Computer Security Symposium 2020
26 - 29 October 2020

Eth2Vec: 深層学習による言語処理に基づいたスマートコントラクトの安全性解析ツールの設計

芦澤 奈実[1,a)]　矢内 直人[1]　クルーズ ジェイソン ポール[1]　岡村 真吾[2]

概要：Ethereum スマートコントラクトはブロックチェーン上で稼働するプログラムであり，既に多くのプログラムが Ethereum 上で永続的に稼働している．一方、スマートコントラクトでは脆弱性が多く指摘されているにもかかわらず、既存のセキュリティ解析ツールは精度か速度いずれかに問題がある。本稿では，脆弱性を高精度かつ高速に検出する静的解析ツール *Eth2Vec* を提案する．既存の機械学習ベース安全性解析ツールは特徴量を解析者が手動で与えているが、Eth2Vec は言語処理に関する深層学習を通じて脆弱な EVM バイトコードの特徴量を暗黙知に学習する．これにより, 解析対象の EVM バイトコードを入力に与えられた際, 学習済みの脆弱なコードとの類似度を通じて脆弱性を検出できる．Etherscan など既存の公開データベースを用いて実験も行っている。

キーワード：Ethereum, スマートコントラクト、ブロックチェーン、深層学習、静的解析

Integration of Deep Learning with Smart Contract Security Analysis

Nami Ashizawa[1,a)]　Naoto Yanai[1]　Jason Paul Cruz[1]　Shingo Okamura[2]

Abstract: Ethereum smart contracts are programs run on blockchains and many contracts have been deployed recently. In spite of identifying vulnerabilities on the smart contracts, the existing security analysis tools have limitation on either accuracy or throughput. In this paper, we present a new static analysis tool which provides a high accuracy and a high throughput on the vulnerability analysis. The presented tool takes EVM byte codes of an analysis target as input and then identifies vulnerabilities by obtaining features by virtue of a deep-learning-based assembly clone detection, Asm2Vec (IEEE S&P 2020). Here, we newly develop modules to incorporate EVM byte codes into Asm2Vec as well because it does not originally provide a compiler for the smart contracts.

Keywords: CSS 2020, LaTeX, style files

1. はじめに

1.1 背景

ブロックチェーン上でプログラムの実行機能を提供するプラットフォームとしてのスマートコントラクトにおいて、最大のシェアを持つものが Ethereum [26] であり、分散アプリの開発にしばしば用いられる。大雑把には、Ethereum スマートコントラクトではプログラムがブロックチェーン上に格納され、ランタイム環境である Ethereum Virtual Machine (EVM) により EVM バイトコードとして実行される。開発者の観点からは、主な組み込み関数だけを通じてスマートコントラクトを構築することで、ブロックチェーン技術の恩恵を得ることが可能となる。

ブロックチェーンはその透明性および非中央集権化された機能により、誰にでも稼働中の EVM バイトコードが読めることから、攻撃者観点からはコードの悪用が容易である [28]。加えて、金銭的に価値のある情報を扱うため、しばしば被害額の大きな犯罪を誘引してしまう。例えば、

[1] 大阪大学
Osaka University
[2] 奈良工業高等専門学校
National Institute of Technology, Nara College
[a)] n-ashizawa@ist.osaka-u.ac.jp

2016 年 6 月の”The DAO”事件では、60 億円相当以上の仮想通貨 Ether が脆弱性を踏み台に盗まれた。

文献 [28] によると、Ethereum はコードの複雑さや知見の不足からセキュリティの保証が難しい一方, 一度ブロックチェーン上にデプロイされたコントラクトはその透明性からコードの修正も削除もできず、永続的に踏み台として利用される。実際に多くの攻撃が報告 [1] されていることから、開発者にとっては、その記述したコードが脆弱かあらかじめ特定できることが望ましい。このような観点から、Ethereum スマートコントラクトについてプログラムの安全性解析を行うツールが多数開発されている [4]。

上述した背景において、本稿の主題は「Ethereum コントラクトのソースコードから、高速かつ高精度で多岐に渡る脆弱性を判定する静的解析手法の実現」である。ここでいう静的解析とは、解析対象のソースコードのみを与えられ、実行することなくコードに何らかの脆弱性があるか判定することを意味する。直観的には、コードの静的解析をあらかじめ行うことで、脆弱なコードがデプロイ及び踏み台にされるような状況を未然に防ぐことが可能となる。

しかし、静的解析は一般に二つの問題点がある。(1) 精度が限られていること、また、(2) 解析に大幅な時間を要することである。例えば、EVM のバイトコードからのディスアセンブリ [2], [17], [22], [27] は、プログラムそのものが脆弱か判定する機能は一般には持たず、また、既存研究はアセンブリの可読性の改善に注力している。このため解析自体はしばしば手動化されており、偽陽性・偽陰性が増加する。一方、解析対象のコードから制御フローグラフ (CFG) を抽出するシンボリック実行 [3], [14], [23], [25] は、解析を自動化することで高い精度を挙げている。しかし、CFG を構築するために解析対象のプログラムが取りうる全状態を探索する必要があり、計算時間が膨大となる。

1.2 貢献

本稿では Ethereum スマートコントラクトの高水準言語 Solidity から脆弱性の有無を高速かつ高精度に検出する静的解析ツール *Eth2Vec* を提案する。Eth2Vec は言語処理に関する深層学習に基づいており、解析対象の Solidity を入力することで、類似したコードを出力する。これにより、例えば脆弱なコードとの類似度計算を通じて、脆弱性の専門知識を持たないユーザでも解析が行えるようになる。とくに、Solidity のコンパイラも内蔵することで、開発者においても手元で記述した高水準言語の解析が行える。

本稿の技術的貢献は、Ethereum スマートコントラクトの機械学習に基づく解析において、学習サンプルが不十分な脆弱性についても高精度で解析できる点にある。機械学習を使った既存の解析手法 [16], [24] も知られているが、特徴量の抽出が適切に行えていない脆弱性については、判定精度が著しく低下する. 一般に機械学習では特徴量を手動で抽出するが、スマートコントラクトはコードサンプルや公開利用可能な知見が不十分であることから、特徴量の抽出自体が容易ではない [28]。（詳細は 3 節に記載する。）

Eth2Vec では言語処理の観点から深層学習を用いることで, 上述した問題を解決する。既存手法 [16], [24] が手動で特徴量を与えて学習することに対し、大まかには深層学習は入出力のみに注目することで特徴量を暗黙知に利用することができる[*1]。すなわち、深層学習を使うことで特徴量の抽出を Ethereum スマートコントラクトの学習における技術的課題から切り分けることができる。また、言語処理に関する深層学習を用いる点もスマートコントラクトの解析において新規性がある。上述した着想のもと, Eth2Vec では Solidity のコンパイルを通じて EVM バイトコードを取得したうえで、EVM バイトコードを処理可能な深層学習に基づく学習機構を新たに設計した。これにより、従来よりも高い精度の脆弱性の検出が可能となった。

2. Ethereum スマートコントラクト

Ethereum においてスマートコントラクトはブロックチェーン上にデプロイされ、「EVM の文脈で決定的に実行される変更不可能なプログラム」として扱われる。このとき、他のブロックチェーン技術同様、変更不可能性を通じてスマートコントラクトコードの信頼性が保証される。これは一度デプロイされたコードは変更も削除もできないことも意味する。

スマートコントラクトはコントラクトアドレスを識別として与えられ、このアドレスを通じて Ether の受け取りと関数の実行を行う。トランザクションおよびその情報は EVM に入力として与えられ、コントラクトが実行される。また、ピアにコントラクト関数を実行する利権を付与するために、Ethereum は gas と呼ばれる Ether の支払”燃料”を設定している。gas は計算の複雑さに依存しており、無限ループなどを防ぐ狙いがある。

スマートコントラクトは一般に Solidity などの高水準言語を用いて記述される。その後に、EVM 内で実行されるバイトコードにコンパイルされる。EVM はスタックを用いる機構となっており、その命令セットは不正確な実装状態を回避するために最小状態を維持する。EVM はネットワーク内のすべてのピアで実行される単一コンピュータといえる。各ピアは EVM のローカルコピーを実行し、処理したトランザクションとスマートコントラクトがブロックチェーンに記録される。

3. 問題設定

本節では問題設定として、解析対象とする状況と要素技術として機械学習について述べる。その後、本稿で取り組

[*1] Momeni ら [16] はニューラルネットワークも用いているが、文献を読む限りでは手動で特徴量を与えている。

む技術的課題について述べる。

3.1 解析対象

本稿では Ethereum スマートコントラクトの静的解析、とくに高水準言語である Solidity を対象にした安全性解析として、脆弱性の有無及びその種別の判定を行う。これはスマートコントラクトの開発者が手元で開発したコードを解析する状況を想定している。

潜在的な応用例として、Ethereum スマートコントラクトのコード作成者が自ら記述しているコードに脆弱性がないか確認できる。このとき、コード作成者は脆弱性に関する知識を有していなくても構わないものとする。すなわち、脆弱性の有無を自動的に判定する機能をツール内部にあらかじめ設けることで、ユーザは自らのコードをツールに入力として与えるだけで、どのような脆弱性があるか確認できることを目指す。これは C 言語などの従来言語と比べて、Ethereum スマートコントラクトは標準化された知見など著しく不足しているためである [28]。また、Solidity のコードが難読化されている状況は考えない。これは著者らが認識している限りでは、難読化された Ethereum スマートコントラクトのコードは存在しないためである。

なお、以降では Ethereum スマートコントラクトを単にスマートコントラクト、また、Solidity で記述されたコードをコントラクト、ひとつ以上のコントラクトから構成されるコードをコントラクト・ファイル、脆弱性の評価用となるコントラクトを評価用コントラクト、コントラクトをコンパイルした結果得られるコードを EVM バイトコードと呼ぶ。また、コントラクト内における最大のコード単位は関数であり、ライブラリ関数も関数に相当する。

3.2 機械学習

本稿では安全性解析に機械学習を用いる。機械学習は大まかには、データを与えてモデルを学習させることで、未知のデータをモデルに与えた際に、それがどの特徴量の集合に最も近いか予測する。このとき機械学習のモデルに対し、脆弱なスマートコントラクトのコードとその脆弱性の種別を学習させることで、予測への入力として解析対象のコードを与えた際に、安全性の解析として脆弱性の有無とその種別の判定が可能となる。なお、モデルの学習は Ethereum スマートコントラクトの公開データベースを利用して行うものとする。本稿で取り扱う機械学習の問題は以下のように定式化される。

問題の定式化：コントラクトの集合を $\mathcal{C}$, 互いに独立した脆弱性の集合を $\mathcal{V}$、$\mathcal{V}$ のサイズを $|\mathcal{V}|$ とする。また、各コントラクト $c_i \in \mathcal{C}$ が持つ脆弱性を、任意の l において、$V_i = \{v_1^i, \cdots, v_l^i\} \in \mathcal{V}^l$ とする。このとき、機械学習によるスマートコントラクトの脆弱性は以下のモデルを探索する問題として定義される：

モデル M は, 任意の整数 $n \in \mathbb{N}$ においてコントラクトと脆弱性の組 $CV = \{(c_1, V_1), \cdots, (c_n, V_n)\}$, および解析対象のコントラクト $c_t \in \mathcal{C}$ を入力に与えられ、$d = |\mathcal{V}|$ 個の要素を持つ実数値の集合 $\{\epsilon_i^{c_t}\}_{i \in [1,d]} \subseteq \mathbb{R}^d$ を出力する関数、すなわち $M(CV, c_t) \to \{\epsilon_i^{c_t}\}$ とする。ここで、任意の i において $\epsilon_i^{c_t}$ は $\mathcal{V}$ に属する脆弱性に関する確率を表す。

3.3 技術的課題

機械学習を用いたスマートコントラクトの解析ツール [12], [13], [16], [21], [24] はこれまでにも提案されているが、既存ツールでは既知の脆弱性に対し実際に解析できる脆弱性の種類が限られている。すなわち、上述した定式化においては $d \leq |\mathcal{V}|$ となっていた。例えば、Momeni ら [16] のツールでは 36 種類の脆弱性を学習したにもかかわらず、高精度で検知できた脆弱性は 16 種類のみであった。

上述した脆弱性の種類が限られる課題は、既存研究では特徴量の抽出が適切に行えていないことに起因する。機械学習は一般に手作業で特徴量を抽出し、モデルへの入力として明示的に与えたうえで学習を行うことが前提である。しかしながら、Ethereum スマートコントラクトは C 言語など従来言語の解析と比べて歴史が浅く、どのような特徴量に着目すべきか非自明である。直観としては、最適となるようなモデル M の要件が不明であることを意味する。この背景には十分なコードサンプルが得られていないこと [28]、すなわち、上述した定式化においては $\mathcal{C}$ と $\mathcal{V}$ として現実に利用できる空間が限られていることも挙げられる。

4. 提案方式

本節では提案方式 Eth2Vec を提案する。まず、前節で述べた技術的課題に対する設計方針を述べたのち、ツール全体像と用いる要素技術について紹介する。

4.1 設計方針

Eth2Vec では前述した技術的課題を、言語処理に関する深層学習を用いることで解決を図る。大まかには深層学習は「特徴量抽出をブラックボックスに扱う」ことで、特徴量の抽出を学習に関する技術的課題から切り分けている。これにより、注目すべき特徴量が不明であり適切に抽出できない脆弱性に関しても、脆弱なコード自体の学習を通じて解析ができるようになる。

ここでいう言語処理の深層学習とは、Word2Vec のようにニューラルネットワークにテキストデータを直接与え、各単語あるいは各段落ごとにベクトル化したうえで類似度を計算することを意味する。コードを処理する場合は、関数単位などでこれらの処理を行う。とくにコードの安全性解析においては、脆弱なコードを用いてモデルを学習させ、また、解析対象のコードを入力した際に脆弱なコードとの

類似度計算を行うことで脆弱性の有無を判定できる。

この実現に向けて、Eth2Vec ではバイトコードを処理するニューラルネットワークとしてアセンブリ用クローン検知ツール Asm2Vec [5] を要素技術として利用する。なお、Asm2Vec は Ethereum スマートコントラクトには非対応であり、そのままでは EVM バイトコードは解析はできない。このため、Eth2Vec の設計に際し、Asm2Vec の入力に変換するモジュールも新たに設計した。詳細については以下に記載する。

4.2 ツール全体像

Eth2Vec は大きく二つのモジュールから構成される。アセンブリ用ニューラルネットワーク Asm2Vec [5] と, Solidity から Asm2Vec が参照する入力を生成する EVM Extractor である。

Eth2Vec の処理の流れを図示すると図 1 になる。まず、バイトコードを扱えるニューラルネットワークとして Asm2Vec [5] を用いる。Asm2Vec はバイトコードを json 形式に変換することで教師なし学習を行い、各リポジトリから取得したアセンブリコードの類似度計算を行うツールである。大まかには関数単位、ブロック単位、命令単位の順でコードをベクトル化することで学習し、類似度計算を行う。このとき、EVM Extractor は EVM バイトコードを構文解析することで、この Asm2Vec に与える入力として json ファイルを用意する。つまり、Ethereum においてコントラクト単位、関数単位、ブロック単位、命令単位に分けることで Asm2Vec に渡していく。なお、コントラクトの脆弱性については、学習データをあらかじめ既存の解析ツール [6], [14] を用いて脆弱性の有無とその種別を把握することで、脆弱なコードとの類似度計算から評価用コントラクトの脆弱性を判定する。

4.3 要素技術

前述した通り Eth2Vec では要素技術に Asm2Vec [5] と EVM Extractor を用いる。以下に各機能の詳細を述べる。

4.3.1 Asm2Vec

Asm2Vec [5] はアセンブリコードに対する教師なし学習を通じてコードクローンを検知する技術である。直観的には Word2Vec の拡張であるが、アセンブリコードに拡張したことで、コードのベクトル化する対象範囲が複雑化および広範囲化している。

より具体的には、複数の関数からなるアセンブリコードを与えられ、命令 (instruction) 単位でベクトル化および類似度計算を行う。これを命令に関するブロック単位、関数単位およびリポジトリ単位で再帰的に行うことで、分布が同じになるようなコードをクローンとして認識する。形式的には以下を表現するような目的関数として計算される：

$$\sum_{f_s}^{RP}\sum_{seq_i}^{\mathcal{S}(f_s)}\sum_{in_j}^{\mathcal{I}(seq_i)}\sum_{t_c}^{\mathcal{T}(in_j)}\log \mathbf{P}(t_c|f_s, in_{j-1}, in_{j+1}), \quad (1)$$

ここで RP はリポジトリ、f_s は各リポジトリ関数、seq_i は複数の命令からなるブロック、in_j は各命令、t_c は現在の命令に関するトークンであり、$\mathcal{S}(f_s)$ は複数の命令ブロック、$\mathcal{I}(seq_i)$ は命令のリスト、$\mathcal{T}(in_j)$ はトークンのリストをそれぞれ表す。詳細は後述するが、Eth2Vec の実現にあたり、これを Ethereum スマートコントラクトに応用する。

4.3.2 EVM Extractor

EVM Extractor は EVM バイトコードを、Asm2Vec が受け取れる形式に構文解析するモジュールである。EVM Extractor は以下に示す階層構造のように、Solidity ファイルを構文解析する。

```
data(dict)
 ├─ name(string):  file name
 ├─ md5(string):  md5 hash
 └─ functions(list)
     ├─ name(string):  function name
     ├─ sea(number)
     ├─ see(number)
     ├─ id(number)
     ├─ call(list of number)
     └─ blocks(list)
         ├─ name(string):  block name
         ├─ bytes(string)
         ├─ sea(number)
         ├─ eea(number)
         ├─ id(number)
         ├─ call(list of number)
         └─ src(list of list)
```

まず最上位にある `data` が解析対象となるコントラクト・ファイルを指しており、第 2 階層がファイル依存の情報を指しており、第 3 階層が関数依存、第 4 階層がブロック依存の情報をそれぞれ表す。また、各層において `sea` は現在の関数あるいはブロックの開始アドレス、`see` あるいは `eea` はそれぞれの層における次の関数あるいはブロックの開始アドレスをそれぞれ表す。また、`call` が現在の関数あるいはブロックから呼ばれた関数あるいはブロックの識別子を指す。最下層として、`src` に実際の EVM バイトコードの命令をその番地とともに格納する。

4.4 目的関数の設計

Eth2Vec の目的関数は、Asm2Vec [5] の目的関数を 4.3.2 節で述べた構文に拡張することで設計する。具体的には、(1) 式を継承することで、以下のようになる：

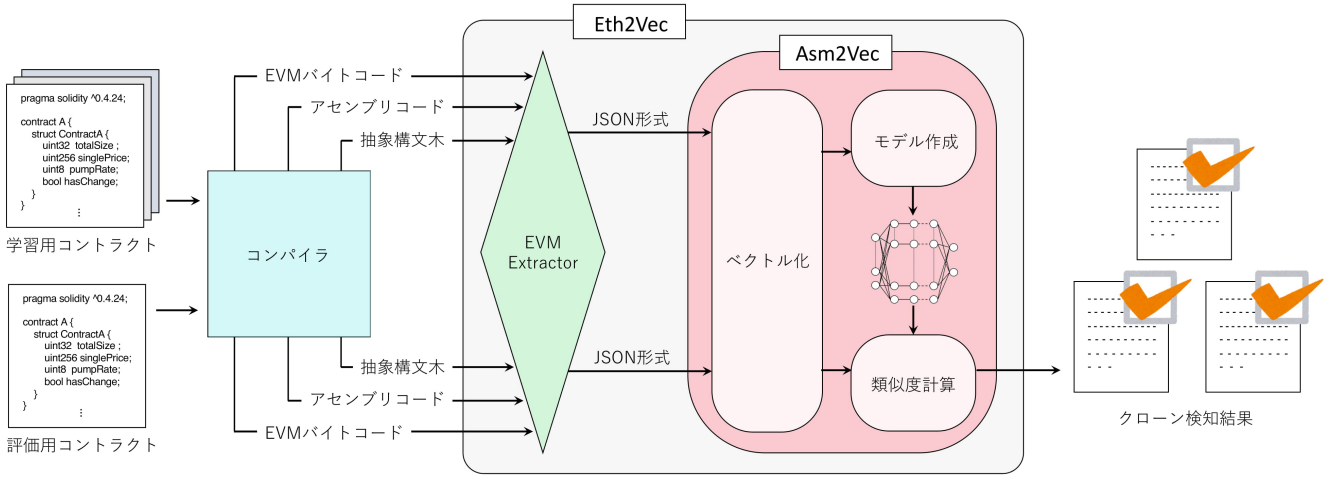

図 1 Eth2Vec の全体図

$$\sum_{C_i}^{Dict}\sum_{f_s}^{\mathcal{U}(C_i)}\sum_{seq_i}^{\mathcal{S}(f_s)}\sum_{in_j}^{\mathcal{I}(seq_i)}\sum_{t_c}^{\mathcal{T}(in_j)}\log \mathbf{P}(t_c|C_i, in_{j-1}, in_{j+1}), \tag{2}$$

ここで $Dict$ はコントラクト・ファイル、C_i は各コントラクト、$\mathcal{U}(C_i)$ は複数の関数、f_s は関数をそれぞれ表す。それ以外の部分は (1) 式と同様である。ここで、c_i までが前節に述べた第 2 階層の `file` までで含まれる。(2) 式は、コントラクトとその命令群をあたえられ、現在の命令トークン t_c のログ確率を最大化している。直観的に、現在の命令と隣接する命令の文脈から得られるベクトルを通じて、コントラクトごとの意味論を計算する。

上述した内容を表現する目的関数の設計詳細について以下に述べる。コントラクト・ファイル $Dict$ を与えられ、まず各コントラクト C_i における関数 f_s をベクトル表現し、$\overrightarrow{\theta}_{f_s}$ とする。また、隣接する命令 in の平均ベクトル表現 $\mathcal{CT}(in)$ として、その命令自身のベクトル表現とオペランドのベクトル表現の連結として、以下を定義する。

$$\mathcal{CT}(in) = \overrightarrow{v}_{\mathcal{P}(in)} || \frac{1}{\mathcal{A}(in)} \sum_{t}^{\mathcal{A}(in)} \overrightarrow{v}_t, \tag{3}$$

ここで $\mathcal{P}$ はある一つの命令 in に対するトークン、$\mathcal{A}(in)$ は in に対するオペランドのリストを、$||$ は連結をそれぞれ表す。このとき、任意のコントラクト c_i において、その最小単位である f_s 及び j 番目の命令 in_j に関する $\mathcal{CT}(in_{j-1})$ と $\mathcal{CT}(in_{j+1})$ で平均化することで、隣接する命令との関数 f_s 内での結合を含めた評価式を以下のとおり定義する。

$$\delta(in_j, f_s) = \frac{1}{3}\left(\overrightarrow{\theta}_{f_s} + \mathcal{CT}(in_{j-1}) + \mathcal{CT}(in_{j+1})\right). \tag{4}$$

これにより、(2) 式における ログ確率は $\mathbf{P}(t_c|C_i, in_{j-1}, in_{j+1}) = \mathbf{P}(t_c|\delta(in_j, f_s))$ と置き換えることができる。文献 [5] によると、$X = (\overrightarrow{v}_t)^T \times \delta(in_j, f_s)$ と置いたとき、これはシグモイド関数 $\sigma(X) = \frac{1}{1+e^{-X}}$ を用いることで、言語処理における k-負例サンプリング [11], [15] として以下のように近似できる.

$$J(\theta) = \log \mathbf{P}\left(t_c|\delta(in_j, f_s)\right) \approx \log\left(\sigma\left(X\right)\right) + \sum_{i=1}^{k} \mathbb{E}_{t_d \sim P_n(t_c)}\left(\llbracket t_d \neq t_c \rrbracket \log\left(\sigma\left(X\right)\right)\right). \tag{5}$$

ここで $\llbracket t_d \neq t_c \rrbracket$ は恒等関数であり、関数が成り立つとき 1、そうでないなら 0 を返す。また、$\mathbb{E}_{t_d \sim P_n(t_c)}$ はトークン t_c に関するノイズ分布 $P_n(t_c)$ に従う、トークン t_d を標本化するサンプリング関数である。上述した (5) 式が Eth2Vec の目的関数となる。直観的には現在の命令トークン t_c の確率を最大化させ、それ以外の命令トークン t_d を下げている。このとき、導関数を通じて勾配計算は θ_{f_s} に関して以下のように設定できる。

$$\frac{\partial J(\theta)}{\partial \overrightarrow{\theta}_{f_s}} = \frac{\overrightarrow{v}_{t_c}}{3}\left(1 - \sigma\left(X\right)\right) + \frac{\overrightarrow{v}_{t_c}}{3}\sum_{i}^{k} \mathbb{E}_{t_d \sim P_n(t_c)}\left(\llbracket t_d \neq t_c \rrbracket \left(1 - \sigma\left(X\right)\right)\right)$$

上述した勾配計算は以下のように近似計算することで、より直観に即した勾配を持たせることができる。

$$\frac{\partial J(\theta)}{\partial \overrightarrow{\theta}_{f_s}} \approx \frac{\overrightarrow{v}_{t_c}}{3}\sum_{i}^{k} \mathbb{E}_{t \sim P_n(t_c)}\left(\llbracket t = t_c \rrbracket - \sigma\left(X\right)\right) \tag{6}$$

直観的には (6) 式は現在の命令に一致するトークンなら勾配を正の方向に、そうでないなら負の方向に動かすように本来の導関数を近似している。同様に、現在の命令トークン $\overrightarrow{v}_{t_c}$ に関する勾配も以下のように近似される。また、ある命令に対するトークン $\overrightarrow{v}_{\mathcal{P}(in)}$ と現在の命令に関するオペランド $\overrightarrow{v}_t$ も同様に勾配を計算できる。

$$\frac{\partial J(\theta)}{\partial \overrightarrow{v}_{t_c}} \approx \left(\llbracket t = t_c \rrbracket - \sigma\left(X\right)\right) \cdot \delta(in_j, f_s) \tag{7}$$

上述した勾配を用いて学習を行う。なお、Asm2Vec はアセンブリで同様の計算を用いているため、学習アルゴリズム自体は Asm2Vec に準拠する。詳細は紙面上割愛する。

4.5 実装

Eth2Vec は Kam1n0[*2] と py-solc-x[*3] を用いて実装されている。まず核モジュールである Asm2Vec は Kam1n0 のアプリケーションとして実装されており、このうち Eth2Vec の実装では主に DisassemblyFactoryIDA.java と ExtractBinaryViaIDA.py を変更した。Asm2Vec では、これらは IDA[*4] を実行してディスアセンブリしたバイナリコードの情報を Asm2Vec 内に保存するクラスである。具体的には、ExtractBinaryViaIDA.py はディスアセンブリしたバイナリコードの情報を IDA から抽出し json ファイルとして保存したのち、これを DisassemblyFactoryIDA.java で読み込むことで、Asm2Vec 内にバイナリコードの情報が保存される。しかし、Eth2Vec の実装に際し、EVM バイトコードは IDA では扱えない。このため、IDA を呼び出さず py-solc-x を用いて Solidity をコンパイルし、その際に生成されるアセンブリ, 抽象構文木 (AST), バイナリコードを参考に, コードの情報を抽出するように ExtractBinaryViaIDA.py を変更した。なお、学習方法は Asm2Vec [5] の実装をそのまま利用する。

5. 実験

本節では Eth2Vec の実験評価について述べる。まず実験目的を述べたのち、Eth2Vec の評価に用いるデータセットと学習方法を述べる。その後に実験結果について示す。

5.1 実験目的

Eth2Vec の性能評価として、既存の脆弱なコントラクトコードの学習を通じて、解析対象のコードの脆弱性が検知できるか確認する。この実現にあたり、本稿ではまず Solidity で記述されたコードに対するクローンの検知精度を評価する。次に、学習済みのコードと解析対象のコードの類似度計算を通じて、保持するすべての脆弱性の種類を判定できるか確認する。

5.2 実験設定

実験はスマートコントラクトのクローン検知と発見できる脆弱性の種類、二つの段階に分けて行う。以下に各実験の設定について述べる. なお、現在公開されている既存の機械学習ベース静的解析ツールとして SmartEmbed [9] があり、本稿ではこれをベースラインとして比較する。他の機械学習ツール [16], [24] は本稿執筆時点では非公開であり、実験ができなかった。

5.2.1 クローン検知

学習データに対するクローンとして与えた評価用コントラクトが、実際にクローンとして認識されるか確認する。具体的には、学習データとして SmartEmbed [9] で記載されていたコントラクトのデータセットを利用する。このとき、評価用となるデータセットとして SmartEmbed [9] のデータセットからひとつずつ関数のクローン検知を行い、それぞれの関数に対してクローンとして判定された関数の数を数える。直観的には学習データに含まれるコントラクトを評価用コントラクトとして与えることで、すべてのコントラクトを該当するコントラクトのクローンとして認識されるか確認する。なお、クローンとしての閾値は 0.8 とする。

5.2.2 脆弱性の種類数

評価用コントラクトを通じて、クローンと判定されたコントラクトが実際に脆弱であるか確認する。学習には前節同様、SmartEmbed [9] で記載されていたデータセットを利用し、評価用データセットとして、どのような脆弱性を持つか既知であるコントラクトのデータセットを使用する。このとき、評価用データセットが持つ既知の脆弱性を、正解データ (Ground Truth) とする。

比較対象のクローン検知ツールとして SmartEmbed を用いる。評価用データセットのコントラクトを Eth2Vec、SmartEmbed のそれぞれに与えた際に出力されるクローンの持つ脆弱性が、正解データに一致するか確認する。なお、クローンがどのような脆弱性を保持するかは、既存の解析ツール [6], [14] を適用することで取得する。しかし SmartEmbed は脆弱性検知を備えたツールであり、既存の解析ツールを用いなくても、評価用コントラクト及びそのクローンがどのような脆弱性を持つか判定可能である。そのため結果には、既存の解析ツールで脆弱性の有無を判定したものと、SmartEmbed 自体で判定したものの両方の結果を記述する。

5.3 実験結果

表 1 に、Eth2Vec でクローン検知を行った結果を示す。「関数」は各コントラクトファイルに含まれる関数の数を表し、「クローン」は各コントラクトファイルに対して検知したクローンの数を表す。

表 1 ファイルに含まれる関数とクローンの数

	平均	合計
関数	18.87	3774
クローン	114.07	22814

表 2 は Eth2Vec と SmartEmbed を用いて脆弱性検知を行った結果ある。表 2 には、検知を試みた脆弱性の種類や数、及び Eth2Vec と SmartEmbed で検知した脆弱性の割合を示す。Eth2Vec でクローン検知を試みた結果、いくつかの関数については全く同じ内容の関数を発見できているが、全く異なる関数をクローンとして判定している場合も

[*2] Kam1n0 version 2.0.0: https://github.com/McGill-DMaS/Kam1n0-Community
[*3] py-solc-x: https://pypi.org/project/py-solc-x/
[*4] IDA: https://www.hex-rays.com/products/ida/

表 2　SmartEmbed と Eth2Vec のクローン検知精度

脆弱性の種類	コントラクト	コントラクト内の脆弱性の数	SmartEmbed		Eth2Vec
			既存の解析ツール	SmartEmbed	既存の解析ツール
Overflow/Underflow	EthConnectPonzi	2	1.00	0.00	0.00
	BecToken	5	1.00	0.00	0.60
	integer_overflow_1	1	0.00	0.00	0.00
Blockhash/Timestamp	SmartBillisons	6	1.00	0.00	0.67
	Ethraffle	1	1.00	0.00	0.00
	LuckyDoubler	1	1.00	0.00	0.00
Implicit Visibility/HoneyPot	Multiplicator	2	1.00	0.00	0.00
	PrivateBank	4	1.00	0.00	0.25
	KingOfTheHill	3	1.00	0.00	0.00
	RichestTakeAll	1	1.00	0.00	0.00
Gas Consumption/Gas Limit	Simoleon	4	1.00	0.00	0.00
	Polyion	4	1.00	0.00	0.00
	Pandemica	2	1.00	0.00	0.00
TransferFlaw/ERC-20 Transfer	UselessEthereumToken	2	1.00	0.00	0.00
	TacoToken	2	1.00	0.00	0.50
Reentrancy	Reentrancy	1	0.00	0.00	0.00

見受けられた。これは、EVM Extractor により抽出される情報量が本来 Asm2Vec が入力として求める情報量より少ないことに起因すると考えられる。

また、脆弱性検知においてはクローン検知の精度が SmartEmbed を下回るために、SmartEmbed よりも検知できた脆弱性の割合が低くなった。しかし、SmartEmbed がコントラクト (integer_overflow_1、Reentrancy) の脆弱性を発見できなかったのは、SmartEmbed の学習データ内にそれらのコントラクトが含まれないため、クローンとして検知できないためと考えられる。これらのコントラクトに対する類似度は 0.8 を超えていたが、同じ脆弱性は含まなかった。一方で、Eth2Vec は評価用コントラクトの各関数に対して同じコントラクトの同じ関数以外をクローンとして検知した場合、そのクローンは評価用コントラクトと同じ脆弱性を含んでいた。つまり、Eth2Vec は SmartEmbed と比べて、評価用コントラクトと全く同じコントラクトが学習用データに含まれていなくても、同じ脆弱性を含むクローンを検知することができると考えられる。

Eth2Vec において、今後改善されるべき制約条件を以下に記載する。まず、教師あり学習は実装できていない。テキスト処理においては教師なし学習よりは教師あり学習の方が分類問題に強いことが知られている [10]。直観的には、解析対象のコードがどのような脆弱性を持つかという評価は脆弱性に関する分類問題であり、教師あり学習にすることで更なる性能の改善が期待できる。また、実験で用いたデータセットを除いて、現実世界におけるコントラクトを評価できていない。現実世界にデプロイされたコントラクトを実際に評価することで、潜在的な問題を発見することも必要である。

6. 関連研究

機械学習による Ethereum の安全性解析）特徴量抽出を自動化する研究として, Word2Vec と LSTM を組み合わせた Reentrancy 脆弱性を解析する VulDeeSmartContract [19] がある。このツールは Reentrancy に特化しており、Eth2Vec は同様のアプローチでより汎用的な脆弱性検知を可能にしたといえる。

また、コード間の類似度計算ができるツールとして、スマートコントラクトのコードクローンを検出する EClone [12] がある。これはニューラルネットワークは用いていないが、コード間のベクトル計算に適したシンボリック実行も設計している。このため、EClone と組み合わせることで、Eth2Vec の検知能力の改善が期待できる。

最後に、SmartEmbed [9] は、FastText というテキストのベクトル化技術を用いて、スマートコントラクトのバグ検出を行っている。これが Eth2Vec に最も近い。一方、SmartEmbed では発見できるバグの種類数は議論しておらず、本研究とは主たる問題が異なる。また、Word2Vec を用いた Eth2Vec とは技術的側面も異なる。

Ethereum の安全性解析）Ethereum の解析はシンボリック実行 [14] が主流である。シンボリック実行は未知の変数をシンボル変数とする点が、プログラムの外にあるブロックチェーンの情報を参照するスマートコントラクトと相性が良い。近年の動向 [3], [8], [18], [25] では、解析範囲の拡大に注力している。1.1 節に述べたとおり解析に時間はかかるが、これらを教師あり学習としてラベル生成に用いることで、Eth2Vec の解析能力の向上が期待できる。

最後に、コードを実際に実行する動的解析 [6], [20] もあ

るが、動的解析で被害を未然に防ぐためには、解析者は攻撃パターンを自ら実装・実行することが求められる [3]。様々な攻撃を想定した汎用パターン [7] も知られているが，いずれにせよ解析者自身が攻撃に関する知見を求められるため、解析できる状況が著しく制限される。このため、静的解析の方が現実的といえる。

7. まとめ

本稿では Ethereum スマートコントラクトの脆弱性を深層学習で評価するツール Eth2Vec を提案した。Eth2Vec の利点は、特徴量抽出を深層学習により技術的に切り分けることで、脆弱性の検知精度の向上を図った。脆弱性の検知及び精度の向上は今後の課題である。

謝辞 本研究の一部は JSPS 科研費 18K18049 およびセコム財団挑戦的研究助成の助成を受けたものです。

参考文献

[1] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Proc. of POST 2017*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.

[2] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[3] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura. Hunting for re-entrancy attacks in ethereum smart contracts via static analysis. *arXiv preprint arXiv:2007.01029*, 2020.

[4] M. Di Angelo and G. Salzer. A survey of tools for analyzing ethereum smart contracts. In *Proc. of DAPPCON 2019*, pages 69–78. IEEE, 2019.

[5] S. H. Ding, B. C. Fung, and P. Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proc. of IEEE S&P 2019*, pages 472–489. IEEE, 2019.

[6] J. Feist, G. Grieco, and A. Groce. Slither: A static analysis framework for smart contracts. In *Proc. of WETSEB 2019*, pages 8–15. IEEE, 2019.

[7] C. Ferreira Torres, M. Steichen, R. Norvill, B. Fiz Pontiveros, and H. Jonker. Ægis: Shielding vulnerable smart contracts against attacks. In *Proc. of AsiaCCS 2020*. ACM, 2020.

[8] J. Frank, C. Aschermann, and T. Holz. ETHBMC: A bounded model checker for smart contracts. In *Proc. of Usenix Security 2020*. USENIX Association, 2020.

[9] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[10] F. Hill, K. Cho, and A. Korhonen. Learning distributed representations of sentences from unlabelled data, 2016.

[11] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proc. of ICML 2014*, pages 1188–1196, 2014.

[12] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun. Enabling clone detection for ethereum via smart contract birthmarks. In *Proc. of ICPC 2019*, pages 105–115. IEEE, 2019.

[13] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun. Eclone: Detect semantic clones in ethereum via symbolic transaction sketch. In *Proc. of ESEC/FSE 2018*, page 900–903. ACM, 2018.

[14] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proc. of CCS 2016*, pages 254–269. ACM, 2016.

[15] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proc. of NIPS 2013*, pages 3111–3119, 2013.

[16] P. Momeni, Y. Wang, and R. Samavi. Machine learning model for smart contracts security analysis. In *Proc. of PST 2019*, pages 1–6. IEEE, 2019.

[17] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen. Visual emulation for ethereum's virtual machine. In *Proc of NOMS 2018*, pages 1–4. IEEE, 2018.

[18] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *Proc. of IEEE S&P 2020*, pages 414–430. IEEE, 2020.

[19] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access*, 8:19685–19695, 2020.

[20] M. Rodler, W. Li, G. O. Karame, and L. Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Proc. of NDSS 2019*. Internet Society, 2019.

[21] J. Song, H. He, Z. Lv, C. Su, G. Xu, and W. Wang. An efficient vulnerability detection model for ethereum smart contracts. In *Proc. of NSS 2019*, volume 11928 of *LNCS*, pages 433–442. Springer, 2019.

[22] M. Suiche. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF CON*, 25:11, 2017.

[23] C. F. Torres, J. Schütte, et al. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proc. of ACSAC 2018*, pages 664–676. ACM, 2018.

[24] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, pages 1–1 (Early Access), 2020.

[25] K. Weiss and J. Schütte. Annotary: A Concolic Execution System for Developing Secure Smart Contracts. In *Proc. of ESORICS 2019*, volume 11735 of *LNCS*, pages 747–766. Springer, 2019.

[26] G. Wood. Ethereum: A secure decentralised generalised transaction ledger byzantium version. https://ethereum.github.io/yellowpaper/paper.pdf.

[27] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In *Proc. of Usenix Security 2018*, pages 1371–1385. Usenix Association, 2018.

[28] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.