卒業論文 2018 年度 (平成 30 年)

低対話型 Honeypot のコマンド拡張による収集ログの変化の計測

慶應義塾大学 総合政策学部
菅藤 佑太

卒業論文要旨 - 2018 年度 (平成 30 年度)

低対話型 Honeypot のコマンド拡張による 収集ログの変化の計測

PC の普及や IoT デバイスのシステム高度化により，高度な処理系を組むことが可能になった．これによりデバイス上に Linux 系などの OS が搭載された機器が広く人々に使われるようになった．また，Linux 系 OS にリモートログインする手法として SSH がある．これを用いて不正に侵入する攻撃が行われている．侵入された際に侵入者がどのような挙動をしているのかを知る手段として，Honeypot がある．Honeypot は SSH で侵入しやすいような環境を作ることで，侵入者にログイン試行に成功したと検知させ，その際に実行したコマンドのログを収集するものである．また現在では Shell の挙動をエミュレートした Honeypot が広く使用されており，この Honeypot は実行できるコマンドが少ない実装になっている．そのため Honeypot への侵入者に侵入先が Honeypot であると検知されてしまう．そこで事前実験では Honeypot のコマンドを拡張し，拡張をしていない Honeypot とコマンドの拡張をした Honeypot で侵入ログを収集した．収集したログを確率的な算出方法を使用することで比較した結果，より多くの侵入者のコマンド実行ログのパターンを取得できることを示した．本研究ではコマンドを拡張した Honeypot の侵入ログがどれほど実際の OS に不正な SSH の侵入をされた際の侵入ログが近似したのかを検証した．評価として，拡張をしていない Honeypot とコマンドの拡張をした Honeypot と，さらに実際の OS を使用した Honeypot で侵入ログを収集し，比較を行なった．この 3 つの侵入ログを自然言語処理の意味解析を用い，一セッションにおけるコマンドログの意味をベクトル空間上に表現することで，拡張した Honeypot で収集した侵入ログが，拡張をしていない Honeypot で収集した侵入ログよりも，一般的な UNIX ユーザーの実行するコマンドログから離れることを示した．

キーワード:
1. 自然言語処理，2. 意味解析，3. Honeypot，4. SSH

慶應義塾大学 総合政策学部
菅藤 佑太

Abstract of Bachelor's Thesis - Academic Year 2018

Measurement of changes in collection logs by command extension of low interactive Honeypot

By the spread of PCs and the system advancement of the IoT device. I was able to make high processing system. The apparatus that the OS's such as the Linux system were equipped with on a device came to be in this way used for people widely. In addition, technique to perform a remote login for the Linux system OS includes SSH. An attack to invade using this illegally is carried out. A window includes Honeypot what kind of ways an intruder behaves when it was invaded. Honeypot lets an intruder detect it when I succeeded in a login trial by making the environment where it is easy to invade in SSH and collects the log of the command that I carried out on this occasion. In addition, Honeypot which emulated behavior of Shell is used widely now, and there are few commands that this Honeypot can carry out is implemented. Therefore it is detected by an intruder to Honeypot when invasion is Honeypot. Therefore I expanded the command of Honeypot by the prior experiment and collected invasion log in Honeypot and Honeypot which I expanded of the command which were not expanded. As a result of having compared the log that I collected by using a probabilistic calculation method, I showed that I could acquire a pattern of the command practice log of more intruders. Invasion log of Honeypot which expanded the command inspected how long invasion log when it was invaded of unjust SSH was similar to the real OS in this study. I collected invasion log more in Honeypot using the real OS and compared it with Honeypot and expanded Honeypot of the command which were not expanded as an evaluation. I used semantic analysis of the natural language processing in the invasion log of these three and showed that I left the command log that the UNIX user more general than the invasion log that I collected in Honeypot which the invasion log that I collected in Honeypot which I expanded by expressing a meaning of the command log in one session on a vector space did not expand carried out.

Keywords :
1. Natural Language Processing, 2. Semantic analysis, 3. Honeypot, 4. SSH

Keio University, Faculty of Policy Management Studies
Yuta Sugafuji

目 次

図 目 次

表 目 次

第1章　序論

本章では本研究の背景，課題及び手法を提示し，本研究の概要を示す．

1.1　本研究の背景

1.1.1　通信機器の普及

PC の普及や IoT デバイスのシステム高度化により，高度な処理系を組むことが可能になった．これによりデバイス上に Linux 系などの OS が搭載された機器が広く人々に使われるようになった．また，Linux 系 OS にリモートログインする手法として SSH がある．これを用いて不正に侵入する攻撃が行われている．

1.1.2　honeypot

侵入された際に侵入者がどのような挙動をしているのかを知る手段として，Honeypot がある．これは実際の OS を用いたり，Shell の擬似的な挙動をアプリケーション上で実現し，敢えて SSH で侵入しやすいような環境を作ることで，侵入者にログイン試行に成功したと検知させ，その際に実行したコマンドのログを収集する．

1.2　本研究の問題と仮説

SSH の Honeypot は大きく二種類に分けることができ，一つは低対話型 Honeypot，もう一つは高対話型 Honeypot である．低対話型 Honeypot は実際の Shell の挙動をエミュレートしたアプリケーションである．
高対話型 Honeypot は実際の機器を設置し，その中に侵入させログを収集する．その設置時には他のホストに攻撃できないようにネットワークの設定や，root の権限が取られないように user 権限の設定を適切に行う．高対話型 Honeypot は低対話型 Honeypot と比較すると，Honeypot への侵入者が実行できるコマンドが多く，挙動も本物の OS と差異が極めて少なく，侵入先が Honeypot であると極めて検知しされにくい．そのため，高精度な攻撃ログを取得することができる．しかし，Honeypot として適切な設定を行なった OS が，設置後に発見された新たな OS の脆弱性を突かれることで，踏み台にされ他のホストに攻撃をしたりウイルスに犯されてしまうなどの危険を孕んでいる．そのため，設置コストが高く普及率も非常に低い．[3]

一方で低対話型 Honeypot はアプリケーションであるため，root 権限を取られるような危険が極めて少なく，アプリケーション内での脆弱性に限った問題しか存在しない．そのため設置コストが低く，比較的誰でも安全に設置できるため，普及率が高い．しかし，あくまでエミュレーションを行なったアプリケーションであるため，実際の Shell とは異なる挙動や，Honeypot に特有な挙動をすることがある．そのため，設置した Honeypot に侵入した悪意のあるユーザーに侵入先が Honeypot であると検知されてしまう可能性がある．本研究では低対話型 Honeypot に着目する．低対話型で実際の攻撃ログに近いログを収集するには，先述の Honeypot であることの検知を回避する必要がある．そこで本研究では，低対話型に実装されているコマンドの出力を，実際の Shell に近似することで検知を回避できるのではないかと考えた．

1.3 提案手法の実装

先述の Honeypot であることの検知を回避するために，本研究では低対話型 Honeypot を実際の Shell の挙動に近似するために，2 つの実験を行なった．一つは実際の Shell に実装されているが低対話型 Honeypot に実装されていないコマンドの実装した．もう一つは低対話型 Honeypot に特有の異常な挙動を修正を行った．

1.4 予備実験

本研究の予備実験として，SSH の低対話型 Honeypot に実装されていないコマンドで，悪意のある侵入者が使うコマンドを実装することで拡張を行なった低対話型 Honeypot と，素の低対話型 Honeypot でそれぞれ収集したコマンドログの比較を行なった．追加実装を施した SSH の低対話型 Honeypot の方がコマンドパターンとして多く収集できることを示した．

1.5 本研究の評価

提案手法の実装で拡張した低対話型 Honeypot と，素の Honeypot と，高対話型 Honeypot を設置し，それぞれ侵入者が実行したコマンドのログを収集し，比較を行った．収集したコマンドのログはコマンド 1 つ 1 つごとに自然言語処理の手法を用いて意味解析をし，コマンドの意味をベクトル空間上に表現した．本研究では，高対話型のログに近似することで，高度なログが収集できていると考えた．そこで，素の Honeypot と拡張した低対話型 Honeypot と高対話型 Honeypot と比較して，どれほど一般的な UNIX ユーザーの実行するコマンドログから離れたのかを評価した．

1.5.1 予備実験

本研究の予備実験として，SSH の低対話型 Honeypot に実装されていないコマンドで，悪意のある侵入者が使うコマンドを実装することで拡張を行なった低対話型 Honeypot と，素の低対話型 Honeypot でそれぞれ収集したコマンドログの比較を行なった．追加実装を施した SSH の低対話型 Honeypot の方がコマンドパターンとして多く収集できることを示した．

1.6 本論文の構成

本論文における以降の構成は次の通りである．

2 章では，本研究の要素技術となる Shell と Honeypot と自然言語処理について整理する． 3 章では，本研究における問題の定義と，解決するための要件，仮説について説明する． 4 章では，本提案手法について解説する． 5 章では，本研究の事前実験や Honeypot の拡張についての実装方法や実装例について述べる． 6 章では，求められた課題に対しての評価を行い，考察する． 7 章では，関連研究を紹介し，本研究との比較を行う． 8 章では，本研究のまとめと今後の課題，展望についてまとめる．

第2章　本研究の要素技術

本章では，本研究の要素技術となるShellとHoneypotと収集データの扱いについて各々整理する．

2.1　Honeypot

使われているデバイスへの不正なSSHによって侵入された際，実際に攻撃が行えない環境へとフォワードし，その中で攻撃を試行させ，侵入者のログを収集する手段としてHoneypotがある．SSHのHoneypotは低対話型Honeypotと高対話型Honeypotの大きく二種類に分けることができる．

2.1.1　低対話型Honeypot

SSHの低対話型Honeypotは実際のShellの挙動をエミュレートしたアプリケーションである．実際のShellの挙動をエミュレートしただけのアプリケーションなので，脆弱性がアプリケーション内に限られる．そのため，root権限を侵入者に許してしまい，踏み台にされてしまうなどの危険が極めて少ない．しかし，エミュレーションには限界があるため，コマンドやその挙動について，実際のShellとは異なる挙動をすることがある．そのため，侵入者に侵入先がHoneypotであると検知されてしまう．検知されることで，攻撃者は実際の攻撃を行わず，本来取れるはずの攻撃ログが収集できない可能性を含んでいる．そのため，収集ログの精度に問題がある．

2.1.1.1　Kippo

Kippoは，悪意のあるSSHのログイン試行者や侵入者の挙動やログを記録するために使用されるPythonで実装されたSSHの低対話型Honeypotである[4]．Kippoは前身のKojoney[5]に大きく影響を受けている．ネットワークはTwisted[6]というフレームワークで組まれている．Kippoのプロジェクトは低対話型Honeypotとして2009年に登場し，Raspberry Piなどを筆頭としたシングルボードコンピュータの普及と相まって広く設置された．Kippoの機能の特徴としては収集したコマンドログ を時系列データとして保存されており，”playlog”というKippo内にあるプログラムを実行することで，過去のコマンドログ を実際にタイピングしてるかのように出力できる．また，侵入者によってダウンロー

ドされたファイルも実行ができないように保存できる．Kippo は後述の Cowrie の後継実装である．[7] Kippo は IoT デバイスの高度化広く設置された SSH の低対話型 Honeypot のうちの一つであったが，実装されているコマンドも 77[8] と少なく，また Kippo 特有の異常な挙動が存在するなどと多くの問題があった．

2.1.1.2 Cowrie

Cowrie は Python で実装された SSH の低対話型 Honeypot であり，実装は Kippo のコマンドの拡張や，攻撃者がリダイレクトでマルウェアを送り込む手法をとって送り込んだマルウェアを収集可能にしたりするなど，様々な機能を拡張したものである．Kippo 特有の異常な挙動を改善しており，実装コマンド数は 92[9] と Kippo より少し多くなっているものの [10]，Cowrie 特有の異常な挙動もまだまだ多い．

2.1.2 高対話型 Honeypot

高対話型 Honeypot は脆弱性を残した実際の OS を用いた Honeypot である．実際の OS をそのまますると，その OS から外部の他のホストへと攻撃することができてしまう．また，予期しない OS の脆弱性を突かれることで，OS 自体を完全に侵入者に制御されてしまう問題がある．そのため，Honeypot として適切な設定を行う必要がある．

2.1.2.1 Honeynet

2.1.2 で先述した通り，Honeypot で使用される OS から外部への通信で他のホストを攻撃したりするなどの攻撃を行ってしまう問題や，予期しない OS の脆弱性を突かれることで，OS 自体を完全に侵入者に制御されてしまう問題があるため，Honeypot として適切な設定を行う必要がある．そのため，Honeynet ではネットワーク全体を honeywall という独自のファイアウォールの機能を実行する．これは Honeypot のネットワークの設定を管理するだけではなく，ネットワーク介して送信されるすべてのデータの中央集権のポイントして機能する．これによってネットワークが危険にさらされた侵入者からの攻撃から保護することが可能である．[11]

2.1.3 SSH の Honeypot の比較

以上をまとめた SSH の低対話型 Honeypot と SSH の高対話型 Honeypot の比較を行った表を表 2.1 に示す．

表 2.1: 種類ごとの Honeypot の比較

Honeypot の種類	設置コスト (リスク)	Honeypot であることの検知されにくさ
低対話型 Honeypot	設置コストが低い	検知されやすい
高対話型 Honeypot	設置コストが高い	検知されにくい

2.2 Shell

Shell は OS のユーザーのためにインタフェースで，カーネルのサービスへのアクセスを提供するソフトウェアである．本研究での”Shell”はコマンドラインシェルのことを指す．

2.2.1 Secure Shell

Secure Shell（セキュアシェル、SSH）は、暗号や認証の技術を利用して、安全にリモートコンピュータと通信するためのプロトコルである．パスワードなどの認証部分を含むすべてのネットワーク上の通信が暗号化される．[12]SSH における問題としては，通信する上での認証方法には鍵認証を推奨されているが，デフォルトではパスワード認証になっている．パスワード認証のままだとパスワードの総当たり攻撃を受けたり，パスワードが標準のままの設定であることで不正なログイン試行によって侵入を許してしまう．

2.2.2 BusyBox

BusyBox は標準 UNIX コマンドで重要な多数のプログラムを単一のバイナリファイルに含むプログラムである．BusyBox に含まれる，多数の標準 UNIX コマンドで必要とするプログラムの実行ファイルは，Linux という OS を BusyBox だけでディストリビューションできるよう，”Linux 上で最小の実行ファイル”として設計されている．一般にインストールされる実行ファイルは一部だけを実装できるように選択することができる．一般的には BusyBox のコマンドは 200 以上も用意されている．[13][1]．BusyBox をインストールして実際に各コマンドを実行するためには，BusyBox 内にある各コマンドにアクセス可能なように path を通すだけで良い．

2.3 t-SNE

高次元データをで可視化する際に用いられる手法の一つ．高次元空間上の類似度と低次元空間上の類似度をそれぞれ確率分布 pij と pij で表現して，qij と qij を最小化するように確率分布のパラメータを最適化を行う．[14]

[1]今回使用した BusyBox に含まれるコマンドの数は 219

2.4 自然言語処理

人間が日常的に使っている自然言語をコンピュータに処理させる一連の技術である．本研究において，自然言語処理は意味解析のために使用した．意味解析には様々な手法があり，現在では大きくシソーラス解析とベクトル空間分析がある．

2.4.1 混合隠れマルコフモデル

混合隠れマルコフモデルはいくつかの種類のある観測できない行動に対し，その行動に依存する何らかの観測を用いて隠れているデータのモデルを構築するモデルのことである．隠れ変数を Zn とし，観測値を Yn とすると，隠れマルコフモデルは以下の図 2.1 のように示すことができる．

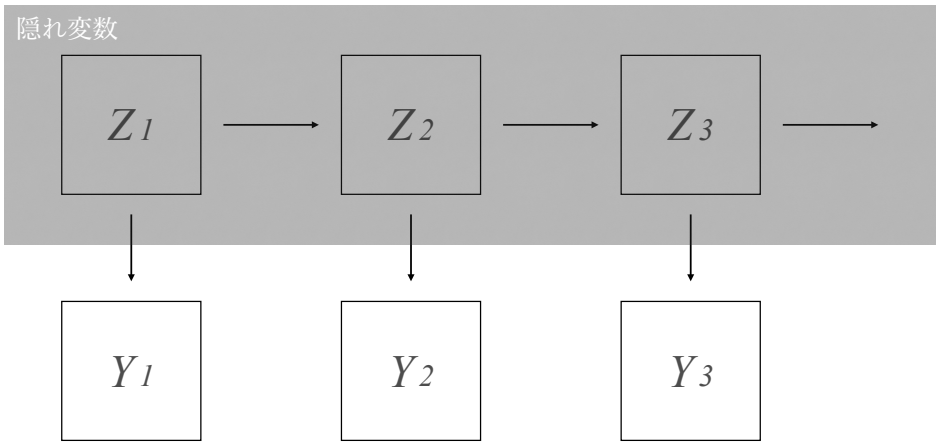

図 2.1: 隠れマルコフモデル

2.4.2 シソーラス解析

シソーラスとは単語を意味レベルで分解し，抽象度の高いものから低いものへと遡っていくことができ，それを体系づけた類語辞書のことである．シソーラスには様々な言語において有名な辞書が存在する．有名なシソーラスとしては Princeton University の WordNet がある．[15]

2.4.2.1 Wordnet

WordNet は英単語が synset と呼ばれる同義語のグループに分類され，簡単な定義や，他の同義語のグループとの関係が記述されているデータベースである．WordNet のデータベースは約 11 万 5000 の synset に分類された約 15 万語を収録し，全体で 20 万 3000 の単語と意味の組み合わせがある．[16]

2.4.3 ベクトル空間解析

単語の意味を表現するため，単語の文章での出現回数や，その単語の周辺の単語をマトリクス上に表現することで，その単語を数学的に解釈できるようにしている．

2.4.3.1 ベクトル空間モデル

ベクトル空間モデルとは文章を多次元空間上にベクトルとして表現し，それぞれのベクトルの比較を行うことで類似度を算出するためのモデルである．文章の類似度が高いほどベクトルの方向が近いということなので，比較した文章のベクトルのなす角が小さければ文章の類似度が高いということになる．
m 個の単語が使用されている文章 d における各単語の重要度を $w_{d1}, w_{d2}, w_{d3}, \ldots, w_{dm}$ とすると，文章 d のベクトルは以下のように表される．

$$\vec{d} = (w_{d1}, w_{d2}, w_{d3}, \ldots, w_{dm})$$

また，同様にして n 個の単語が使用されている文章 e をベクトル表現すると，

$$\vec{e} = (w_{e1}, w_{e2}, w_{e3}, \ldots, w_{en})$$

と表すことができる．したがって，$\vec{d}$ と $\vec{e}$ のなす角 θ における $\cos\theta$ は以下のように表される．

$$\cos\theta = \frac{\vec{d} \cdot \vec{e}}{|\vec{d}||\vec{e}|}$$

ベクトル化した時の単語の重要度は TF-IDF のアルゴリズム (※ 2.4.3.1.1) を用いて算出した重みを用いることで，これを表すことができる．[17] 上記の例であれば，文章 d における単語の重要度が $tf(t_1,d)\cdot idf(t_1), tf(t_2,d)\cdot idf(t_2), tf(t_3,d)\cdot idf(t_3), \ldots, tf(t_m,d)\cdot idf(t_m)$ であるので，文章 d のベクトルは以下のように表される．

$$\vec{d} = (tf(t_1,d)\cdot idf(t_1), tf(t_2,d)\cdot idf(t_2), tf(t_3,d)\cdot idf(t_3), \ldots, tf(t_m,d)\cdot idf(t_m))$$

また，同様にして文章 e もベクトル表現すると，

$$\vec{e} = (tf(t_1,e)\cdot idf(t_1), tf(t_2,e)\cdot idf(t_2), tf(t_3,e)\cdot idf(t_3), \ldots, tf(t_n,e)\cdot idf(t_n))$$

と表すことができ，これを $\cos\theta = \frac{\vec{d}\cdot\vec{e}}{|\vec{d}||\vec{e}|}$ に代入すると (※$m \leqq n$)，

$$
\begin{aligned}
\cos\theta &= \frac{\vec{d}\cdot\vec{e}}{|\vec{d}||\vec{e}|} \\
&= ((tf(t_1,d)\cdot idf(t_1))(tf(t_1,e)\cdot idf(t_1)) + (tf(t_1,d)\cdot idf(t_2))(tf(t_2,e)\cdot idf(t_2)) + \ldots \\
&+ (tf(t_m,d)\cdot idf(t_m))(tf(t_m,e)\cdot idf(t_m)))\cdot \\
&\frac{1}{\sqrt{(tf(t_1,d)\cdot idf(t_1))^2 + (tf(t_2,d)\cdot idf(t_2))^2 + \ldots + (tf(t_m,d)\cdot idf(t_m))^2}}\cdot \\
&\frac{1}{\sqrt{(tf(t_1,e)\cdot idf(t_1))^2 + (tf(t_2,e)\cdot idf(t_2))^2 + \ldots + (tf(t_n,e)\cdot idf(t_n))^2}} \\
&= \frac{\sum_{i=1}^{m}((tf(t_i,d)\cdot idf(t_i))(tf(t_i,e)\cdot idf(t_i))}{\sum_{i=1}^{m}\sqrt{(tf(t_i,d)\cdot idf(t_i))^2}\sum_{i=1}^{n}\sqrt{(tf(t_i,e)\cdot idf(t_i))^2}}
\end{aligned}
\tag{2.1}
$$

と表すことができる．これがベクトル空間で TF-IDF で抽出した単語の重み付けを行い，二つの文章の類似度を算出するモデルである．

2.4.3.1.1 TF-IDF

TF とは Term Frequency のことで，文章内での単語の出現頻度を表す．数式では以下のように表される．

$$tf(t,\,d) = \frac{n_{t,\,d}}{\sum_{s\ni d} n_{s,\,d}}$$

$tf(t,\,d)$ は TF の値で，文章 d 内に含まれる単語 t の出現頻度を表す．
$n_{t,\,d}$ は文章 d における単語 t の出現回数を表す．
$\sum_{s\ni d} n_{s,\,d}$ は文章 d における全ての単語の出現回数を表す．
以上を踏まえ TF の値とは，

$$\text{文章 d 内に含まれる単語 t の出現頻度} = \frac{\text{文章 d における単語 t の出現回数}}{\text{文章 d における全ての単語の出現回数}}$$

を数式で表したものである．
IDF とは Inverse Document Frequency のことで，ある単語が様々な文章においてどれほど使われているのかを表す．数式では以下のように表される．

$$idf(t) = \log\frac{N}{df(t)} + 1$$

$idf(t)$ は IDF の値で，単語 t が全文章数 N でどれほど使われているのかを表す．
N は全文章数を表す．

$df(t)$ は単語 t が出現する文章の数を表す.
以上を踏まえ IDF の値とは,

$$\text{単語 t が全文章数 N でどれほど使われているのか} = \frac{\text{全文章数}}{\text{単語 t が出現する文章の数} + 1}$$

を数式で表したものである.
このような TF の値と IDF の値を重みとすることで，文章を特徴付ける単語の抽出をするものが TF-IDF である．上記の TF と IDF の値より，if-idf の値は

$$ifidf(t,d) = tf(t,d) \cdot idf(t)$$

から算出することができる.

2.4.3.2 word2vec

word2vec は 2 層からなるニューラルネットワークである．word2vec には 2 つのアーキテクチャがあり，一つは $ContinuousSkip-gramModel$，もう一つは $ContinuousBag-of-WordsModel$ である．$ContinuousSkip-gramModel$ は入力に文章中の任意の単語を用意し，出力に文章においてその任意の単語の前後の周辺語を用意し，ニューラルネットワークに読み込ませることで第一層から第二層への重みを獲得することが目的である．$ContinuousBag-of-WordsModel$ では逆に出力に文章中の任意の単語を用意し，入力に文章においてその任意の単語の前後の周辺語を用意し，同様にしてニューラルネットワークに読み込ませることで第一層から第二層への重みを獲得することが目的である．本研究ではより精度の高い $ContinuousSkip-gramModel$（以降，$Skip-gramModel$ と呼ぶ.）を使用した．[1]

2.4.3.3 Continuous Skip-gram Model

$Skip-gramModel$ は先述の通り，与えられた単語に対してその周辺語を予測するためのモデルのことである．このモデルは 2 層からなるニューラルネットで，入力には One-hot ベクトルを用いる．One-hot ベクトルとは $(0,0,0,\ldots,1,\ldots,0)$ のように，単語のインデックスから抽出する単語だけを 1 と表記することで表現するベクトルのことである.
入力層から隠れ層への重みは $V \times N$ のマトリクス W で表され，W の各列は単語ベクトルとなっている．隠れ層から出力層への重みはマトリクス W を転置した $N \times V$ のマトリクス W' となっている.
これをモデル化したものの出力の条件付き確率を考える.

$$p(w_O|w_I) = \frac{exp(v'^{\mathrm{T}}_{W_V} \cdot v_{w_I})}{\sum_{W_v \in V} exp(v'^{\mathrm{T}}_{W_V} \cdot v_{w_I})} \tag{2.2}$$

この w_I は入力する単語，w_O は w_I の周辺語を表す．v_{w_I} や　$v'^{\mathrm{T}}_{W_V}$ は単語を表すベクトルであり，v は入力ベクトルで v' は出力ベクトルである．コンテクストサイズとは先述したように，入力単語の周辺語をどこまでとするかのサイズのことである．$p(w_O|w_I)$ はコンテクストサイズを考慮していない確率であるが，このコンテクストサイズを考慮して先述したモデルの同時確率 $p(w_{O,1}, w_{O,2}, w_{O,3}, \ldots, w_{O,C}|w_I)$ を考える．

$$p(w_{O,1}, w_{O,2}, w_{O,3}, \ldots, w_{O,C}|w_I) = \prod_{c=1}^{C} \frac{exp(v'^{\mathrm{T}}_{W_c} \cdot v_{w_I})}{\sum_{W_v \in V} exp(v'^{\mathrm{T}}_{W_V} \cdot v_{w_I})} \tag{2.3}$$

この $p(w_{O,1}, w_{O,2}, w_{O,3}, \ldots, w_{O,C}|w_I)$ という確率を表す関数 $\prod_{c=1}^{C} \frac{exp(v'^{\mathrm{T}}_{W_V} \cdot v_{w_I})}{\sum_{W_v \in V} exp(v'^{\mathrm{T}}_{W_V} \cdot v_{w_I})}$ が最大となるような単語ベクトル v を求めることが，このモデルの目的である．
このモデルを用いてニューラルネットを構築する．先述の通り，入力層 x は One-hot ベクトルを用いる．One-hot ベクトルとは $(0, 0, 0, \ldots, 1, \ldots, 0)$ のように，単語のインデックスから抽出する単語だけを 1 と表記することで表現するベクトルのことである．
隠れ層 h は，入力層から隠れ層への重み W を入力データ x にかけたものである．したがって隠れ層 h は

$$h = Wx$$

と表すことができる．また，任意の入力 w_I $(= x_i)$ は重み W が掛けられるが，入力が One-hot ベクトルなので，w_I に対応する単語ベクトルがそのまま出力されることになる．したがって，隠れ層は

$$h = Wx_{w_I} = v_{w_I}$$

と表すことができる．
出力層 u_c は，隠れ層 h に隠れ層から出力層への重み W' が掛けられたものであるので，

$$u_c = W'h = W'v_{w_I}$$

と表すことができる．また，出力層はコンテクストサイズに応じて出力のユニット数 c が変動する．したがって，任意のユニット C における最終的な出力 $y_{c,i}$ に softmax 関数を掛けて，

$$\begin{aligned} y_{c,i} &= \frac{exp(u_{c,i})}{\sum_{v=1}^{V} exp(u_{c,v})} \\ &= \frac{exp(v'_i \cdot v_{w_I})}{\sum_{v=1}^{V} exp(v'_v \cdot v_{w_I})} \\ &= p(w_i|w_I) \end{aligned}$$

と表され，式 (2.2) で表した確率と同じになることが確認できる．
したがって式 (2.3) で表された同時確率 $p(w_1, w_2, w_3, \ldots, w_C|w_I)$ の最大化をするために，

単語ベクトル v と単語ベクトル v' を最適化する．すなわち重み W と重み W' を最適化することを考える．word2vec では最適化のために確率的勾配降下法を用いており，目的関数として以下の式 (2.4) を定める．

$$E = -\log p(w_1, w_2, w_3, \ldots, w_C|w_I) \tag{2.4}$$

後述の損失関数の導出を円滑にするため，最大化問題から最小化問題へするために負の符号を付し，また同時確率であることから確率の値が極端に小さくなる可能性を考慮し，対数を取ることで乗法から和法へと変換することでアンダーフローを防いだ．
式 (2.4) に式 (2.3) を代入すると，

$$\begin{aligned}
E &= -\log p(w_1, w_2, w_3, \ldots, w_C|w_I) \\
&= -\log \prod_{c=1}^{C} \frac{exp(u_{C,w_C})}{exp(\sum_{v=1}^{V} exp(u_{C,V})} \\
&= -\sum_{C=1}^{C} \log \frac{exp(u_{C,w_C})}{exp(\sum_{v=1}^{V} exp(u_{C,V})} \quad (\because \log_a MN = \log_a M + \log_a N) \\
&= -\sum_{C=1}^{C} (\log exp(u_{C,w_C}) - \log \sum_{V=1}^{V} exp(u_{C,V})) \quad (\because \log_a \frac{M}{N} = \log_a M - \log_a N) \\
&= -\sum_{C=1}^{C} (u_{C,w_C} - \log \sum_{V=1}^{V} exp(u_{C,V})) \quad (\because \log_e exp(x) = x)
\end{aligned} \tag{2.5}$$

となる．この式 (2.5) を重み W と重み W' で偏微分し，誤差を求めることを考える．
まずは重み W' で E を偏微分し，W' の更新式を得る．また，以下の図 2.2 は，入力ベクトルが出力ベクトルのどこに含まれているのかを表したものである．

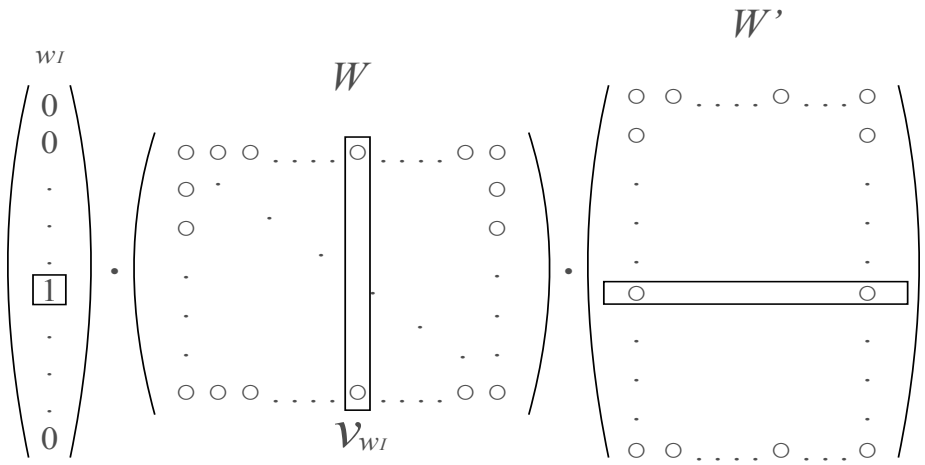

図 2.2: 入力ベクトルが出力ベクトルのどこに含まれているか

つまり，v'_{ij} は u_C の内，i 番目の要素 $u_{c,i}$ に含まれていると言える．コンテクストサイズは C なので，C 個の多変数関数であり，連鎖律を用いると，

$$\frac{\partial E}{\partial v'_{ij}} = \sum_{C=1}^{C} \frac{\partial E}{\partial u_{c,i}} \frac{\partial u_{c,i}}{\partial v'_{ij}} \tag{2.6}$$

とすることができる．式 (2.6) の右辺より，E を $u_{c,i}$ でまずは微分することを考える．式 (2.5) を変形して，

$$\begin{aligned} E &= -\sum_{C=1}^{C}(u_{C,w_C} - \log \sum_{V=1}^{V} exp(u_{C,V})) \\ &= -\sum_{C=1}^{C} u_{C,w_C} + \sum_{C=1}^{C} \log \sum_{V=1}^{V} exp(u_{C,V}) \end{aligned} \tag{2.7}$$

式 (2.7) の内，$\sum_{C=1}^{C} u_{C,w_C}$ を $u_{c,i}$ で微分することを考える．

$$\sum_{C=1}^{C} u_{C,w_C} = u_{1,w_1} + u_{2,w_2} + \ldots + u_{C,w_C} \tag{2.8}$$

と変換することで，

$$\frac{\partial \sum_{C=1}^{C} u_{C,w_C}}{\partial u_{c,i}} := t_{c,i} \left\{ \begin{array}{ccc} & 1 & \\ (i = wc) & 0 & otherwise. \end{array} \right\} \tag{2.9}$$

とすることができる．次に $\sum_{C=1}^{C} \log \sum_{V=1}^{V} exp(u_{C,V})$ の部分について，

$$
\begin{aligned}
g = \sum_{V=1}^{V} exp(u_{C,V}) f = \log g \frac{df}{dg} \frac{dg}{du_{c,i}} &= \frac{1}{g} exp(u_{c,i}) \\
&= \frac{exp(u_{c,i})}{exp(\sum_{v=1}^{V} exp(u_{c,v})} \\
&= y_{c,i} \qquad (2.10)
\end{aligned}
$$

式 (2.9) と式 (2.10) より，

$$
\frac{\partial E}{\partial u_{c,i}} = -t_{c,i} + y_{c,i} \qquad (2.11)
$$

さらに，式 (2.6) の $\frac{\partial u_{c,i}}{\partial v'_{ij}}$ の部分は，

$$
\frac{\partial u_{c,i}}{\partial v'_{ij}} = v_{w_{Ij}} \qquad (2.12)
$$

$$
(\because u_{c,i} = v'_{i1} v_{w_I} 1 + v'_{i1} v_{w_I} 1 + \ldots + v'_{ij} v_{w_I} j + \ldots + v'_{iN} v_{w_I} N) \qquad (2.13)
$$

したがって，式 (2.6) と式 (2.11) と式 (2.12) より，

$$
\frac{\partial E}{\partial v'_{ij}} = \sum_{C=1}^{C} (y_{c,i} - t_{c,i}) v_{w_{Ij}} \qquad (2.14)
$$

が得られる．

以上をまとめると，Skip-gram Model のニューラルネットワークは以下の図 2.3 のようになる．

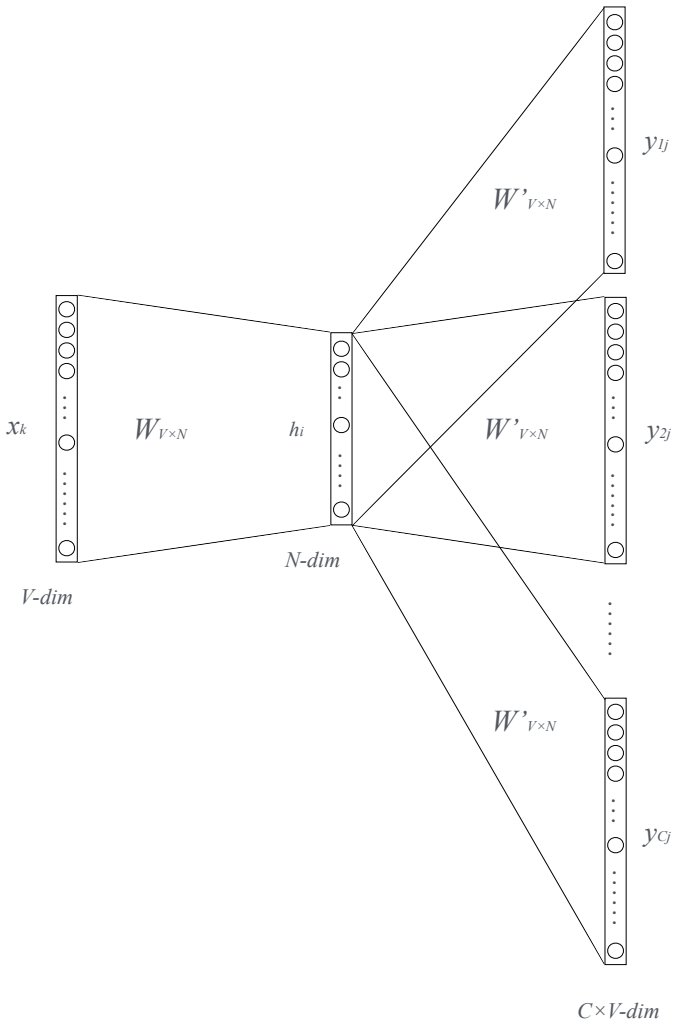

図 2.3: コンテクストサイズ C の Skip-gram Model のニューラルネットワーク

2.4.3.4 SCDV

SCDV とは word2vec で算出したベクトル空間をクラスタリングして，クラスタリング結果も考慮に入れてベクトル空間上に再表現する手法のことである．以下は公式ドキュメントである．まず，ある単語 wiwi のベクトル wviwvi を取得 (word2vec)．ある単語 wiwi の IDF 値 idf(wi)idf(wi) を計算する．GMM で全単語ベクトルについてクラスタリングを行い，ある単語が各クラスタ ckck に属する確率 P(ck—wi)P(ck—wi) を取得する．語彙空間における各単語 wiwi について 5 を繰り返す．各クラスタ ckck についてクラスタの数だけ (*) を繰り返す．クラスタの数だけ計算したら，(**) を実行する．(*）クラスタを考

慮した新たな単語ベクトル wcvikwcvik を wvi × P(ck—wi)wvi × P(ck—wi) として算出．(**) IDF を考慮した新たな単語ベクトル wtviwtvi を idf(wi) × Kk wcvikidf(wi) × kK wcvik で算出 (は concatenation)．各ドキュメント (n から Nn から N) について 9 の操作を繰り返す．ドキュメント DnDn についてベクトルを初期化し，DnDn に含まれる単語 wiinDnwiinDn についてベクトルを足し合わせていき平均する．最後に得られた文書ベクトル DnDn をスパースにして SCDVDnSCDVDn を得る．[18]

第3章　本研究における問題定義と仮説

本章では，１章で述べた背景より，本章では，現状の Honeypot の問題点を整理し，この問題をどのように解決すれば良いのかを定義する．

3.1　本研究における問題定義

現状の Honeypot の問題点を列挙し，整理する．

3.1.1　SSH Honeypot の現状の問題

Honeypot には運用する上で大きな問題が 2 つある．一つは設置した Honeypot に侵入した悪意のある侵入者が侵入先を Honeypot であると検知してしまう問題である．もう一つは Honeypot に侵入を許した侵入者に Honeypot を設置した機器から他のホストに攻撃を仕掛けられてしまう，所謂踏み台攻撃の踏み台にされる問題である．
以下の図 3.1 は，悪意のある侵入者が不正に機器に侵入してから踏み台にして他の機器に攻撃を仕掛けるまでの一般的なフロ–である．問題として，2 番目のフロ–の悪意のある侵入者が侵入した先が Honeypot であると検知してしまうことが考えられる．高対話型 Honeypot で使用した OS 自体の新たな脆弱性を突かれることに限った状況で，3 番目のフロ–の Honeypot への侵入者に Honeypot を設置した機器から攻撃が仕掛けられてしまう危険があることが問題として挙げられる．

本研究ではこの中でも，2 番目のフロ–の SSH の低対話型 Honeypot が設置した Honeypot に悪意のある侵入者が侵入先を Honeypot であると検知してしまう問題に着目した．

3.1.1.1　SSH の低対話型 Honeypot における問題

SSH の低対話型 Honeypot は実際の Shell の挙動をエミュレートしたものであるのでコマンドやその挙動についての機能が限定されており，実際の Shell の機能として不足がある．また SSH の低対話型 Honeypot 特有の以上な挙動も存在する．さらに，Honeypot の username のデフォルトが決まっているため，username から Honeypot であることを検知されてしまう問題もある．これらの検知手法を用いて，侵入者に侵入先が Honeypot であると検知され，本来取れるはずの攻撃ログが収集できない問題がある．

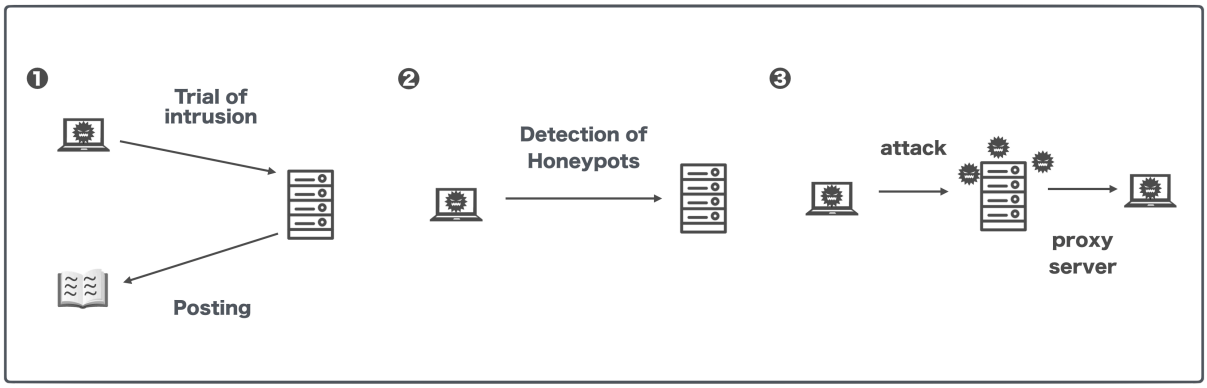

図 3.1: 不正な SSH 侵入者の想定行動フロー

3.1.1.1.1 Honeypot の Username の問題

SSH の低対話型 Honeypot の内、特に Cowrie は kippo を改良したものであるため，kippo のデフォルトの username である”Richard”が cowrie の username となっている．このため，username が”Richard”で，honeypot 特有の挙動をした場合に，侵入者に Honeypot であると検知されてしまう．

3.1.1.1.2 Honeypot のコマンドの実装の問題

SSH の低対話型 Honeypot は実際の Shell の挙動をエミュレートしたものであるのでコマンドやその挙動についての機能が限定されており，実際の Shell の機能として不足がある． 2.2.2 で述べたように，”Linux 上で最小の実行ファイル”となるよう設計されている BusyBox に含まれるコマンドの数が 200 以上あるのに対し，現状で広く使われている SSH の低対話型 Honeypot である Cowrie に実装されているコマンドは 2.1.1.2 でも述べた通り，92 しか存在しない．また，SSH の低対話型 Honeypot 特有の挙動が存在し，以下にその 1 例であるプログラム 3.1 とプログラム 3.2 を示す．

プログラム 3.1: 正しい Shell の挙動

```
1 nadechin@cpu:~$ echo -n test
2 testnadechin@cpu:~$
```

プログラム 3.2: Kippo 特有の異常な挙動の例

```
1 root@localhost-neco:~$ echo -n hello
2 -n hello
3 root@localhost-neco:~$
```

プログラム 3．1 が通常の挙動でプログラム 3．2 が SSH の低対話型 Honeypot の挙動である．echo コマンドの-n オプションは改行出力末尾にしないようにするものである．しかし，実際の Shell の出力は改行がされない一方，Honeypot の挙動ではオプション部分

も出力されてしまい，末尾も改行されてしまう．これは SSH の低対話型 Honeypot 特有の挙動であるため，この挙動を観測することによって侵入者に侵入先が Honeypot であると検知されてしまう可能性がある．

3.1.2 本研究の問題

3.1.1.1 で列挙した SSH の低対話型 Honeypot の問題の中で，実際の Shell に実装されているコマンドの不足がある．また SSH の低対話型 Honeypot に特有の異常な挙動も存在するため，設置した Honeypot が悪意のある侵入者に侵入先を Honeypot であると検知されてしまい，実際の OS に悪意のある侵入者が侵入した時の侵入ログとの違いが大きく出てしまう問題に着目した．

3.2 問題解決のための要点

3.1.2 で着目した問題を解決するためには，以下 2 つの手法を取る必要がある．

コマンドの追加実装: 実際の Shell に実装されているコマンドで，SSH の低対話型 Honeypot に実装されていないコマンドを実装する

既実装コマンドの修正: SSH の低対話型 Honeypot に特有の異常な挙動をする既実装コマンドを修正する

3.3 仮説

3.3.1 と 3.3.2 で示す問題解決のための要点を踏まえると，SSH の低対話型 Honeypot に侵入した悪意のある侵入者に侵入先を Honeypot であると検知させず，SSH の低対話型 Honeypot に悪意のある侵入者が侵入した時の侵入ログを，実際の OS に悪意のある侵入者が侵入した時の侵入ログに近似できるのでないかと考えた．

3.3.1 コマンドの追加実装

実際の Shell に実装されているコマンドで，SSH の低対話型 Honeypot に実装されていないコマンドを実装することで，Honeypot への侵入者が実行できるコマンドの少なさによる，Honeypot であることの検知を回避することができる．

3.3.2 既実装コマンドの修正

SSH の低対話型 Honeypot に特有の異常な挙動をする既実装コマンドを修正することで，Honeypot について認知している侵入者が，侵入先を Honeypot であると検知することを回避することができる．

第4章　本研究の手法

本章では， 3.3 節で述べた仮説を検証するために，本研究の手法について概説する．

4.1　問題解決の為のアプローチ

3.2 で述べた問題解決のための 2 つの要件を満たすために，本研究では低対話型 Honeypot に実装されていないコマンドを実装し，さらに低対話型 Honeypot の既実装コマンドで，低対話型 Honeypot に特有の異常な挙動をするコマンドの修正を行う．略図を図 4.1 に示す．

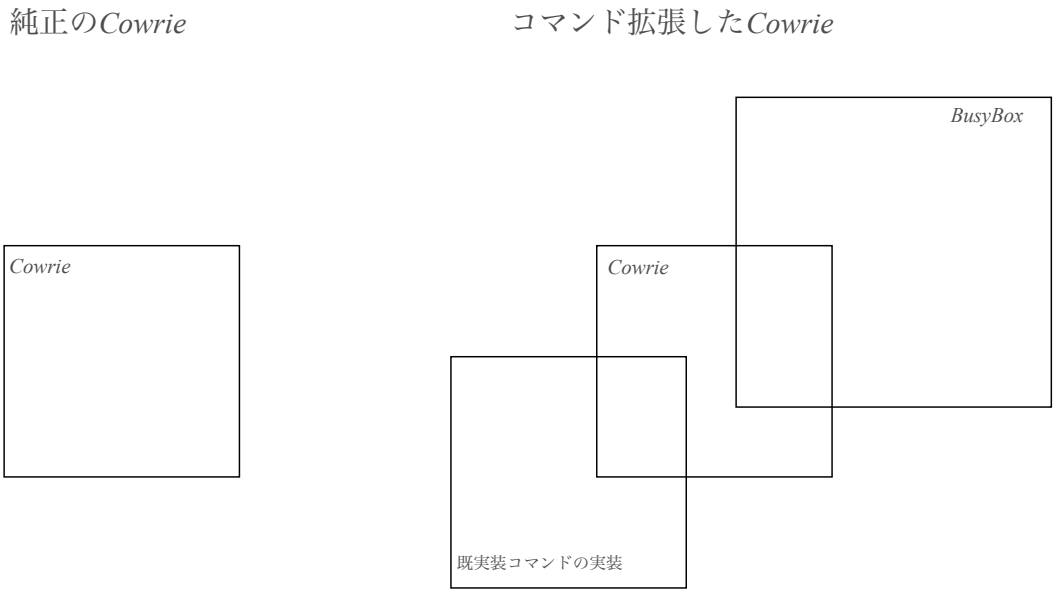

図 4.1: 低対話型 Honeypot の拡張

第5章　実装

本章では，低対話型 Honeypot と高対話型 Honeypot の設置環境についても示し， 4.1 節で述べた手法を用いて純正の Honeypot にどのようなコマンドを実装し，Honeypot 特有の異常な挙動を修正したのかを説明する．

5.1　実装環境

本研究で実装するシステムを構成するためのハードウェアおよびソフトウェアについて説明する．表 5.1 に詳細なバージョンを示す．

表 5.1: 実装環境

ハードウェア/ソフトウェア	実装環境	Version(date)
純正の Cowrie	CentOS6	1．4．0
修正済みの Cowrie	CentOS6	1．4．0（self made）
Honeywall	CentOS6	1．4

5.1.1　純正の Honeypot で未実装のコマンド種類

本研究において純正の Honeypot は Cowrie[19] を使用し，実際の Shell には実装されているが，純正の Honeypot で未実装のコマンドについては BusyBox[13] に含まれるコマンドの実装を行なった． 2.2.2 や 2.1.1.2 で紹介した通り，BusyBox に含まれるコマンドの種類が 219 ある中で，Cowrie の実装コマンド数は 92 しか存在しない．この差分を Python で実装する．
また BusyBox に含まれるコマンドと Cowrie の実装コマンド，Cowrie に未実装のコマンドの一覧は付録 A の表 A.1 に示しておく．

5.1.2 純正の Honeypot で未実装のコマンドの実装

5.1.1 で示した Cowrie に未実装のコマンドについての一部の実装を示す．他の実装は A.1.2 の表に示す．Cowrie に実装されているコマンドは cowrie/src/cowrie/commands/ に格納されており，ここに追加コマンドを実装する．

以下の 5.1 に cowrie/src/cowrie/commands/base.py に dmidecode コマンドを追加実装したものを示す．[20]

プログラム 5.1: dmidecode

```
class command_dmidecode(HoneyPotCommand):

    def call(self):

        self.write(b# dmidecode 3.1
Getting SMBIOS data from sysfs.
SMBIOS 3.0.0 present.
Table at 0xDDBA4000.

(snip)

Handle 0x0055, DMI type 13, 22 bytes
BIOS Language Information
    Language Description Format: Long
    Installable Languages: 1
        en|US|iso8859-1
    Currently Installed Language: en|US|iso8859-1

Handle 0x0056, DMI type 127, 4 bytes
End Of Table\n)
commands['dmidecode'] = command_dmidecode
```

追加実装以外ではファイルを追加することで，コマンドを追加したものを以下の 5.2 に示す．[21]

プログラム 5.2: diff コマンド

```
# coding: utf-8
import sys

class Diff:
  def __init__(self, a, b):
    self.a = a
    self.b = b
```

```
8
9   def solve(self):
10    self.result = None
11    self.createTable()
12    self.selectMatches()
13    self.connectPath()
14    self.genResult()
15    return self.result
16
17  def createTable(self):
18    self.table = [[x == y for y in self.b] for x in self.a]
19
20  def selectMatches(self):
21    self.matches = [ (i, j)
22      for i in range(len(self.table))
23      for j in range(len(self.table[i]))
24      if self.table[i][j]]
25
26    self.max_path = []
27    self.path = []
28
29    self.search((-1, -1)) # 左上から探索
30    # print self.max_path
31
32  def search(self, pos):
33    def isReachable(pos, match):
34      return match[0] > pos[0] and match[1] > pos[1]
35
36    if pos != (-1, -1): self.path.append(pos)
37    is_term = True
38    for match in self.matches:
39      if isReachable(pos, match):
40        is_term = False
41        next_pos = match
42        self.search(next_pos)
43
44    if is_term: # 終端の場合
45      if len(self.path) > len(self.max_path):
46        self.max_path = list(self.path) # 最良経路の更新
47    if pos != (-1, -1): self.path.pop()
48
49  def connectPath(self):
50    # self.path = [(0, 0)]
51    self.path = []
52    prev = (0, 0)
53    for pos in self.max_path:
54      p = prev
```

```
55        while p[0] < pos[0]:
56          p = (p[0] + 1, p[1])
57          self.path.append(p)
58        while p[1] < pos[1]:
59          p = (p[0], p[1] + 1)
60          self.path.append(p)
61        p = (p[0] + 1, p[1] + 1)
62        self.path.append(p)
63        prev = p
64      p = prev
65      while p[0] < len(self.table):
66        p = (p[0] + 1, p[1])
67        self.path.append(p)
68      while p[1] < len(self.table[0]):
69        p = (p[0], p[1] + 1)
70        self.path.append(p)
71      # print self.path
72
73    def genResult(self):
74      prev = (0, 0)
75      i_a = 0
76      i_b = 0
77      self.result = []
78      for pos in self.path:
79        if pos == (0, 0): continue
80        if pos[1] == prev[1]: # 縦への移動 : a の -
81          self.result.append((self.a[i_a], '-'))
82          i_a += 1
83        elif pos[0] == prev[0]: # 横への移動 : b の +
84          self.result.append((self.b[i_b], '+'))
85          i_b += 1
86        else: # 斜めへの移動
87          self.result.append((self.a[i_a], ' '))
88          i_a += 1
89          i_b += 1
90        prev = pos
91        # print self.result
92
93    def printResult(self):
94      for line, sign in self.result:
95        print sign, line,
96
97  def diff(filename_a, filename_b):
98    a = []
99    with open(filename_a) as f:
100     for line in f:
101       a.append(line)
```

```
102
103   b = []
104   with open(filename_b) as f:
105     for line in f:
106       b.append(line)
107
108   # a = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
109   # b = ['a', 'b', 'x', 'c', 'y', 'e', 'g']
110   solver = Diff(a, b)
111   result = solver.solve()
112   solver.printResult()
113
114 if __name__ == "__main__":
115   if len(sys.argv) != 3:
116     print "Usage: python %s fileA fileB" % sys.argv[0]
117     quit()
118   diff(sys.argv[1], sys.argv[2])
```

さらに，コマンド実行時に出力結果を print 関数で出力するのみのコマンドを追加したものの例を以下の 5.3 示す．本例では mount コマンドを以下のように実装した．本研究の実装はこちらがほとんどとなっている．

プログラム 5.3: mount コマンド

```
1  # coding: utf-8
2
3  string = '''sysfs on /sys type sysfs (rw,nosuid,nodev,
       noexec,relatime)
4  proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
5  udev on /dev type devtmpfs (rw,nosuid,relatime,size
       =16459820k,nr_inodes=4114955,mode=755)
6  devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,
       gid=5,mode=620,ptmxmode=000)
7  tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size
       =3294084k,mode=755)
8  /dev/sda1 on / type ext4 (rw,relatime,discard,data=ordered)
9  securityfs on /sys/kernel/security type securityfs (rw,
       nosuid,nodev,noexec,relatime)
10 tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
11 tmpfs on /run/lock type tmpfs (rw,nosuid,nodev,noexec,
       relatime,size=5120k)
12 tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,
       mode=755)
13 cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,
       nodev,noexec,relatime,xattr,release_agent=/lib/systemd/
       systemd-cgroups-agent,name=systemd)
```

```
14 pstore on /sys/fs/pstore type pstore (rw,nosuid,nodev,
     noexec,relatime)
15 cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,
     nosuid,nodev,noexec,relatime,net_cls,net_prio)
16 cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,
     nodev,noexec,relatime,freezer)
17 cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,
     noexec,relatime,pids)
18 cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,
     nodev,noexec,relatime,cpuset)
19 cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid
     ,nodev,noexec,relatime,cpu,cpuacct)
20 cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,
     nodev,noexec,relatime,memory)
21 cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,
     nodev,noexec,relatime,devices)
22 cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,
     nodev,noexec,relatime,hugetlb)
23 cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,
     noexec,relatime,rdma)
24 cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev
     ,noexec,relatime,blkio)
25 cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,
     nodev,noexec,relatime,perf_event)
26 systemd-1 on /proc/sys/fs/binfmt_misc type autofs (rw,
     relatime,fd=26,pgrp=1,timeout=0,minproto=5,maxproto=5,
     direct,pipe_ino=11662)
27 debugfs on /sys/kernel/debug type debugfs (rw,relatime)
28 mqueue on /dev/mqueue type mqueue (rw,relatime)
29 hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,
     pagesize=2M)
30 configfs on /sys/kernel/config type configfs (rw,relatime)
31 fusectl on /sys/fs/fuse/connections type fusectl (rw,
     relatime)
32 /dev/sdb1 on /mnt type ext4 (rw,relatime,data=ordered)
33 lxcfs on /var/lib/lxcfs type fuse.lxcfs (rw,nosuid,nodev,
     relatime,user_id=0,group_id=0,allow_other)
34 tmpfs on /run/user/1000 type tmpfs (rw,nosuid,nodev,
     relatime,size=3294084k,mode=700,uid=1000,gid=1000)'''
35
36 print(string)
```

print 関数で出力するのみのコマンドを追加の手法として python のライブラリである subprocess を用いて標準入出力を行う．以下の本例 5.4 では ar コマンドを示す．

プログラム 5.4: ar

```
1 # coding: utf-8
2
3 import subprocess
4 import sys
5
6 cmd = "ar␣t"
7 opt = sys.argv[1]
8 cmdall = cmd + "␣" + opt
9 subprocess.call(cmdall.split())
```

第6章　評価と考察

本章では，予備実験と5章で実装した本研究での提案手法の評価とその考察を述べる．

6.1　予備実験

SSHの低対話型HoneypotであるCowrieはコマンドの実装数が少なく，Cowrie特有の異常な挙動が多い．そのため，侵入者にHoneypotであると検知されることで，本来実際のOSへの攻撃であれば取れるはずであった侵入ログが取れない問題がある．また，収集ログを分析する際に，これまで用いられてきた”危険なコマンド”としてインデックスを作り，それらを危険なコマンドとしてパターンマッチングする手法では，今後出現してくる様々なコマンドパターンなどに対応できない．
予備実験では，Cowrieはコマンドの実装数が少なく，Cowrie特有の異常な挙動が多いため，コマンドの追加実装を行い，Cowrie特有の異常な挙動を修正した．実装を施していない純正のCowrieとCowrieにBusyBoxに含まれるコマンドを実装した修正済のCowrieの両方でコマンドログの収集を行い，比較することで，収集ログのパターンの変化を観測できるのではないかと考えた．評価として収集した二つのログをSkip-gramモデルを用いてスコアリングし，どちらがより多くのコマンドログのパターンを収集できているのかを検証した．
その結果，より多くのコマンドパターンを取れたのがCowrieにBusyBoxに含まれるコマンドを実装した修正済のCowrieであるということが明らかとなった．

6.1.1　予備実験の手法

実装を施していない純正のCowrieに対して，これには実装されていないがShellには実装されているコマンドを実装したものを設置し，ログを収集した．

6.1.2　侵入ログの収集環境

6.1.3　実装

純正のCowrieにBusyBoxに含まれるコマンドを実装し，またHoneypot特有の異常な挙動を修正した．ここでの実装を紹介する．

6.1.4 評価

実装を施していない純正の Cowrie と Cowrie に BusyBox に含まれるコマンドを実装した修正済の Cowrie の両方で侵入ログの収集を行い，Word2vec の Skip-Gram Model により次のコマンドの予測，スコアリングを行い比較を行うことで，差異を評価した．スコアリングでは，あるコマンドが実行された時に次のコマンドの出やすさを予測したため，次に実行されるコマンドがスコアとして高い数値を出せばそのコマンドパターンがパターンとして存在しやすいものであることを示す．予備実験の評価に関しては第 7 章の評価で一部説明している．本研究の評価と評価手法の違いは，モデル化を純正の Honeypot に BusyBox に含まれるコマンドを実装したものしか行っていないため，実際の OS に近いログが取れたことが証明できておらず，比較する対象が少なかった．

6.1.5 結果

SSH の低対話型 Honeypot の稼働期間は 12/10 から 2/1(54 日間) で，収集できたものとしてコネクション数，パターン数，コマンド数を以下の表 6.1 に記す．

表 6.1: 予備実験の結果

検知したもの	純正の Honeypot	修正済の Honeypot
コネクション数	19829	27914
パターン数	53	91
コマンド数	470	841

また，モデル化を行い純正の Cowrie と Cowrie に BusyBox に含まれるコマンドを実装した修正済の Cowrie のスコアリングを行なった結果を以下の図 6.1 に記す．横軸はコマンド拡張を行なった Honeypot か素の Honeypot であるかを示している．縦軸は予備実験の評価手法によって算出されたコマンドの表れやすさを数値化したものであり，数値が高くなればのコマンドパターンがパターンとして存在しやすいものであることを示している．予備実験ではコマンドの拡張実装をしたものの方がコマンドパターンとして存在しにくいものを観測できたという結果になった．

本研究の予備実験では，Cowrie に実装されていないコマンドで悪意のある侵入者が使うようなコマンドを実装し，何の追加実装も施していない Cowrie で取れた侵入者の実行コマンドログと，追加実装を施した Cowrie の侵入者の実行コマンドログを比較することで，追加実装を施した SSH の Cowrie の方がコマンドパターンとして多く収集できることを示した．

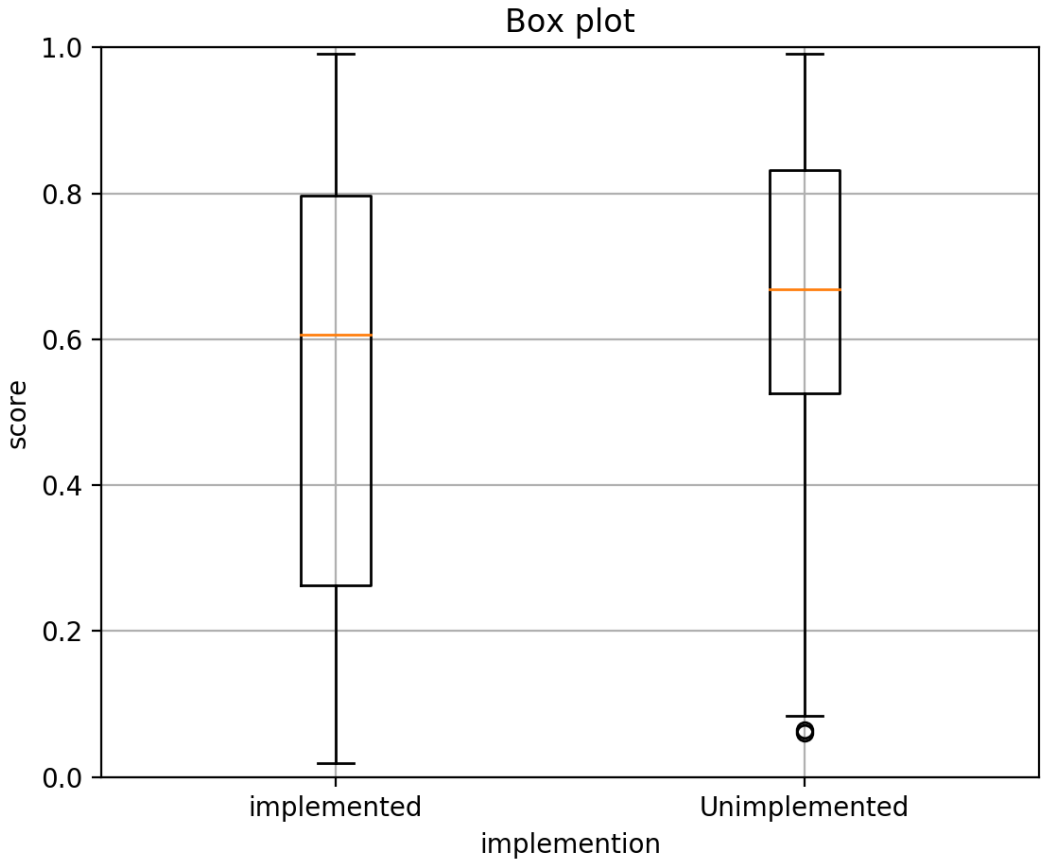

図 6.1: 純正の Cowrie と修正済の Cowrie のスコアリングによる比較

6.2 評価手法

本研究の仮説の検証手法としての評価として，3.2 節で述べた要件に対して評価を行う．予備実験では，素の低対話型 Honeypot よりも，コマンドを拡張した Honeypot の方がコマンドパターンが多く収集できることを示した．本研究では拡張した Honeypot で収集したコマンドログが，どれほど一般的な UNIX ユーザーの実行するコマンド [22] から離れたのかを評価した．

本研究では，以下の三種類の Honeypot を設置する．

1. 広く利用されている SSH の低対話型 Honeypot
2. 実際の Shell には実装されているが，1. の Honeypot で未実装のコマンドを実装した Honeypot
3. 広く利用されている高対話型 Honeypot

本研究の評価として上記の 3 つの Honeypot でのコマンドログの収集したが，高対話型 Honeypot のコマンドログの収集数が極端に少なかったため，今回の評価では純正の低対話型 Honeypot と，修正済の低対話型 Honeypot を比較する形での評価を行なった．高対話型 Honeypot ののコマンドログの収集量が少なかった考察については 6.3 で述べる．

これ以降，1．の広く利用されている SSH の低対話型 Honeypot のことを ”純正の低対話型 Honeypot ”， 2．の実際の Shell には実装されているが，1．の Honeypot で未実装のコマンドを実装した Honeypot のことを”修正済の低対話型 Honeypot”， 3．の広く利用されている高対話型 Honeypot のことを”高対話型 Honeypot”と呼ぶこととする．

また予備実験では，純正の Honeypot に実装されていないコマンドで悪意のある侵入者が使うようなコマンドを実装し，純正の Honeypot で取れた侵入者の実行コマンドログと，修正済の Honeypot の侵入者の実行コマンドログを比較することで，修正済の Honeypot の方がコマンドパターンとして多く収集できることを示した．予備実験における収集ログの比較の概念図を図 6.2 に示す．

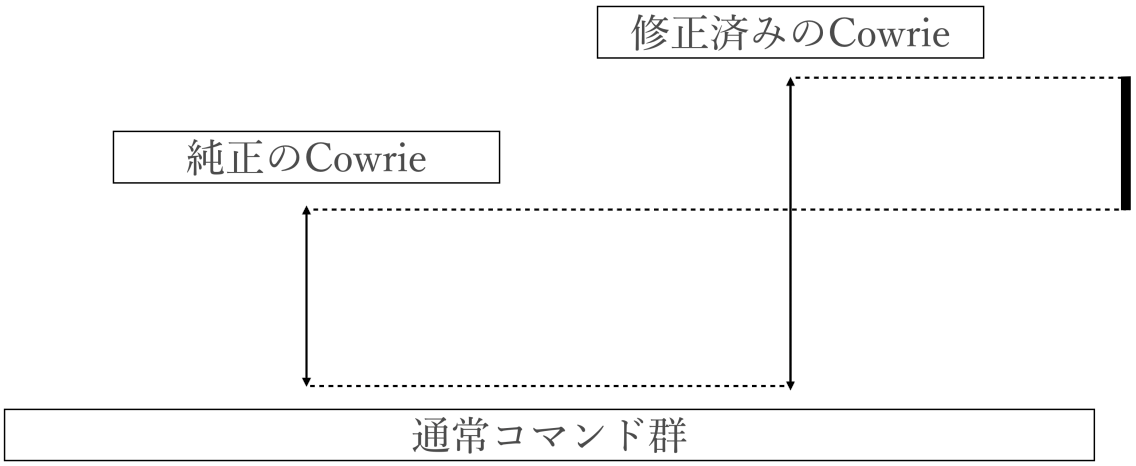

図 6.2: 予備実験の評価の概念図

この予備実験では評価として何の追加実装も施していない SSH の低対話型 Honeypot で取れた侵入者の実行コマンドログと追加実装を施した SSH の低対話型 Honeypot の侵入者の実行コマンドログとを比較したのに対して，本件研究の評価手法では，純正の Honeypot で取れた侵入者の実行コマンドログと修正済の Honeypot の侵入者の実行コマンドログと高対話型 Honeypot の侵入者の実行コマンドログを比較することで，修正済の Honeypot の侵入者の実行コマンドログが一般的な UNIX ユーザーの実行するコマンドから離れたのかを評価した．

6.2.1 コマンドログのスコアリング手法の実装の提案

コマンドログの比較を行う手法は多く存在する．例えば評価基準として，あるコマンドが実行された時に，そのコマンドは危険であるとしたブラックリストを作成するパターンマッチングの手法がある．また，攻撃であるとされたコマンドを **??**で説明したマルコフモデルで学習させることで，攻撃性を表現する手法がある．しかしパターンマッチングであれば静的解析であるので未知の攻撃に対応ができず，マルコフモデルであれば現在の状

態だけに依存して次の状態への推移確率が決まるので，未知の特徴量を無視してしまうので，いずれも未知の攻撃に対応できない．しかし， 2.4 で説明した意味解析をコマンドログに導入することで，コマンド名が別でも同じような内容のコマンドを実行しようとした時に，それが同じような内容であることを検知できる自然言語処理における意味解析のニューラルネットワークのモデルを評価基準とすることで未知の攻撃にも対応できる．そのため，本研究では自然言語処理における意味解析のニューラルネットワークのモデルを評価基準とした．

6.2.2 機械学習を用いたコマンドログのスコアリング

本研究では，評価基準となる，自然言語処理における意味解析のニューラルネットワークのモデルとして Word2vec の skip-gram モデルを採用した．純正の低対話型 Honeypot で収集した侵入ログで skip-gram モデルの隠れ層の重みを学習させ (これをモデル 1 とする)，同様にして高対話型 Honeypot で収集した侵入ログも skip-gram モデルの隠れ層の重みを学習させる (これをモデル 2 とする)．次に修正済の Honeypot で収集したログをセッション開始からセッション終了までに打たれたコマンドごとに (以降これを 1 セッションごとと呼ぶ) モデル 1 とモデル 2 のそれぞれに入力していき，出力された数値 a を活性化関数としてソフトマックス関数をかけることで，0 ≦ a ≦ 1 の範囲を取るようにし確率的な数値として出力することでスコアリングを行う．このため入力に対して多数存在する出力を全てを合計すると 1 になる．純正の低対話型 Honeypot や高対話型 Honeypot の収集ログをモデル化する際、入力層として収集ログのコマンドの入力に対してそのコマンドの周辺のコマンドを出力として与えることでこれを学習させる．
例えば 3 つのコマンドが打たれたとしたとしたものを以下のプログラム 6.1 に示す．

プログラム 6.1: 3 つの実行コマンドの例

```
1  $ uname
2  $ free
3  $ ps x
4  $
```

モデルを構築する際には”free”コマンドを入力にした時に，出力として”uname”コマンド”ps”コマンドを用意しておくことで，free が入力として与えられた時に他 2 つの出力される周辺のコマンドが出力する確率が高くなるようにする．また，実装としては周辺語をどこまで広げるのかはパラメータとして window size で与えることができ，上記の例の周辺語は”1”であり，window size を”2”にすればモデル化する際に出力層に与えられる数は 4 つとなる．以下の図 6.3 にモデル化のフローを示す．

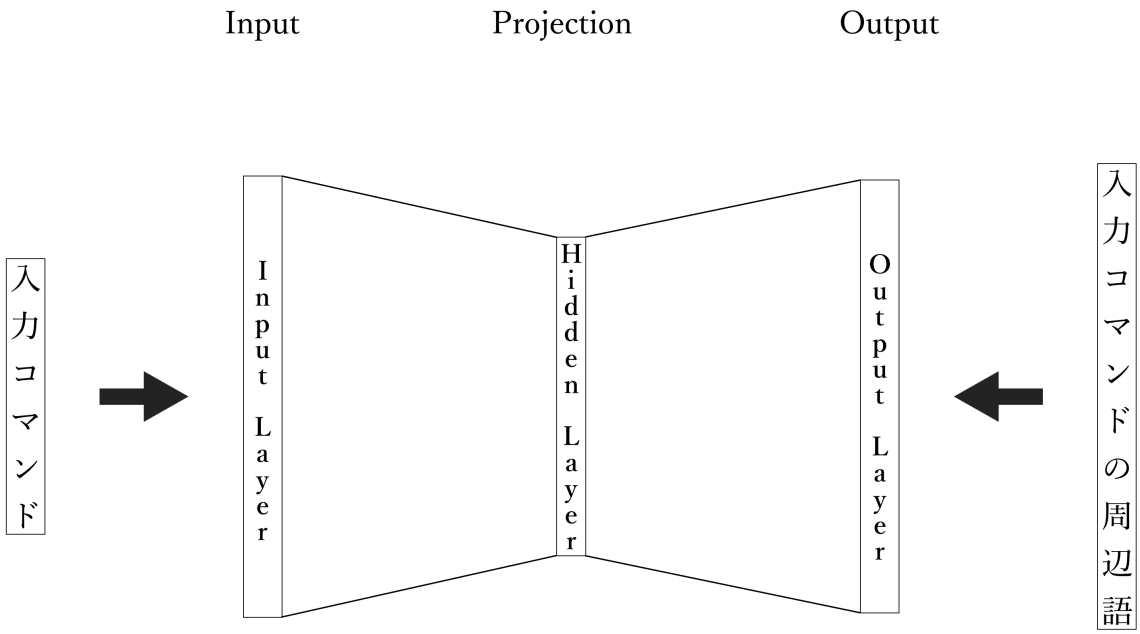

図 6.3: 評価のフロー [1][2]

また，このようにして Honeypot の収集ログに対して各々のモデルを構築する．モデルは *HiddenLayer* に単語ベクトルのインデックスとして構築される．ここで SCDV 2.4.3.4 を使い，word2vec で取得した単語ベクトルを使い，idf 値を計算する． 2.4.3.1.1 次に単語ベクトルごとにクラスタリングすることで，ある単語があるクラスタに所属する確率を算出する．クラスタを考慮した新たな単語ベクトルを再構築し，idf 値を考慮した新たな単語ベクトルを再構築する．1 セッションごとに含まれる単語ベクトルを平均化し，文章ベクトルを得る．

6.2.2.1 コマンド群データのベクトル表現の比較

本研究の評価手法によって任意の 2 つのコマンドログを比較した時の，各々のログの違いを文章分類の手法で算出した結果は以下の図 6.2，図 6.3，図 6.4，のようになった．

表 6.2: 修正前と後の honeypot のコマンドログの比較

	正解率	適合率	再現率	F1 値	session 数
素の低対話型 Honeypot	0.807909	0.782520	0.872437	0.825036	3951
修正済の低対話型 Honeypot	0.807909	0.842795	0.738251	0.787067	3660

表 6.3: 修正前の honeypot と一般 UNIX ユーザーのコマンドログの比較

	正解率	適合率	再現率	F1 値	session 数
素の低対話型 Honeypot	0.778404	0.788169	0.765108	0.776467	3657
一般の UNIX ユーザー	0.778404	0.769086	0.791863	0.780308	3613

表 6.4: 修正後の honeypot と一般 UNIX ユーザーのコマンドログの比較

	正解率	適合率	再現率	F1 値	session 数
素の低対話型 Honeypot	0.822134	0.793420	0.892497	0.840047	3972
修正済の低対話型 Honeypot	0.822134	0.863229	0.744887	0.799703	3618

上記の表から，素の Honeypot と一般ユーザーのコマンドログが一番近く，修正済の Honeypot と一般ユーザーのコマンドログが一番遠いことが分かった．
また，素の honeypot と修正済の Honeypot，一般の Unix ユーザーのコマンドログの 200 次元のコマンド文章ベクトルを以下の図 6.2.2.1 に示す．