TC5	Yes	N/A	Yes	N/A	Failed to compile	-	Yes	N/A	Yes	N/A
TC6	No	-	No	-	No	-	Yes	No	Yes	Yes
TC7	No	-	No	-	No	-	Yes	No	Yes	Yes
TC8	No	-	No	-	No	-	Yes	No	Yes	Yes
Total Time	1.160s		0.519s		1.116s		3.106s		0.183s	

Figure 1: Experimental results, where column "Found" indicates whether a bug was detected, followed by column "CE" showing whether a counterexample was provided. The line "Total Time" represents the CPU time used for verification

SWC Bug ID	Vulnerability	TC
SWC-101	Integer Overflow	TC1,2
	Integer Underflow	TC3,4
SWC-115	Authorization through tx.origin	TC5
SWC-110	Static array out-of-bounds	TC6
	Dynamic array out-of-bounds	TC7,8

Figure 2: Test case design based on SWC registry

Referências

- ESBMC no GitHub: `https://github.com/esbmc/esbmc`
- Release 7.9: `https://github.com/esbmc/esbmc/releases/tag/v7.9`
- Wei et al. 2023 (arXiv): `https://arxiv.org/abs/2301.10268`
- SWC Registry: `https://swcregistry.io`
- Song et al. 2022: `https://ssvlab.github.io/lucasccordeiro/papers/icse2022.pdf`

[10] `https://arxiv.org/abs/2301.10268`