1L-01

前処理による静的解析ツール Solhint の解析性能性能向上

橋本　樹 †　　小宮　常康 ‡

電気通信大学 †, ‡

1　はじめに

スマートコントラクトは資産を扱うプログラムである。このプログラムは Ethereum というプラットフォーム上で実行され、Solidity というプログラミング言語で記述される。スマートコントラクトは経済合理性の観点より攻撃される可能性が高いため、プログラマは静的解析などにより脆弱性を検知することが望ましい。17 種類のスマートコントラクトの脆弱性を検知する静的解析ツールに Solhint[1] がある。Solhint が検知する攻撃手法 Reentrancy Attack は Ethereum に対して 2015 年から 2020 年までに 2 番目に多く使用された [2]。しかし Solhint では関数をまたいだフローによって引き起こされる Reentrancy Attack の脆弱性は検知できない。そこで本研究では、プログラム変換による Solidity プログラムの前処理により、Solhint による Reentrancy Attack の脆弱性検知性能を向上させることを図った。

2　Reentrancy Attack

スマートコントラクトが Reentrancy Attack [3] の攻撃を受けると、プログラマが再帰的に呼び出されることを想定していない関数がフォールバック関数により再び呼び出され、二重送金などの不正が行われる。

図 1 は Reentrancy Attack の攻撃ベクトルのあるスマートコントラクトである。プログラマは関数 withdraw が実行された際、トランザクション送信者に対して Ether 保有量を転送し、スマートコントラクトに保管されている Ether 保有量 shares が 0 に書き換えられることを想定している。

クラッカーがスマートコントラクト Reentrancy を攻撃するシナリオは次の通りである。クラッカーはまず攻撃対象として、Reentrancy Attack の脆弱性を見つけだす。次にその脆弱性をついて二重送金を実現するために、クラッカーはフォールバック関数が実行されると攻撃対象スマートコントラクトの関数 withdraw が呼び出されるスマートコントラクトを作成してデプロイする。フォールバック関数は Ether 受信時に自動的に呼び出される。この状態で攻撃対象の関数 withdraw を呼び出してメソッド transfer により Ether が転送されると、Ether の受信に伴いフォールバック関数（攻撃対象の関数 withdraw) が呼び出されることとなる。このフォールバック関数呼び出しは shares の書き換え (6 行目）の前に割り込んで実行されるため、Ether 保有量が複数回不正に引き出されてしまう。

```
1 contract Reentrancy {
2   mapping(address => uint256) shares;
3   function withdraw() external {
4     uint256 amount = shares[msg.sender];
5     msg.sender.transfer(amount);
6     shares[msg.sender] = 0;
7   }
8 }
```

図 1　Reentrancy Attack の脆弱性のあるコード

3　Solhint における Reentracy Attack の検知方法と問題点

Solhint はパースされた AST を探索することで、個々の関数本体ごとに脆弱性を生じうるコーディングパターンの検知を行う。Reentrancy Attack のアンチパターンは「Ether 転送メソッド transfer の呼び出し後、ステートを書き換える」である。まず AST 中の ContractDefinition (コントラクト定義) 型ノードを参照し、ステートの名前をメモする。次に FunctionDefinition (関数定義) 型の内部ノードにおいて MemberAccess (メンバアクセス) 型ノードを参照し、メソッド transfer の呼び出しの確認後、ExpressionStatement (式文) 型ノードを参照し、メモしたステートが書き換えられるかどうかを行きがけ順の深さ優先探索することによって検知する。

例えば図 2 において、関数 getFirstWithdrawalBonus 内で、関数 withdrawReward の呼び出しでメソッド

Improvement of vulnerability detection capabilities of Solhint by preprocessing Solidity programs

†Tatsuki Hashimoto, The University of Electro-Communications

‡Tsuneyasu Komiya, The University of Electro-Communications

Copyright ©2022 Information Processing Society of Japan.
All Rights Reserved.

```
1  function getFirstWithdrawalBonus(
2    address payable recipient) public {
3       require(!claimedBonus[recipient]);
4       rewardsForA[recipient] += 100;
5       withdrawReward(recipient);
6       claimedBonus[recipient] = true;
7  }
8  function withdrawReward(
9    address payable recipient) public {
10      uint amount = rewardsForA[recipient];
11      rewardsForA[recipient] = 0;
12      recipient.transfer(amount);
13  }
```

図 2　Cross Function Reentrancy Attack の脆弱性のあるコード

transfer を呼び出した後 (12 行目) にステートが書き換えられる (6 行目）にも関わらず、transfer とステートの書き換えが同一関数内に記述されていないため Solhint はこの脆弱性を見落とす。脆弱性を検知するためには関数 withdrawReawrd 本体も調べる必要がある。つまり Solhint による解析では Reentrancy Attack の脆弱性のある実行フローが複数の関数を経由している場合、脆弱性を検知することができないという課題がある。

4　前処理による Solhint の検知性能向上

本研究では、先に述べた課題を図 3 のように前処理により、解析しやすい Solidity プログラムに変換することで解決する。既に Solhint は広く使われており、またアンチパターンは Solhint にハードコーディングされたものである。そのため Solhint 自体に手を入れることなく解析性能の向上が可能である本手法はユーザーにとって利便性が高い。

本研究におけるプログラム変換では関数呼び出しを呼び出される関数の本体にインライン置換する処理を行う。静的解析ツール Solhint の解析は AST を探索することによって実行フローを調べるため、プログラム変換後のプログラムは AST に変換できるよう文法を満たす必要があるが、安直なインライン展開によって生じ得る、例えば図 4 のようなローカル変数の衝突 (1, 2 行目）や型の不整合 (3 行目）などは Reentrancy Attack の解析に影響がない。そこでこれらはそのまま残すことにした。再帰呼び出しは、前処理の無限ループを回避するために、再帰呼び出しされる関数本体に一回だけ置換する。インスタンスメソッドは実行時にディスパッチが決定する仕様であるため、virtual 関数呼び出しをどのオーバーライド関数本体に置換すべきか静的に決定することができない。例えばスマートコントラクト A が B を継承して virtual 関数 im1 をオーバーライドしている場合、im1 の呼び出しは図 5 のように A と B の im1 を連結したコードに置換する。

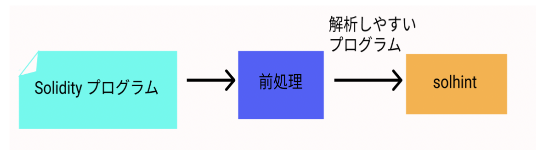

図 3　プログラム変換による前処理

```
1  uint aVal = 0;
2  uint aVal = 1;
3  bool result = "hello";
```

図 4　ローカル変数の衝突と型の不整合の例

```
1  if (true) {
2    console.log("im1 fnDef of B");
3  } else if (true) {
4    console.log("im1 fnDef of A");
5  }
```

図 5　virtual 関数のインライン展開後の例

5　おわりに

本研究では複数の関数を経由する実行フローに Reentrancy Attack の脆弱性がある場合においても、プログラム変換による前処理により、Solhint による解析で脆弱性を検知可能にすることを提案した。Solhint が検知する脆弱性 Multiple Sends においても同様の問題があったが、本研究の前処理により Multiple Sends の脆弱性検知性能も向上している。

参考文献

[1] Protofire, "Solhint", https://github.com/protofire/solhint.
[2] Christof Ferreira Torres, Antonio Iannillo, Arthur Gervais and Radu State, "The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts", arXiv 15 Jan 2021
[3] Nicola Aztei, Massimo Baroletti, and Tiziana Cimoli, "A survey of attacks on Ethereum smart contracts", Conference on Privacy Security and Trust 2017.

Copyright ©2022 Information Processing Society of Japan.
All Rights Reserved.