機密実行環境で動作するSolidity処理系

山地 博之[1]　新城 靖[1]

1. 序論

プログラムを安全に実行する仕組みとして，機密実行環境（Trusted Execution Environment, TEE）が普及している．機密実行環境は高い安全性を保障する一方で，永続化を行うプログラムの開発が難しい．例えば単純に暗号化してファイルに書き出すという実装では，ファイルを古いものと置き換えることによるロールバック攻撃に脆弱となる．

この問題に対して本研究では，機密実行環境で永続的なプログラミング言語を実行可能にする仕組みを提供する．永続的なプログラミング言語は言語機能として永続的な変数を持ち，開発者がそのような変数を明示的に保存しなくても処理系が自動的に保存する．本研究では，永続的なプログラミング言語の 1 つである Solidity の処理系を，機密実行環境で動作させる．Solidity は，ブロックチェーン技術の 1 つである Ethereum において，スマートコントラクトを記述するために設計された．Solidity では，状態変数と呼ばれるデータに加えた変更は，ブロックチェーンに永続的に記録される．

本研究では処理系が行う状態永続化を，機密実行環境が提供する暗号化機能と，OS が提供するファイルシステムを用いて実装する．さらにモノトニックカウンタを用いてロールバック攻撃を防止する仕組みを実装する．

2. 前提条件

本研究で用いる機密実行環境は，ソフトウェアの隔離機能に加えて，実行するプログラムに対して固有の鍵を生成し，それを使って暗号化・復号を行う機能を備えるものを想定する．また信頼できるモノトニックカウンタ，および信頼性を問わない永続記憶が利用可能であることを前提とする．

3. 脅威モデル

本研究では次のような攻撃を想定する．

- Solidity のプログラムが実行時に使用するメモリに対するデータの読み取りや改竄
- 永続化したファイルに対するデータの読み取りや改竄
- 過去に正しい手順で保存されたファイルを復元することによるロールバック攻撃

4. 提案手法

4.1 概要

本論文では，機密実行環境ではなく，保護が適用されない通常の実行環境を，通常実行環境と呼称する．Ethereum を用いた永続的なアプリケーションは，ブロックチェーン上で動作する Solidity のプログラムと，それを利用するフロントエンドのプログラムによって構成される．それと同様に，本研究が提案する処理系を用いたアプリケーションは，機密実行環境で動作する Solidity のプログラムと，それを利用する通常実行環境で動作するプログラムによって構成される．

本研究が提案する処理系を用いたアプリケーションの構成と動作を**図 1** に示す．通常実行環境では Solidity の関数を呼び出すプログラムが動作する．このプログラムは C++で記述され，機械語にコンパイルされた後，通常実行環境にロードされる．機密実行環境では Solidity 処理系が動作する．Solidity 処理系は Solidity からコンパイルされたバイトコードをロードする．通常実行環境のコードは，機密実行環境で動作する Solidity 処理系を介して Solidity の関数を呼び出すことができる．Solidity 処理系は，通常実行環境からの関数呼び出しの要求に応じてバイトコードを実行し，結果を返す．

4.2 状態変数の永続化

Solidity の関数の実行によって状態が変更された場合は，それをマーシャリングして暗号化し，ファイルに書き出す．プログラムを再起動した時は，書き出したファイルを読み出し，復号してアンマーシャリングすることで状態を復元する．暗号化には，機密実行環境が提供するプログラムに対して固有な鍵を使用し，永続化した状態が他のプログラムに復号されることを防ぐ．

また Ariadne[1] の手法に従って，ファイルの置き換えによるロールバック攻撃を防ぐ．ファイルの読み書きはモノトニックカウンタの操作とともに行い，ファイルにカウンタ値を記録し，読み取り時にカウンタ値の一致を確認する．

なおファイル IO やモノトニックカウンタの操作は，機密実行環境から直接行うことができない．そのためこれら

[1] 筑波大学

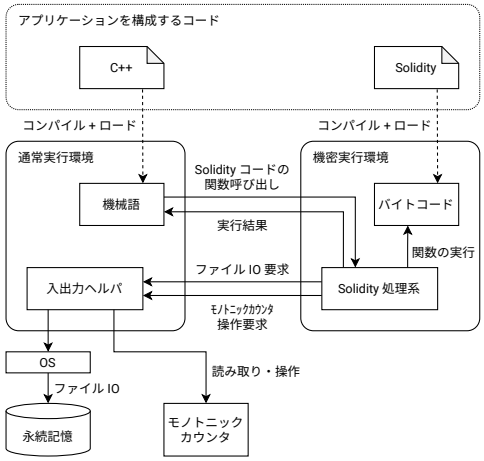

図 1 処理系を用いたアプリケーションの構成と動作

の処理は通常実行環境で動作するコードとして実装し，機密実行環境からの要求に応じて操作を実行する．

4.3 攻撃耐性

Solidity のプログラムは機密実行環境で動作する処理系によって実行される．そのため，実行時に使用するメモリに対するデータの読み取りや改竄は，ハードウェアにより防がれる．

状態を永続化する際は，処理系のプログラムに対する固有の鍵を用いた暗号化を行う．したがって永続化したファイルに対するデータの読み取りや改竄は防がれる．

永続化したファイルが過去のものに置き換えられた場合，記録された値とモノトニックカウンタの不一致によりロールバック攻撃を検知し，復元を中止する．したがって状態を過去のものに戻すことはできない．

5. 実装

本研究では，Intel Software Guard Extensions (Intel SGX) により実現される機密実行環境を使用する．また SGX の機能を利用するための開発キットとして Open Enclave SDK[*1]を使用し，C++を用いて開発を行う．Solidity 処理系には，Enclave EVM (eEVM)[*2]を使用する．eEVM は SGX による機密実行環境で動作するが，永続化の仕組みを持たず，Solidity の状態変数は終了とともに失われる．本研究では 4 章で述べた方法で，eEVM に永続化の実装を追加した．

6. アプリケーションの例

本研究では，4 章で述べた方法に基づき，PIN (Personal Identification Number) で文字列を保護するアプリケーションを実装した．このアプリケーションを構成するコードのうち Solidity で記述する部分は，文字列の保存と取得，

[*1] https://openenclave.io/sdk/
[*2] https://github.com/microsoft/eevm

```
uint triesLeft; bytes32 pin; bytes32 secret;

function getSecret(bytes32 p) public returns (bytes32) {
    if (triesLeft == 0)
        return bytes32("Locked out");
    if (pin == p) {
      triesLeft = 3;
      return secret;
    } else {
      triesLeft--;
      return bytes32("Incorrect PIN");
    }
}
```

図 2 文字列の取得を行う getSecret 関数

および PIN の変更を行う関数を提供する．このうち文字列の取得を行う `getSecret` 関数を**図 2** に示す．この関数は，引数 `p` で与えられた PIN が正しければ状態変数 `secret` に保存された文字列を返す．状態変数 `triesLeft` は，誤った PIN による取得の試みに対する許容回数を表す．与えられた PIN が正しいときは `triesLeft` を 3 に設定し，そうでないときはデクリメントする．

7. 関連研究

ScriptShield[2] は，Intel SGX により実現される機密実行環境でスクリプト言語を実行するためのフレームワークである．ScriptShield は言語処理系を機密実行環境で動作させる点で本研究と共通している．ScriptShield は永続化時の暗号化やロールバック攻撃への耐性を提供しないが，本研究ではその点に対応する．

8. まとめ

本研究は，機密実行環境で動作する永続的なプログラムの開発を容易にすることを目的とし，機密実行環境で動作する Solidity 処理系を実装する．この処理系はロールバック攻撃に耐性を持つため，プログラムの開発者は簡単に永続的なプログラムを開発できる．

本研究では，Intel SGX により実現される機密実行環境で動作する処理系を実装し，永続的なプログラムが実行可能であることを確認した．また実験により，永続化したファイルによるロールバック攻撃に耐性を持つことを確認した．今後の課題として，Ethereum で動作する Solidity と同様に，C++以外の様々な言語からコードを実行可能にすることが挙げられる．

参考文献

[1] Strackx, R. and Piessens, F.: Ariadne: A Minimal Approach to State Continuity, *25th USENIX Conference on Security Symposium*, p. 875–892 (2016).
[2] Wang, H., Bauman, E., Karande, V., Lin, Z., Cheng, Y. and Zhang, Y.: Running Language Interpreters Inside SGX: A Lightweight,Legacy-Compatible Script Code Hardening Approach, *2019 ACM Asia Conference on Computer and Communications Security*, p. 114–121 (2019).