システムログ書式の構造に着目したシステム異常検出手法の検討

水谷 正慶[1]

概要：コンピュータシステムのセキュリティ侵害の検出には様々なきっかけがあるが，その 1 つにシステム管理者による目視のログ閲覧が挙げられる．セキュリティ侵害が発生したシステムでは通常と異なる種類のログが出現する可能性が高く，これを発見することでシステムの異常に気づくことができる．従来のシステムログを利用した異常検出ではログの流量やログに出現するパラメータの異常値に着目していたが，本稿ではログの種類に着目した異常検出について述べる．ログの種類の異常を見つけるためにはすべてのログのパターンを把握する必要があるが，これは開発者とセキュリティ担当が異なるなどの理由から現実的に難しい．そこで本稿ではログの種類を自動的に分類する手法を用いて出力されるログのパターンについての知識がなくても異常検出ができる手法を検討し，実際のログデータを用いて手法の有効性を検証した．

キーワード：ログ分析

Experience of system anomaly detection by levelaging log message format structure

Masayoshi Mizutani[1]

Abstract: There are various triggers to detect security breaches of computer systems, one of which is the visual log browsing by system administrator. In a system in which a security breach has occurred, there is a high possibility that a log of a type different from usual is likely to appear, and by discovering this, it is possible to notice a system abnormality. In conventional detection of anomalies using system logs, attention was paid to the flow rate of logs and abnormal values parameters appearing in logs. In this paper, we describe abnormality detection focusing on log types. In order to find abnormality in log type, it is necessary to grasp all the patterns of logs, but this is realistically difficult for reasons such as different developers and security officers. In this paper, we propose a method to detect anomalies without knowledge of the log patterns output by automatically classifying the log types, and to examine the effectiveness of the method using actual log data I verified.

Keywords: Log analysis

1. はじめに

インターネットを介して発生する脅威は日々巧妙化しており，企業や組織の情報資産を守るセキュリティ担当者が攻撃に対応するのはより困難になりつつある．2000 年以降，ボットネットの拡大や Advanced Persistent Threat など，経済的利益を得るためにセキュリティ侵害をする攻撃者の活動は増加しつつあり，巧妙化に拍車をかけている．攻撃者が利益を得るためのセキュリティ侵害も様々な方式があるが，本稿ではユーザにサービスを提供するために稼働しているサーバに対する侵入に焦点を当てる．不特定多数のユーザが利用するサーバであればクレジットカード番号など金融関連情報を含む個人情報の奪取が見込めるため，これが攻撃者の動機となっている．また，近年では仮想通貨のマイニングのために計算資源を利用したり，ランサム

[1] クックパッド株式会社
Cookpad Inc.

ウェアによって重要データを暗号化した上で身代金を要求するといった攻撃が実施できる．さらには安定して稼働するホストとして，別の攻撃のための踏み台として利用したり，Command and Control サーバとして利用されるなど，セキュリティ侵害後の価値は多岐にわたる．2000 年後半から 2016 年頃にかけて発生した Drive by Download 攻撃や 2016 年初旬から広まったランサムウェア攻撃によりユーザが利用している PC へのリスクが増した [1] が，サーバへの攻撃は依然として対策が必要となっている．特にサービスで管理する個人情報だけでなく，仮想通貨に代表される直接的な資産の奪取やサービスの継続に必要な情報資産を利用不能な状態にされて脅迫されるといった事案も発生しており，以前よりサービスを提供，あるいはそれに関連したサーバの重要度は以前よりも高まっていると言える．

サーバへのセキュリティ侵害対策にも防止を対象とした様々な手法が取り入れられているが，本稿ではセキュリティ侵害の検知に着目し，さらにセキュリティ担当者が目視によってログをチェックする部分に焦点をあてる．通常，サーバでは提供しているサービスやミドルウェアなどのログ，さらにリモートログインのログなど，サーバに関連するログが出力されている．これらのログはセキュリティ侵害の検知，およびセキュリティ侵害検知後の対応において役立てられる．これらのログからセキュリティ侵害を検知する方法として Security Information & Event Manager (SIEM) やホストベースの Intrusion Detection System (IDS) のようにログを読み取って分析するツールがある．ただし，現状これらの多くはログが出現する文字列やパターンなどを事前にルールとして組み込んだ上で検知に利用している．

一方，セキュリティ担当者がログからセキュリティ侵害を見つけるというケースもある．セキュリティ担当者がログを見る場合，特定の文字列やパターンを見つけようとするだけではなく「通常と異なるログがでていた」というケースが挙げられる．「通常と異なる」という状態を網羅的に定義するのは難しいが，本稿では以下のような事象であると定義する．

(1) **出現するログの種類が通常と異なる** ログを見る人物がログをカテゴリ分けして，既出のカテゴリに属さいないログを検出する場合．例えばサービスに対するアクセスのログが出力され続けている途中に，1 箇所だけシステムのエラーが出現している場合，脆弱性を利用されてシステム側に干渉された可能性を疑うことができる．

(2) **出現するログの値が通常と異なる** ログのカテゴリは同じだが，そのログ中に出現する値が既出のログと異なる傾向をもつ場合．先述したサービスに対するアクセスログをもとに考えると，ログ中でサービスの実行結果を示す情報が表示されるべき箇所に，エラーメッセージやリソースとは関係ないシステムの情報などが出現する事象が挙げられる．これは攻撃によってサービスやシステムが意図しない動作を起こした影響だと見ることができる

本稿では（1）の出現するログの種類が通常と異なるケースに着目し，ログ書式の構文を自動推定する手法を用いた異常の検出方法を検討する．ソフトウェアから出力されるログの種類は，多くの場合ソフトウェア内で定義されている書式と同一と考えることができる．ログは多くの場合，出力される際の事象に応じて値が埋め込まれ，同じ種類のログでも同一の出力結果にはならない．そのため単純な文字列比較ではログの種類を識別するのは難しいが，文字列処理の技術を応用して出力されているログから書式を推定することで種類の識別が可能になる．これによって，通常出現するログの集合から通常出現するログの種類の集合を生成できるようになり，その集合から外れるログの種類の出現を異常とみなすことができるようになる．本稿ではこのような異常検出プロセスのためのログ書式推定手法を検討，およびプロトタイプ実装し実データを用いて実験した結果，そして明らかになった課題について述べる．

2. 課題

本節では解決すべき課題とそれに関連する本稿の仮説と定義について述べる．

2.1 用語定義

本稿での用語を以下のように定義する．

- 同じサービスを提供しているプロセス群を**ドメイン**とする
- 開発者やソフトウェア利用者が動作状況を確認できるようにソフトウェアが出力するテキスト形式のメッセージを**ログ**とする．
- ログには，どのようなテキストを出力するべきかのテンプレートが定められているものとする．このテンプレートを**フォーマット**とする．例えば C 言語の printf 文で表現すると，第 1 引数になるフォーマット文（例 `"Fail to open %s"`）が本稿で述べるフォーマットとなる．
- 本稿ではログを発生させるきっかけとなる事象を**イベント**と呼ぶ．イベントは 0 個以上の**属性値**を持つ．この属性値がフォーマットに従って整形され，ログのテキストに変換される．

2.2 課題の定義

本稿では，セキュリティ侵害を含む異常発生時には平常時と異なるフォーマットのログが含まれている，という仮説にもとづいて異常発生時のログを特定することが課題となる．これを分解すると次の 4 点となる．

(1) 平常時のログのセットとして L_0 が与えられ，そこからフォーマットの集合である F が生成できる
(2) $m_0, m_1, \ldots, m_n$ と平常時のログが与えられる．これらの m_i は F に属するフォーマットから生成されたログと判定できる
(3) 異常発生時のログ m_e が与えられたときは，F に属さないログと判定できる
(4) 精度向上のため，L_0 以降に入力された m_i を元にして F の内容を更新することができる

まず（1）について，平常時のログのセットから異常検出の起点となるフォーマットの集合を生成できる必要がある．これは機械学習による判定のためのモデルと同等の意味を持つ．そして（2）と（3）では，生成されたフォーマットの集合および新たに到着したログを用いて，それが過去に出現したログと同じ種別であるかを判定できなければならない．判定の対象となるログは，セットではなく個々のログに対して判定ができるようにする必要がある．詳細については第 4.1 節で述べるが，パフォーマンスおよび実時間性の観点から単体のログに対して処理できる必要がある．さらに（4）として，新たに到着したログに対してフォーマット集合（あるいはモデル）を更新できる必要がある．これは，新たに出現するログを継続的に投入していく必要があり，これらの手法では長期的なログ投入によってのみ起こりうるフォーマットの微細な変化に追従するのが難しくなるためである．例えば日時情報を含むログの場合，1 ヶ月分の平常時ログセットからフォーマットを生成して異常検出を試みると，`"2017 04/01"`と`"2018 04/01"`のような年の違いによって起こる値の差分を吸収できず，異常とみなすことになってしまう．

また，前提条件としてフォーマットの判定にソフトウェアの元になったソースコードや実行ファイルは参照しないものとする．これは，ログ出力の指定方法がソフトウェアやフレームワークごとに異なるために出力箇所を特定する一般化された方法が無いこと，ログそのものとログを出力したソフトウェアのバージョンを一致させて管理しなければならず手間がかかること，プロプライエタリなソフトウェアの場合は仕様の参照が難しいこと，そしてソフトウェア上で定義されているフォーマット情報と出力されたログの関係性を管理するコストが高いことを理由に現実的には困難だと考えられるためである．

3. 関連研究

テキスト形式で出力されたログからフォーマットを推定する手法は先行研究で検討されている．代表的な手法としては SLCT[2] や LogHound[3]，IPLoM[4] が挙げられる．SLCT と LogHound はそれぞれ Apriori algorithm に類似したアルゴリズムを用いてログのセットをクラスタリングし，そのクラスタからフォーマットを推定している．IPLoM もログのセットをパーティションと呼ばれる単位に分割して処理することでクラスタを生成し，それに基づいてログのフォーマットを推定している．これらは基本的に最初に与えられたログのセットからフォーマットを生成するのみのバッチ型処理を想定しており，本稿での目的には沿わない．

一方，Spell[5] や Drain[6] はオンライン型，もしくはインクリメンタル型と呼ばれ，順次ログを投入する手法として提案されている．両者とも大量のログを処理することを想定し効率よく処理できるよう設計されているが，より正確にログの元になったフォーマットを推定することに注力されている．本稿の目的から考えると，フォーマットの正確性，つまり出力元ソフトウェアで規定しているログのテンプレートに沿っているかは本質となる部分ではない．例えば異常検出の処理において，元のフォーマット 1 つに対し 2 つのフォーマットが存在すると認識されたとしても，結果として異常検出ができれば課題の解決になっている．そのため，この 2 つについてはフォーマット生成の過程に参考とできる部分はあるものの，完全に流用するなどはしていない．

また，ログの値を見る異常検出についても先行研究はある．例として DeepLog[7] は推定したフォーマットから出現する値の出現パターンを LTSM で学習させる手法になっている．また Drain もログをフォーマット化しそこに含まれる属性値を取り出すことで Xu らの手法 [8] にならって異常検出を試みている．しかし第 1 節で述べたとおり本稿の目的はログの種類の異常に着目しているため，ログ中に出現する属性値を用いた異常検出については本稿では対象外としている．

4. 設計

4.1 要求事項

本節では，検討手法を実環境において効果的に動作させるために必要となる要件について述べる．

- **実時間性** 本稿における異常検出は主にセキュリティ侵害の検知を目的としている．多くの場合，セキュリティ侵害は時間の経過とともに被害が拡大するため，検出からの対応が迅速であるほど影響範囲を極小化できる．一方でシステムのセキュリティ監視によって発

報された事象は自動的に対応されるわけではなく，実際の侵害なのか誤検知なのかについてセキュリティ担当者の判断が伴うという運用が多い．したがって人間の判断によって生じる範囲の遅延は許容できると考えられ，最大で 10 分程度に収まるべきと言える．

- **時間計算量の抑制** 計算資源の節約は一般的な要求事項ではあるが，ログの処理においては特に重要な制約となる．実時間性の項目とも関連するがログは継続的に発生するため，計算量が大きいと発生するログに対して処理が間に合わなくなってしまう．特に異常検出という処理の特性から過去に受けたデータを利用する必要があるため直列性が高くなり，並列処理で負荷を分散させるのが難しくなる．例えば他のログフォーマット推定手法で多く利用されている Longest Common Sequeuse (LCS) では比較対象となるシーケンス数 n_1, n_2 に対して $O(n_1 \cdot n_2)$ となりシーケンス数の増加に対して負荷が大きくなってしまう．よって，現在モデルが保持しているフォーマット数 $|F|$ ，新規に投入されるログの文字列長 n に対して $O(|F| \cdot n)$ 程度を目標とする必要がある．
- **空間計算量の抑制** 記憶領域の節約も一般的に重要だが，特に投入したログ量に対して配慮する必要がある．ログの処理においてログの投入数はもっとも影響しやすい変数であるため，投入したログ数が影響しないアルゴリズムにする必要がある．具体的には現在モデルが保持しているフォーマット数 $|F|$ に対して $O(|F|)$ 程度となるアルゴリズムが望ましい．

4.2 アルゴリズム

本節では L_0 を用いた学習，およびその後に与えられるログ m_i に対して異常検出をするためのアルゴリズムについて述べる．本稿で検討するアルゴリズムは第 4.1 節で述べたとおり，パフォーマンスや処理の手順に関して制約がある．そのためフォーマット推定の部分も独自の簡素なアルゴリズムを採用して，異常検出を試みた．アルゴリズムに必要な入力は L_0, m_i, th の三種類である．

- L_0：平常時のログのセット
- m_i：L_0 以降に発生したログ
- th：（0 より大きく 1 未満の数値）ログをフォーマットに属すると判断するためのしきい値

これらの入力に基づいて動作する学習（Algorithm 1），検出（Algorithm 2），新しいログの更新（Algorithm 3）のアルゴリズムをそれぞれ示す．学習と検出における基本的なアルゴリズムに大きな違いはなく，入力値と返り値が学習と検出の状態にあわせて調整されているのみである．学習ではモデルとしてフォーマットの集合が返り，検出では新出のフォーマットと判定されれば ϕ，既出であれば当該フォーマットが返る．検出の結果を受けて，更新では検出時に得られたフォーマットおよびログ情報を渡すことでモデルを更新する．

Algorithm 1 Training

```
Require: L_0, 1 > th > 0
  F ← φ
  while m ∈ L_0 do
    T ← tokenize(m)
    f, s ← searchFormat(F, T)
    if f = φ|s < th then
      F ← F ∪ newFormat(T)
    else
      mergeFormat(f, T)
    end if
  end while
  return F
```

Algorithm 2 Detection

```
Require: m, F, 1 > th > 0
  T ← tokenize(m)
  f, s ← searchFormat(F, T)
  if f = φ|s < th then
    return φ
  else
    return f
  end if
```

4.2.1 トークンの分割

Algorithm 1, 2, 3 に登場する `tokenize` 関数はテキスト形式のログをトークンに変換する．本稿でのトークンとはログのテキストを意味のある文字列に分割したものを指す．例として，`"Fail to open /tmp/a.txt"`というログであれば，`"Fail"`, `" "`, `"to"`, `" "`, `"open"`, `" "`, `"/tmp/a.txt"`といったトークンの集合になる．これは本稿で検討する手法に限らず，ログのフォーマット分析ではしばしば用いられる手法であり，第 3 節で挙げた先行研究でも利用されている．パラメータが（`printf` 関数におけるフォーマット指定子の部分に相当）に埋め込まれる属性値は可変長であることが多く，トークン分割によりテキスト形式で一文字ずつパラメータ部分を探すのに比べて問題が単純化できるという利点がある．しかし，ログに出現する文字をどのように分割するかは各開発者に依存しているため，標準的なトークン分割の手法というものはない．先行研究では IP アドレスや国際標準化機構で定義されている日付書式などを抜き出すなどの手法をとっているものもあるが，

本稿では経験則として括弧やカンマ，クォーテーションマークなどの記号に加えて空白文字などで分割する手法を採用した．これは，標準化された日付などの書式を網羅するのはあまり現実的ではなく，さらに書式を検査すると負荷も大きくなりがちという理由にもとづく．

Algorithm 3 Update

Require: m, f, F
  $T \leftarrow tokenize(m)$
  if $f = \phi$ **then**
    $F \leftarrow F \cup newFormat(T)$
  else
    $mergeFormat(f, T)$
  end if

4.2.2 類似するフォーマットの検索

`searchFormat` 関数（Algorithm 4）は与えられたログに対して既存のフォーマットから最も近いものを探しだす．`tokenize` 関数での説明の通り，ログをどのようなトークンに分割したかが推定されるフォーマットに影響を与える一方で，パラメータとなる部分を正確に 1 つのトークンとして抜き出すのは著しく困難である．そのためフォーマットを推定している段階では 1 つのパラメータとなる部分に 1 つ以上のトークンが出現することになり，先行研究では複数のトークンを適切に 1 つのパラメータにまとめるための試みがなされている．SLCT や LogHound ではパラメータではないトークン，すなわち同じ種類のログなら共通して出力される固定のテキストとなる共通部分を抜き出すことによって，トークン長が可変であることに対応している．Drain ではトークン長でログを仕分けした上でフォーマットを推定し，あとから類似するフォーマット同士をグループとしてまとめることで対応している．また，パラメータが埋め込まれて出力されるログは順序性が高いため LCS のようなアルゴリズムを使えば共通部分だけを容易に抜き出すことができる．ただし LCS では計算量が大きくなるという問題があるため，本稿で挙げている要求事項に対して適切ではない．

そこで本稿ではトークン長ごとに保持するトークンを完全に分離するアプローチを採用した．Algorithm 4 で示している `filterByLength` 関数は，ログを分割したトークン集合 T に対してモデルが保持しているフォーマット集合 F からトークン帳が $|T|$ であるフォーマット集合 F' を取り出す．そして，F' から最も T との一致率が高いフォーマットを `calcSimilarity` 関数で計算する．`calcSimilarity` 関数ではフォーマット内のトークンとログのトークンを先頭から比較し一致するトークンの多さによって一致率を判定する．一致率は 0 以上 1 以下の数値で，大きい数値ほど一致率が高い．このトークン数で分割する手法は Drain と似ているが，本稿での手法ではトークン長ごとのフォーマットを生成した後に類似するフォーマットをとりまとめるという手順を省いている．これは本稿の目的が正確なフォーマットを発見することではなく異常検出に利用することなので，本来 1 つであるフォーマットが複数あると判定されても問題ないことからこのようなアプローチを選択した．

弊害としてトークンが可変長となるログが出現した場合，事前に学習したトークン長と異なるフォーマットが出現すると異常と判定されてしまう可能性があるが，十分な量の学習データを用意できれば出現する可能性のあるトークン長を網羅できるのではと考えられる．このアルゴリズムにより，保持しているフォーマットの最大トークン長を M として，計算量は最悪で $O(|F| \cdot n)$)，最良で $O(\frac{|F| \cdot n}{M})$ になり，要求に合致している．さらに通常の動作において $|F|$ が急速に成長するべきではないので，実質的には $O(n)$ に収束することが期待できる．

Algorithm 4 function searchFormat

Require: F, T
  $F' \leftarrow filterByLength(F, |T|)$
  $similarFormat \leftarrow \phi$
  $maxRatio \leftarrow 0$
  while $f \in F'$ **do**
    $s \leftarrow calcSimilarity(f, T)$
    if $s > maxRatio$ **then**
      $maxRatio \leftarrow s$
      $similarFormat \leftarrow f$
    end if
    return $similarFormat, maxRatio$
  end while

5. 実験

第 4 節の設計にもとづいて，ログ異常検出のためのプロトタイプを Go 言語によって実装した [9]．この実装を用いて実データを用いた異常検出の実験，およびパフォーマンスに関する測定を実施した．また，各実験において th は 0.65 に設定した．

5.1 異常検出の実験および結果

異常検出の実験には Amazon EC2 プラットフォーム上で実サービスを提供している約 1,000 台のインスタンスから syslog によって得られるログを利用した．対象インスタンスで提供しているサービスは，公開 Web サービスおよびその周辺システムとなっている．基本的にはインスタンス上で提供されているサービスのアプリケーションログではなく，OS やミドルウェアなどの管理ソフトウェアに関連したログを対象としている．ログは連続した 48 時間分のデータを用意し，24 時間で区切って 2 つのデータセットとした．これらのデータセットを D_1, D_2 とする．データの詳細については表 2 に示す．さらに同じ名前のプロセス名は同じドメインのものであると考えることができるため，実験結果をより細かく議論できるようにデータをログ出力元のプロセス名によって分割した．

実験は D_1 を学習データとして各プロセスごとのモデルを作成したのち，同じプロセスの D_2 ログを順次投入して新しいフォーマットが検出されたかを調査した．結果を表 1 に示す．結果では D_2 で新たに検知されたフォーマット，

表 1　異常検出実験の結果
Table 1　A result of anomaly detection experience

グループ	検知数	プロセス種類数	D_1 のログ数平均値	D_1 のログ数中央値	主なプロセス名（一部関連プロセスを集約）
G_1	0	142	33,157.34	70	apache2, dbus, irqbalance, mysqld, postfix, nptd, ncsd, postgresql, rsyslogd, systemd, zabbix
G_2	1〜5	31	199,482.74	537,574	sshd, sssd, su, CRON, bash, dockerd, chronyd, consul, influxd, cloud-init
G_3	6〜10	11	5,092,074.09	58,529	sudo
G_4	11 以上	13	862,248.53	24,539	kernel, dhclient, java, docker

ログ数平均値は小数点第 3 位以下切り捨て

表 2　異常検出実験データの概要
Table 2　A summary of experience data for anomaly detection

	ログ件数	データサイズ	プロセス種類数
D_1	77,650,359	約 18.76 GB	190
D_2	76,825,413	約 18.64 GB	197

すなわち異常と判定された件数をそれぞれ 0 件，1〜5 件，6〜10 件，11 件以上のグループに分けて分布を示している．これをそれぞれ $G_{1,2,3,4}$ としている．それぞれのグループで対象となったプロセス種類の数，D_1 におけるログ数の平均値，中央値と主なプロセス名を挙げている．本実験は検知精度を論ずるためのものではないが，意図した検出ができているかについて検証する必要がある．そのため，各グループについてログの内容および検出結果の傾向について調査した結果についてまとめる．

まず G_1 は検知数 0，つまり D_1 と D_2 の両方が均一なフォーマットのログのみ出力されていたグループとなっている．プロセス数は 142 で全体の 70%以上になるがログの総数は全体の 5%程度にとどまっており，D_1 においてログ数が 1000 件未満のプロセス数が 98 となってる．具体的には apache，mysqld, dbus, nptd, zabbix 関連などがおよそ 1,000 件前後かそれより少ないログ数である．これはログ数が少ないために異常となるログが出にくいという理由だけではなく，学習のためのデータが少ないことが理由で検出頻度が上がりにくいということを示しており，肯定的な要素であると考えられる．

次に $G_{2,3,4}$ の結果を目視で内容を確認したところ，このグループは D_1 で出現しなかったログを意図どおりに検出できたプロセスとアルゴリズムの不備（主に `tokenize` 関数）によって検出されてしまったプロセスに分かれるとわかった．意図通りに検出できた例としては sshd，su，consul, influxd, bash が挙げられる．例えば，`sshd` では `"Received SIGHUP; restarting."` というログとして臨時に作成したホストの sshd が再起動しているのを捉えていた．また `su` では`"pam_unix(su:auth): authentication failure; logname=****** uid=***** euid=0 tty=/dev/pts/5 ruser=****** rhost= user=****"`（一部マスク済み）として認証失敗のログを検出していた．他にも各プロセスにおいて D_1 では発生していなかった処理の失敗の検出に成功していた．

一方，意図しないログの検出はトークン分割の問題に起因していた．以降に具体的な例を示す．

- `sudo` プロセスのコマンドログ：経由して実行するコマンドは内容が任意長になってしまいかつコマンド長も場合によって全く異なる値になるため固定長のトークンを前提にするとずれが発生してしまう
- 任意長のリスト構造をそのまま出力するログ：ある程度リストの長さがまとまっているログであれば対応可能だが，分散している場合は対応が難しい
- トークン分割に用いる文字が任意の数含まれるログ：JSON のように分割文字が含まれつつ任意の構造を持つものや IPv6 アドレスのように一部の省略が起こりうる値が含まれるもの
- 可読性のために空白文字で余白調整するログ：スペースの数によってトークン数が変化してしまうため
- 任意のテキストメッセージを含むログ：人間が記述した自然言語が含まれるログだと共通した構造に落とし込むことができない場合もある

これらの問題の一部についてはトークン分割のアルゴリズムの改良やヒューリスティックな手法の取り込みが解決方法として考えられる．ただ，完全に任意の文字列が含まれるログはそもそも対応が困難な場合もあるので，ログの種類の選別やログの複雑性の評価などと組み合わせた手法で改善できる可能性があると考えられる．

5.2　パフォーマンスの測定

パフォーマンスの測定には公開データとなっている Blue-Gene/L のログ [10]，および Xu らの研究 [8] で利用されていた HDFS のログ [11] を利用した．これらデータセットをそれぞれ P_1, P_2 とする．測定環境には CPU が 2.9 GHz Intel Core i7，メモリが 16GB，記憶領域に SSD を搭載している MacBookPro を利用した．OS は macOS High Sierra (10.13.6) となっている．測定ではそれぞれのデータの読み込みにかかった時間を time コマンドで実時間を計測した．

表 3　パフォーマンス実験データの概要と結果
Table 3　Data summary and result of performance experience

	データ名	ログ件数	所要時間	フォーマット数
P_1	BlueGene/L	4,747,963	15.57 秒	82
P_2	HDFS	11,197,705	34.22 秒	115

パフォーマンス測定の結果を表 3 に示す．スループットの観点からはそれぞれ毎秒約 30 万件のログを処理できていることがわかる．今回検証のために利用しているデータセット $D_{1,2}$ の規模感と照らし合わせても十分な性能がでていることがわかる．He らの論文 [6] にて同じデータセットを用いたパフォーマンスの測定がなされているが，彼らの提案手法は P_1 の処理に 128.03 秒，P_2 の処理に 425.15 秒を要している．実装の詳細が不明であるため直接的な比較で優位性を論じることはできないが，他の実装と比較しても問題のない性能がでているとわかる．

また，生成されたフォーマット数から記憶領域が圧迫されていないことも読み取れる．本手法は記憶領域の利用がフォーマット数に従って増加するが，100 前後であれば少ない記憶領域でも十分実用的に動作することが期待できる．

6. 考察および今後の課題

6.1 フォーマット推定アルゴリズムの改善および別手法との併用の検討

第 5.1 節で示したとおり，今回の実験で意図通り検出できなかったログの多くはフォーマット推定のアルゴリズム，特にトークン分割の部分の対応によるものであった．これはトークン分割の方法やトークンとフォーマットの比較関数を調整することで一定の改善を見込める．しかし一方で人間による自然言語テキストが出現するログや自由なスキーマの JSON が含まれるログなど，`printf` 関数のフォーマット文のようなテンプレートにイベントの短い属性値を埋め込んで出力する，というフォーマット推定手法の前提が成り立たなくなるログも存在している．したがってフォーマット推定のアルゴリズムを改善するだけではなく，そもそもフォーマット推定によって解くべきログとそうでないログを事前に識別するなど，手法を適用する前の段階で別手法と組み合わせるなどのアプローチを検討する必要がある．

6.2 検出結果に対するセキュリティ侵害の判定

第 5.1 節の通り，今回の実験では一部ではあるが，学習段階で発生しなかったログの種類を新たに入力したログから検出することに成功した．これらは意図した通り学習段階とは異なるイベントから生成されたログであったが，一方でセキュリティ侵害発生の可能性を示唆するログはごく一部のみであり，多くの検出結果はシステムの不具合等を示すログであった．これはシステム運用の観点からの異常検出にも応用可能であることを示す一方で，セキュリティ侵害を発見するという目的のためには本手法だけでは不十分であることを示している．改善の案としてはシステムやサービスのログではなくセキュリティ製品などから出力されるログと組み合わせたり，Indicator of Compromise (IoC) 情報と組み合わせてログのリスクを評価するというアプローチが考えられる．

6.3 長期的な運用におけるモデルの調整

本稿では 24 時間分の学習データに対し新規の 24 時間分のログを投入するという実験を行い，その結果を示した．そのため，短期的には今回の実験で明らかになった性能を発揮できるが，本来であればこの手法を用いた長期的な運用によってより精度の高いモデルへと更新していくことが期待される．しかし，長期的に運用して学習を継続させた場合，モデルが意図しない更新を続けてしまう可能性もある．また取り込むログの性質によっては攻撃者が意図的に細工したログを学習データとして注入できる場合もあり，モデルそのものに対する攻撃も懸念される．これを防ぐために学習結果を反復的に調整するような機構を検討する必要がある．

7. まとめ

本稿では主にサーバから出力されるシステムのログからセキュリティ侵害に関する異常を検出するため，実環境での運用を考慮した設計とそれに基づいたプロトタイプの実装し実データによる実験を行った．その結果，本稿で提案した手法では未だに誤検出が多く実用的な性能は得られなかったが，解決するべき課題や指針が明らかになった．今後は実環境で継続的に手法の改善に取り組むことで，より実用的な異常検出が可能になることが期待される．

参考文献

[1] 日本アイ・ビー・エム株式会社マネージド・セキュリティー・サービス. 2016 年 下半期 Tokyo SOC 情報分析レポート, Mar 2017. `https://www.ibm.com/blogs/tokyo-soc/tokyo_soc_report2016_h2/`.

[2] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. in IEEE IPOM'03 Proceedings, pp. 119–126, 2003.

[3] Risto Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In Finn Arve Aagesen, Chutiporn Anutariya, and Vilas Wuwongse, editors, *Intelligence in Communication Systems*, pp. 293–308, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[4] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 24, No. 11, pp. 1921–1936, Nov 2012.

[5] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *Proceedings - 16th IEEE International Conference on Data Mining, ICDM 2016*, pp. 859–864,

United States, 1 2017. Institute of Electrical and Electronics Engineers Inc.

[6] P. He, J. Zhu, P. Xu, Z. Zheng, and M. R. Lyu. A Directed Acyclic Graph Approach to Online Log Parsing. *ArXiv e-prints*, June 2018.

[7] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pp. 1285–1298, New York, NY, USA, 2017. ACM.

[8] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 117–132, New York, NY, USA, 2009. ACM.

[9] Masayoshi Mizutani. logstruct. `https://github.com/m-mizutani/logstruct`.

[10] CFDR Data USENIX. `https://www.usenix.org/cfdr-data`.

[11] Wei Xu. SOSP 2009 Log Dataset. `http://iiis.tsinghua.edu.cn/~weixu/sospdata.html`.