修 士 論 文

APT攻撃を検知するための
ファイルアクセスの記録と比較手法の研究

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

園田 真人

2016年3月

修 士 論 文

APT攻撃を検知するための
ファイルアクセスの記録と比較手法の研究

指導教員　篠田陽一 教授

審査委員主査　篠田陽一 教授
審査委員　丹康雄 教授
審査委員　知念賢一 特任准教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

1310037 園田 真人

提出年月: 2016 年 2 月

概 要

Advanced Persistent Threat(APT) 攻撃は対象の情報を窃取することを目的として明確な犯意を持って知能的な活動を行う標的型攻撃の一種である。APT 攻撃の特徴として 1. 特定の相手を対象として執拗な攻撃を行う 2. 相手に合わせて侵入方法を選択する 3. 侵入後に長期的な探索を行うことが挙げられる。この 3 つの特徴は、情報の機密性を維持するために高度なサイバーセキュリティ対策を実施している組織に対して情報窃取を行うためであると考えられる。APT 攻撃による被害例として、2010 年には google や yahoo!などの 30 以上の Web サイトを対象とした Operation Aurora や 2011 年には 70 以上の企業や政府機関などを対象とした Operation Shady RAT(Remote Access Trojan) による大規模な情報流出が起きている [1] [2] [3] [4]。APT 攻撃の情報の窃取により発生する大規模な情報流出は、損害賠償による有形資産の減少の可能性や無形資産におけるコーポレートブランドの低下といった甚大な被害を組織にもたらすものである。

APT 攻撃の活動段階は侵入段階、内部活動段階、情報持出段階の 3 つに大別される。侵入段階ではエキスプロイトコードの実行を目的としたスピアフィッシングが用いられる。スピアフィッシングは、脆弱性が報告されてから修正されるまでに侵入を行うゼロデイ攻撃またはヒューマンエラーから侵入を行うソーシャルエンジニアリングを利用するため、APT 攻撃における侵入を完全に防ぐことは困難である。内部活動段階では、侵入したシステムの環境を調べた後に、目的の情報を窃取するためにファイル/ディレクトリの探索が少ない頻度で長期的に行われる。この特性により IDS によるアノマリ検知を回避するため、内部活動を検知することは困難であるとされている。情報持出段階は、踏み台のサーバを経由して窃取した情報を外部に持ち出す段階である。情報の暗号化を行い通信プロトコルを偽造するため、ネットワークトラフィックから情報持出活動を検知することが困難である。また、情報持出段階では攻撃者が目的の情報にアクセスしていることから、この段階に到達した時点で情報流出が始まっている。よって、APT 攻撃による情報流出を防ぐためには、情報持出の段階以前で APT 攻撃を検知し阻止しなければならない。

本研究では、ファイルアクセスログ（FAL）から Tree Structured Log（TSL）を構築し、その TSL を記録し比較を行うことでアノマリ検知を行う fspeek を提案する。fspeek における APT 攻撃の検出手法として、ファイルアクセスのアクセス範囲を尺度としてアノマリ検知を行う File Access Scope T-test（FAST）を提案する。システムを常用する正規のユーザと攻撃者の間では、システム内におけるファイルアクセスの活動の傾向が異なり、活動傾向の差異がファイルアクセスログにおけるアクセスの範囲で表現できることを利用する。FAST では、ファイルアクセスのアクセス範囲を TSL の面積として定量化し、指定した区間における TSL の面積を時系列でから 2 つのグループに分けて、対応のある T 検定を行う。このとき攻撃者が内部で長期間活動している期間があれば、どちらかのグループの面積が大きくなることで T 検定での有意差が生じるためアノマリ検知が可能となる。fspeek は、FAST において長期間の TSL を比較するために、ユーザが利用しているシステ

ム上で常時動作させることを前提としており、長期的に TSL を記録できるように記録するデータ量は FAL よりも小さい。

fspeek を利用して、2015 年 2 月から 11 月の 10ヶ月分の TSL に対して提案手法をもとに実験を行った。2 月から 6 月の read only TSL をグループ A、7 月から 11 月の read only TSL をグループ B とする。そして、2 月から 6 月の read only TSL に探索的なファイルアクセスを混ぜた TSL をグループ A_n とする。また、7 月から 11 月の read only TSL に探索的なファイルアクセスを混ぜた TSL をグループ B_n とする。ここで、B_n 及び A_n の n は read only TSL に混ぜる 1 日当たりのファイルアクセスの回数を示す。n=(1,2,3,4,6,8,12) として各 TSL の面積を計算し、帰無仮説にグループ間に差はないとおいて T 検定を行った。結果として、有意水準 α が 10%のときの t 検定 (B,A_3),(B,A_4),(B,A_6),(B,A_8),(B,A_{12}),(A,B_6),(A,B_8),(A,B_{12}) において有意差があることから、長期間な活動が行われたときに、その活動を検知することが可能であることが分かった。よって、ユーザのソフトウェア操作によるファイルシステムへのアクセスの範囲からアノマリ検知が可能となった。

APT 攻撃の内部活動における長期的で探索的なファイルアクセスが検知可能となった。よって、機密情報が攻撃者に取得される前に攻撃者の長期的な内部活動を検知した場合、攻撃者が侵入しているシステムをネットワークから隔離することで、情報流出を防ぐことが可能となった。また、システムに fspeek を導入した時点で、そのシステムが攻撃者に侵入されていた場合、その攻撃者の活動が停止したときに、蓄積された TSL の比較によって攻撃者が活動停止以前に行っていた内部活動の検知が可能となった。ゆえに、一度システムに侵入した攻撃者による 2 回目以降の内部活動による情報窃取を未然に防ぐことが可能である。以上より、APT 攻撃における内部活動を検知することで、情報窃取による情報流出の被害を軽減させることが可能である。また、本研究の提案した fspeek は、FAL をもとにした TSL を年単位で記録することが可能であり、システムを操作しているユーザに普段行われない活動が含まれていないかどうかを検知するものである。ゆえに、fspeek を稼働させることで、サイバーセキュリティにおける真正性および責任追及性の向上も期待できる。

目 次

第1章　はじめに

本章では、研究の背景・目的・構成を述べる。

1.1　背景

標的型攻撃の一種である Advanced Persistent Threat(APT) 攻撃は 2005 年にアメリカとイギリスの Computer Emergency Response Team により存在が確認され、2010 年以降被害が増大している [5]。2010 年には google や yahoo!などの 30 以上の Web サイトを対象とした Operation Aurora や 2011 年には 70 以上の企業や政府機関などを対象とした Operation Shady RAT(Remote Access Trojan) による大規模な情報流出が起きている [1] [2] [3] [4]。

APT 攻撃の情報の窃取により発生する大規模な情報流出は、損害賠償による有形資産の減少の可能性や無形資産におけるコーポレートブランドの低下といった甚大な被害を組織にもたらすものである。2011 年に起こった RSA の約 4000 万の（SecurID）トークンシードファイル流出による損失額は約 6600 万ドルであり、同じく 2011 年に起こった PlayStation Network に登録されていた 1 億人以上の個人情報が流出した Sony の損失額は 1 億 7000 万ドルとされている [6] [7]。組織の価値及び情報を守るためには APT への対策は欠かせないものとなっている。

セキュリティベンダや研究機関等により、APT 攻撃の対策が行われているが、脆弱性が報告されてから修正されるまでに侵入を行うゼロデイ攻撃や、人間のミスや心理的な隙をついて侵入を行うソーシャルエンジニアリングが利用されるため完全に侵入を防ぐことはできない。

1.2　目的

APT 攻撃は対象の情報取得を目的として行われる、標的型攻撃の一種である。APT 攻撃では侵入後に検知されることを回避しつつ、情報を取得する活動が行われる。この活動は攻撃者が長期的な観点から情報を取得を狙う明確な犯意のある知能的な活動である。APT 攻撃による情報流出を防止するためにファイルシステムへのアクセスを持続的に記録し、比較するシステムを提案する。ファイルシステムのアクセスログを木構造-Tree_Structured_log（TSL）-として蓄積し、単一の TSL の時間的な変化や異なる時期の TSL の比較を行うことで、ファイルシステム上で発生する特異なアクセスの検出が可能と考える。提案するシ

ステムにより潜伏活動を行う攻撃者の検知が可能となり、サイバーセキュリティにおける情報流出の防止に寄与ができる。

1.3 本研究の構成

第 2 章では、APT の特徴及び既存の対策手法とその課題について述べる。第 3 章では、ソフトウェアの操作時に監査システムによって記録されるファイルアクセスログ（FAL）について述べる。第 4 章では、FAL を構造化させた TSL の概要から TSL の特徴及び TSL を用いた演算について述べる。第 5 章では、FAL と TSL を用いてアノマリ検知を行うシステムである fspeek の設計を述べる。次の第 6 章では設計をもとに fspeek の実装を述べる。第 7 章では実装したシステムを使い、実験を行い結果について考察を行う。第 8 章では fspeek の適応性について評価と考察を行う。第 9 章ではまとめ及び今後の展望を述べる。

第2章　APT攻撃に対するサイバーセキュリティの現状と課題

2.1　APT攻撃の特徴と活動段階

本章では、APT攻撃の概要と特徴から既存の対策手法とその課題について述べる。システムに対して侵入活動を行い、データの取得や破壊・改ざんなどを行いサービスを不能にさせる行為をサイバー攻撃という。従来サイバー攻撃の対象は不特定多数であり、愉快犯目的に攻撃を行うものが大半であったとされている。特定の対象に特定の目的で侵入活動を行うサイバー攻撃は標的型攻撃と区別される。

APT攻撃は情報の窃盗を目的としてシステムへの侵入活動を行う標的型攻撃の一種である。システムを破壊またはサービスの提供を不能にさせるといった妨害活動はAPT攻撃には含まれない。また、目的を達成するためにゼロデイ攻撃やソーシャルエンジニアリングなど多彩な攻撃手段を用いた侵入を行う。システム内部に侵入した後は長期的に情報収集活動を行うことで、既存の異常検知システムの検出から逃れる。目的の情報が得られた後は外部に情報を転送し、新たな情報を狙うまたは新たな侵入先への足掛かりとしてシステム内部に潜伏し続ける。

概要よりAPT攻撃には以下の特徴がある。本研究は下記の特徴を前提として進めていくものである。

- 情報の窃盗を目的
- 特定の相手を対象
- 長期的で執拗な攻撃
- 相手に合わせて侵入方法を選択

様々な目的をもつ標的型攻撃とは異なり、APT攻撃は情報を窃盗を目的とする攻撃であるため、侵入方法は異なる場合でも目的は明確なため、以下のように侵入から情報を持ち出すまでの活動を大まかに分類することが可能である。図2.1を用いて、APT攻撃の活動段階についての詳細を記す。

1. 調査

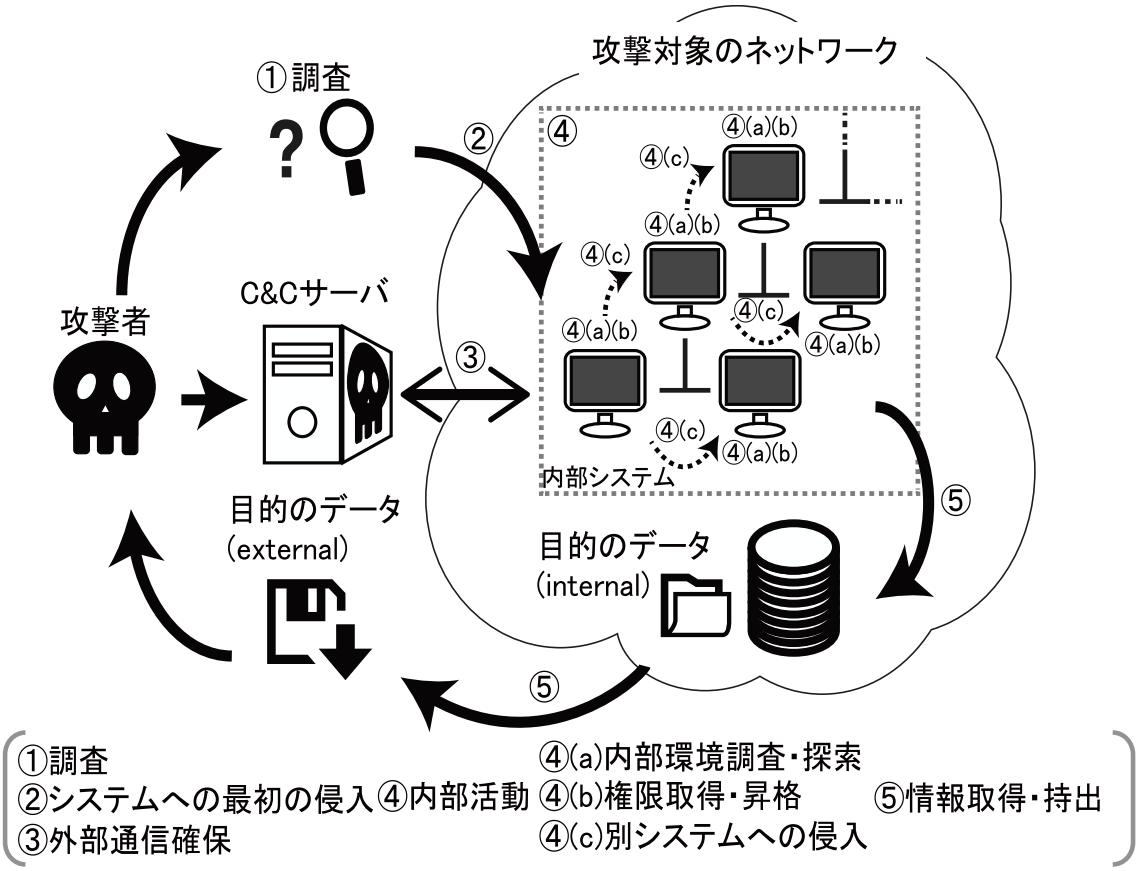

図 2.1: APT 攻撃概要図

攻撃者は対象にアクセスできる方法を調査する。最初に誰でもアクセス可能な情報を取得する。その後、個人情報の売買など方法を問わず対象に関わる情報を収集する。調査対象にされる人物の範囲として年長の指導者や研究者、経営の補佐が中心となる。主対象に接触できない場合は対象とする組織の別部門、協力会社などが侵入の対象となる。現在では Facebook や LinkedIn などの利用者が自身の趣味や経歴などからグループを作成し、交流を行うソーシャル・ネットワーキング・サービス（SNS）を多くのインターネットユーザが利用している。そして SNS を利用する際に自身の経歴や投稿、友好関係を全ユーザに対して公開しているユーザが往々にして存在する。そのため、SNS で一般公開されている情報を次段階での攻撃に利用されることが報告されている。

2. システムへの最初の侵入

   E-mail を用いて特定の個人に対してソーシャルエンジニアリングを行うことが最も多いとされている。ソーシャルエンジニアリングの侵入手段として、有名なスピアフィッシングを例に挙げると、従業員が業務相談会に出席する場合、攻撃者はその会議の講演者を騙ってメールを送信する。そしてメールに添付ファイルされている実行ファイルやハイパーリンクをユーザを実行することで不正な動作を行うエキスプロイトコードが起動され、対象のシステムに侵入される。前段階での調査によって情報量が多ければ多いほどこの攻撃の成功率は上昇する。

   スピアフィッシングの例として内閣サイバーセキュリティセンター（NISC）が 2012 年 1 月に公開した「標的型不審メール攻撃訓練」結果の中間報告 [8] によると、2011 年 10 月から 12 月に内閣官房等 12 の政府機関に所属している約 6 万名を対象に、事前教育を行ったうえで標的型不審メールを模擬したメールを送った結果、平均して 10.1 %のメールが開封されている。この攻撃はヒューマンエラーを利用したもののため、完全に防ぐことは難しい。

3. 外部通信手段の確保

   外部のネットワークから侵入されたシステムへの制御を可能とするためにバックドアを設置する。バックドアには攻撃者が遠隔で侵入されたシステムにコマンドを送信できる機能が含まれている。直接攻撃者がシステムと通信することはなく、Command and Control（C&C）Server を経由してコマンドを送信する。

4. 内部活動

   (a) 内部環境調査・探索

   攻撃者はユーザのアカウント上でコマンドインタプリタを用いて侵入されたシステムの環境情報とネットワークの構成などを調べる。また、情報取得のためにファイルシステムの探索を行う。攻撃者の内部探索の傾向として、横断的または縦断的に探索アルゴリズムのように動くと考えられる。そもそもこの時点

では何処にどんな情報があるか分からないためである。そして、コマンドを入力する間隔は内部のセキュリティシステムに検知されないように半日・1 日おきに行われる。これらの活動はスクリプト処理で自動化する場合もある。

(b) 権限取得・昇格

攻撃者はシステムの脆弱性を利用してエキスプロイトコードを利用して、ユーザアカウントから管理者アカウントへと権限の昇格を狙い、さらなる侵入へと繋げる。特に対象システムが Windows の場合、PwDump [9],cachedump [10] などのツールを用いて Security Account Manager（SAM）データベースやシステムレジストリからパスワードハッシュを入手し、別のシステムへの侵入を行う。これは Pass-the-Hash と呼ばれている。これは Windows のリモートデスクトップ接続や Single Sign On（SSO）のための仕様を利用したもののため、対策が困難である。

(c) 別システムへの侵入

内部調査で得られたネットワーク構成をもとに同ネットワーク内にある別システムへの侵入が行われる。侵入後はそのシステムの内部調査が行われ、対象とする情報が得られるまで 4 の活動が繰り返される。

5. 情報取得・持出

目的の情報を取得した後、外部に情報の持出を行う。ワークステーション、ファイルサーバなどのシステムから取得した目的のデータはパスワード付きの圧縮ファイルで分割され、少しずつメールや外部機器に退避される。一般的な方法として、ステージングサーバを用いて情報の転送を行う。また、転送するファイルには暗号化と圧縮を行い、転送されるファイルだけを見ても情報の持出の検知できないようにする。そして、退避後にステージングサーバにあるファイルを削除することで転送の痕跡を削除する。

2.2 APT 攻撃への対策と課題

前節で述べた活動段階を基に対策と課題について述べる。図 2.2 は、APT 攻撃のフロー図であり、APT 攻撃の対策と APT 攻撃の段階を対応付けたものである。

はじめに、調査段階では SNS や Official Site など一般公開されている情報へのアクセスと直接的または間接的な接触による情報収集を行っている。これはビジネスやプライベートでの社交活動と区別が付かないため、この段階で APT 攻撃を判断し対策をとることはシステムレベルでは不可能である。

次に最初の侵入段階への対策として URL フィルタや標的型メールフィルタを用いた対策が考えられる。メールに記載されている URL や添付されている実行ファイルなどから APT 攻撃を検知する。しかし、これらは攻撃パターンを学習することで効果を発揮する

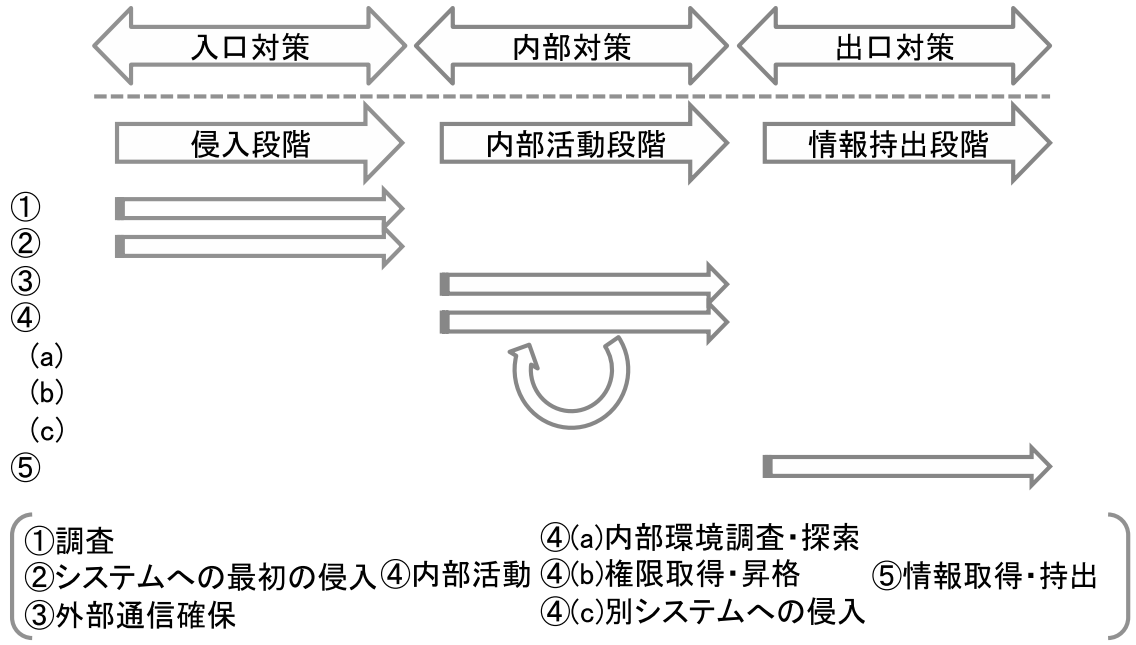

図 2.2: APT 攻撃フロー図

ものなので、その攻撃が初出であった場合防ぐことが困難になる。さらに、ゼロデイ攻撃やソーシャルエンジニアリングによる侵入はシステムレベルでは防げないため、システムへの最初の侵入段階で APT 攻撃を防ぐことは困難であるとされている。

外部通信手段の確保段階では外部との C&C 通信が行われる。通常外部と通信を行う場合痕跡がネットワーク・トラフィックとして記録されるため、アノマリ検知により異常なトラフィックとして検知することが可能である。しかし、C&C 通信は Internet Relay Chat（IRC）や Dynamic Domain Name Service（DDNS）を利用して通信を行うため、C&C 通信特有のトラフィックが検出しにくいという課題がある。

内部活動段階ではシステム内で通常運用されているユーザアカウントを用いて内部での情報探索活動を行う。さらに、そのアカウントで得られた情報を基に別のアカウントまたは上位の管理アカウントへの接続を行う。この時点での侵入検知システムとして以下のような研究が提案されている [11] [12] [13]。

Protecting Against Unexpected System Calls [11] ではプログラムによって呼ばれるシステムコールの出す順番を記録する。システムで使用する全てのプログラムに対してそれぞれ発生するシステムコールの順番を記録したデータセットを作成する。プログラムの実行の際に発生するシステムコールを照合することで、異常なプログラムによって発生するシステムコールを検知する。また、システムコールの順番を記録しているためコードインジェクションを検出することも可能である。危険なシステムコールに着目した Windows 向け異常検知手法 [12] では、ルールベース方式の検知とプログラムの振る舞いを組み合わせたアノマリ検知を行っている。はじめに、不正なプログラムの挙動により定義されたルー

ルをもとに、OS の重要なリソースにアクセスを行うクリティカルなシステムコールを検知する。その検知したシステムコールの発行履歴から、特徴ベクトルを作成し、サポートベクターマシンにより危険なシステムコールか否かを判断して検出する。

ファイルシステムの特性を活かした APT 攻撃検出に関する研究 [13] では、ファイルアクセスの移動距離から APT 攻撃の検出を行う Detection by Fragment Features（DFF）とファイル名からファイルアクセスの順序を用いて検出を行う Sequential Access Pattern（SAP）によるマッチング手法が提案されている。小粒度の時間間隔で起きたファイルアクセスから Bag of Pathname（BoP）という集合を作成し、DFF では、ユーザのシステム操作によって発生するファイルアクセスの断片特徴を BoP によって定義する。この BoP を Support Vector Machine の入力とすることでマッチングを行う。SAP では、検出するファイルアクセスのパターンを想定して記述し、そのパターンと BoP との間でパターンマッチングを行う。

しかし、情報探索活動は既存のファイアウォールや侵入検知システムなどを回避するために長期的かつ断続的に活動が行われる。そのため、ユーザアカウントに紛れた長期的かつ断続的な活動を行う攻撃者を検知しなければならないことが課題となっている。

別システムへの侵入の対策と課題として、Windows Server 2012 R2 と Windows 8.1 以降の OS ならば対策は出来る。新機能である新しいローカルアカウント・リモート管理・認証ポリシーの変更により Pass-the-Hash 攻撃の効果を軽減させることが可能となる。しかし、全てのドメインコントローラーを上記のシステムかそれ以降のバージョンに更新する必要があり、全ての Pass-the-Hash 攻撃を防げるわけではない。

情報取得・持出段階では、入手したデータを暗号化し、分割してステージングサーバを経由して攻撃者がデータをダウンロードする。対策としてネットワークトラフィックのアノマリ検知が用いられるが、データが暗号化されており、通信は外部通信段階と同様に IRC や DDNS などのプロトコルの通信に偽造されているため、この部分での検知が難しいとされている。

2.3 出口対策における情報漏えいについて

第 1.1 節より攻撃者は利益目的で対象の組織の情報を狙っている。そのため、組織の不利益になるような情報が第三者の外部組織に流出することが情報の漏えいにあたると考える。よって、情報漏えいの観点から APT 攻撃の対策について考察を行う。情報漏えいという観点から見ると、最初の調査段階から外部通信手段の確保までの段階は情報漏えいにあたらないため、本論文ではこれらの段階での対策は対象外とする。よって内部活動段階と情報取得・持出段階について考察を行う。

はじめに、内部活動段階は目的の情報が得られるまで、情報探索活動と別システムへの侵入活動が繰り返し行われる段階である。攻撃者が目的とする情報は機密性が高ければ高いほど経由するシステムは多くなる。また、この活動段階ではソフトウェアによる検知を避けるために長期的に活動が行われる。よって、この段階では時間に比例して流出する情

報の機密性が高くなっていくと考えられる。そのため、攻撃者の目的とする情報によっては、ある一定の時間を掛けるまで成果が得られないということが考えられる。そのため、できるかぎり早い時間で内部活動を検知できれば情報漏えいを防ぐことができる。

次に情報取得・持出段階は内部活動段階を経て目的のシステムに侵入が成功し、情報を取得してから持出を行う段階である。ここでは、第 [] 節より、取得した情報を暗号化し、IRC や DDNS などのプロトコルに偽造し、ステージングサーバに情報を転送する。このとき、IRC や DDNS などのパケットサイズが比較的小さなプロトコルで通信されているため、取得したデータのサイズに比例して転送時間が増加していくと考えられる。この手法の対策として、プロトコルのシグネチャによる検知やネットワークトラフィックの監視によるアノマリ検知が考えられる。よって、ステージングサーバに情報を転送している最中に攻撃を検知できれば、その情報の転送を防ぐことができる可能性がある。

しかし、これを情報漏えいという観点で見ると解決が困難な課題が生じる。この段階での APT 攻撃を検知したとしても情報流出を完全に止めたとはいえないからである。何故ならば検知したときの情報持出が最初の情報持出だと断定することができないためである。そして、APT 攻撃を検知できたとして、その通信が遮断されるまでに送られたデータは、攻撃者に渡ってしまうという問題がある。そのデータが分割されており RC4 や MUGI のような逐次暗号化が行われている場合を考える。このとき、攻撃者の手元に転送されてきた情報は、転送されたデータ分だけその情報が読み取り可能だからである。また、情報持出段階では情報を攻撃者が直接操作するシステムに存在してはないとはいえ、その情報の閲覧が可能な状態であることは確かである。以上より、情報漏えい対策という目的で情報取得・持出段階で検知することは困難であることがわかる。

よって本研究の目的である、APT 攻撃による情報流出を防ぐという観点から APT 攻撃の対策を考えると、情報持出よりも前の段階で攻撃者を検知し、活動を止めることが重要となる。

2.4 持続的な活動の検知における課題

前節より、情報持出よりも前の段階で APT 攻撃を検知するために、長期間活動を行う内部活動段階での検知に焦点をあてる。内部活動段階での特徴として、コマンドインタプリタを入力するときのアカウントがそのシステムを正規に使用しているユーザと同じであることと、既存のセキュリティソフトウェアによる検知を避けるために短期間での活動を行わないことが挙げられる。

短期間で大きな活動が行われないということは、内部活動はアノマリ・シグネチャ検知の両方に正規なパターンに含まれる誤差もしくはノイズとして扱われる。これを検知しようとすると、False Positive（FP）または False Negative（FN）の発生率を大幅に上昇させるしきい値に設定しなければならないため、結果として検出精度が著しく低下することが考えられる。よって、短期間でのログを用いたアノマリ・シグネチャ検知が困難であることがいえる。

表 2.1: 侵入期間の統計

調査対象 [組織数]	最小値 [月]	最大値 [月]	平均値 [月]	中央値 [月]
71	1	28	8.4788	7.0000

ここで内部活動段階の特徴であるコマンドインタプリタを用いるという点に着目する。通常コマンドインタプリタを用いて情報の探索を行うときには、ファイルシステムへのアクセスが行われる。そのときに、正規のユーザとしてアクセスしたログが残る。そのため、ログを長期間に渡り記録し続ければ正規のユーザに紛れた攻撃者の何らかの活動が検知できる可能性がある。しかし、ログをそのまま記録し続けることはシステムの運用として現実的ではない。例として、Mac OSX を終日利用したときのログサイズは約 1GB に及ぶ。それを 1 か月記録していくと、ログのサイズは約 30GB となる。これは一人あたりのログサイズであり、仮に 1000 人が Mac OSX を利用している場合、そのログのサイズの合計は約 30TB となる。

また、APT 攻撃におけるシステムへの侵入期間として、2011 年に McAfee が出した世界の 71 の企業、政府期間、非営利団体に対する APT による被害の過去 5 年間の調査結果をまとめたレポート [4] が参考資料として挙げられる。 [4] を参考に 71 の組織を対象とした侵入期間の統計情報を計算したものが [表 2.1] である。これによると、最短で 1ヶ月未満であり、最長で 28ヶ月、平均値が約 8.5ヶ月であり、中央値が 7ヶ月の攻撃が断続的に続いていることがわかる。この中央値である 7ヶ月をログの取得期間とすると、前段落より 1000 人規模の環境でログを全て記録したときのログサイズは 210TB となる。これがログをそのまま記録し続けることが現実的でない理由である。以上よりシステムの運用面から考えて、内部活動を検知するためにはログのどの部分を残し、記録し続けるかを考慮しなければならない。

第3章　ソフトウェアの操作とファイルアクセスログ（FAL）

3.1　ファイルシステムの構成と機能

はじめにファイルとは、コンピュータ上でビットによって構成される、意味をもつ情報の単位である。ファイルにはメタデータである名前、属性や権限といった情報が付与されている。そして、ファイルシステムはOperating System（OS）においてファイルを管理するための機構である。またファイルシステムは、OSにおいて情報を記録するために用いられる記憶装置を論理的に分割したパーティションと呼ばれる領域の上にマウントされるシステムである。そして、ディレクトリは記憶装置上のファイルの管理を階層構造で行うための機構である。ゆえにファイルシステムはディレクトリとディレクトリ上にマウントされるファイルシステムによって構成される。ファイルシステムとディレクトリの関係を図に表したものが図3.1である。

ディレクトリの特徴として1つのディレクトリには複数のファイルを格納することが可能であり、ディレクトリの中にディレクトリを入れることができる。これがファイルシステムにおいて、ファイルとディレクトリが階層構造になる理由である。また、階層構造は別名木構造とも呼ばれる。木構造において木の根にあたるディレクトリはルートディレクトリと呼ぶ。これはファイルシステムにおいて、ファイルまたはディレクトリを参照するときに基準となるディレクトリである。このルートディレクトリの名称は'/'である。

ファイルシステム上の指定したファイルまたはディレクトリにアクセスする場合にパスを利用する。パスとは、木構造におけるファイルまたはディレクトリの位置を表したものである。ルートディレクトリから指定したファイルまたはディレクトリまでの経路を'/'で区切って結合したひと固まりの文字列は絶対パスと呼ばれる。また、自身のディレクトリを'.'で表すことができ、親のディレクトリを'..'で表すことができるため、'.'や'..'から指定したファイルまたはディレクトリまでの経路を'/'で区切って結合したひと固まりの文字列は相対パスと呼ばれる。

ファイルシステムの機能として、ファイル/ディレクトリの操作・検索が挙げられる。ファイル/ディレクトリ操作とはファイルシステム上のファイル/ディレクトリにアクセスする操作であり、その操作の種類や機能については次節で扱う。そして、ファイル/ディレクトリの検索とはファイル/ディレクトリの名前や属性などの情報をもとに探索を行い、そのファイル/ディレクトリの位置を探す機能である。これらの機能を用いることで、OSによるデータ操作が実現されている。

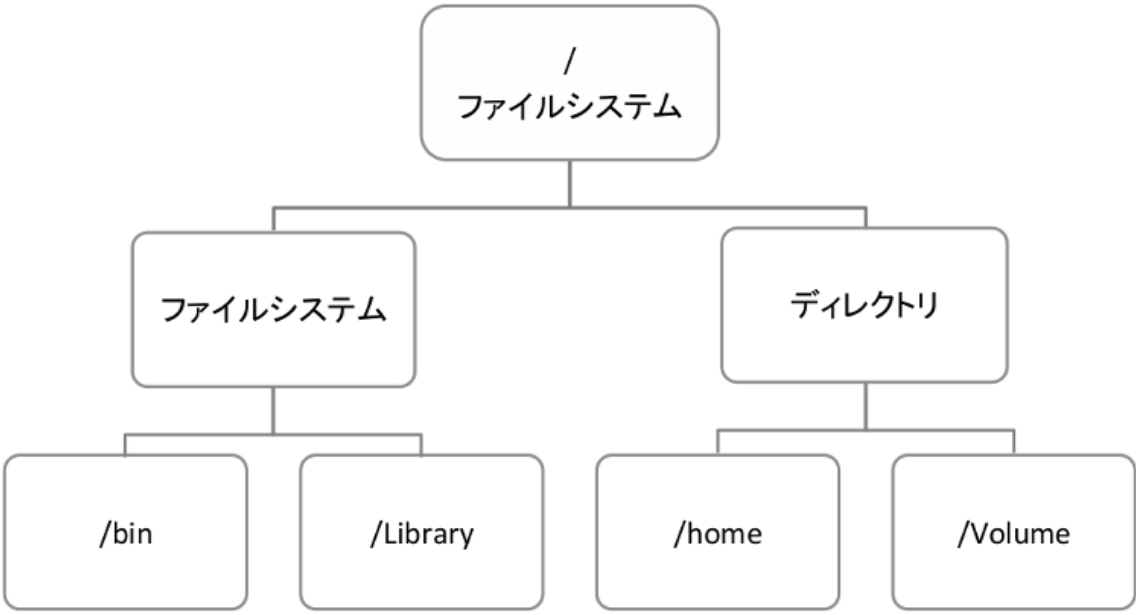

図 3.1: ファイルシステムとディレクトリの関係

3.2 ファイルシステムのファイル/ディレクトリ操作

図 3.2 は、ソフトウェアがファイル/ディレクトリを操作するときに発生するスタックである。まず、ソフトウェアの操作が発生するとき、それに付随してファイル/ディレクトリ操作が発生する。このとき、アプリケーションまたはライブラリがファイル/ディレクトリに対してアクセスを行う。これはシステムコールによって実現される。システムコールは、ファイルシステムへのアクセスを行う。このとき、Virtual File System（VFS）を経由して、ファイルシステムへアクセスを行う。VFS は、種々のファイルシステムへの単一の入力機構を提供するため、アプリケーションまたはライブラリがファイルシステムの種別を気にせずにファイル/ディレクトリへのアクセスが行えるようにしている。そして、デバイスドライバによってファイルシステムはファイル/ディレクトリを記録しているストレージへのアクセスにすることが可能となる。

ファイル/ディレクトリ操作とは、ファイルシステム上のファイル/ディレクトリにアクセスを行う操作のことである。具体的な操作としてファイル/ディレクトリの読み込み、書き込み、作成、名前と位置の変更、削除が挙げられる。

はじめにファイル/ディレクトリの読み込みとは、ファイル/ディレクトリのメタデータである名前、属性、権限などの情報を取り出すことである。ファイル/ディレクトリの作成とは、ファイルシステム上のルートディレクトリ下の指定した位置に新しいファイル/ディレクトリを生成することである。ファイル/ディレクトリの名前の変更とは、ファイル/ディレクトリのメタデータである名前を別の名前に変えることである。位置の変更とはファイル/ディレクトリを指定したパスの下に再配置することである。つまり、変更対象のファイル/ディレクトリは指定したディレクトリの子ファイル/ディレクトリとなる。

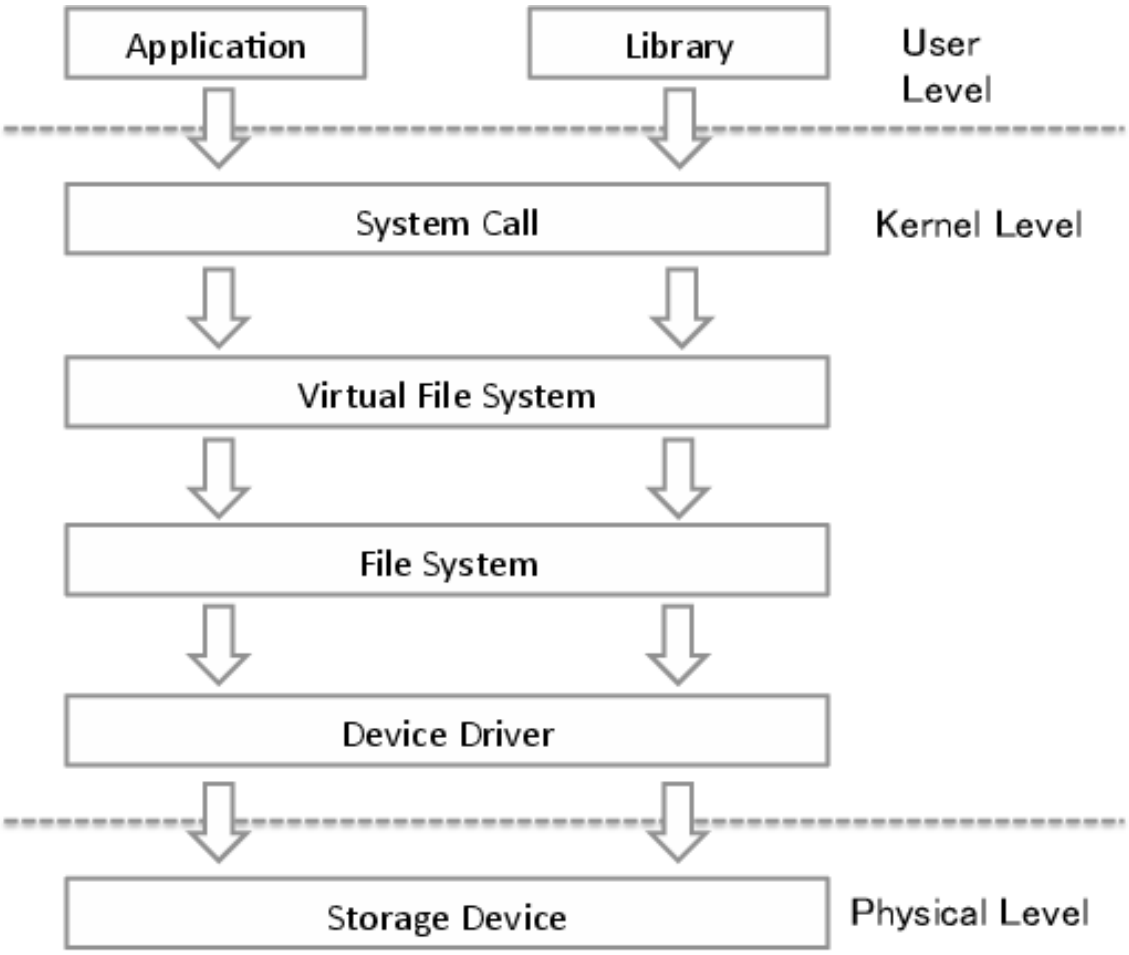

図 3.2: ファイルシステムとシステムコールの関係

ファイル/ディレクトリの削除とは、指定したファイル/ディレクトリをファイルシステム上から消去することである。ただし、指定したディレクトリが子ファイル/ディレクトリを持つ場合、その子ファイル/ディレクトリも同様に消去される。

ファイルシステムにおいて（上記の）ファイル/ディレクトリ操作はシステムコールによって実現される。システムコールとは、カーネルにコア部分の処理の依頼を行い結果を返す関数である。一般に、OS においては OS のカーネル以外のプログラムが物理メモリの確保や開放といったシステムのコア部分への操作を直接行わないためである。前段落で示したファイル/ディレクトリ操作を実現しているシステムコールとして open、read、write、creat、rename、remove が挙げられる。

はじめに open とは、指定したファイル/ディレクトリのパスを引数にファイルディスクリプタを返す関数である。ファイルディスクリプタとは OS がプログラムによるファイル/ディレクトリの参照を 0 からの整数で表した識別子である。よって、open によって得られたファイルディスクリプタを指定することでファイル/ディレクトリに対する操作が出来るようになる。 read とはファイルディスクリプタを引数に読み込んだファイル/ディレクトリのバイト数を返す関数である。そして、ファイルディスクリプタで参照されるファイル/ディレクトリのバイト列をバッファに格納する。write とはファイルディスクリプタとバッファを引数にして、ファイルディスクリプタに書き込まれたバイト数を返す関数である。そして、バッファに格納してあるバイト列をファイルディスクリプタに書き込みを行う。creat とは新しくファイルを作成して、ファイルディスクリプタを返す関数である。rename とはパスを引数にして、そのパスの実体であるファイルの名前を変更または、そのファイルのパスを変更する関数である。remove とは、指定したパスを引数にして、そのパスが示すファイル/ディレクトリ名を削除する関数である。remove で指定したファイルへのリンクがない場合、ファイルシステムからそのファイルは削除されることになる。

3.3 ソフトウェアにより生成されるログの種別と構成

ソフトウェアによる生成されるログは大別して 2 種類存在する。コンピュータが動作するために実行されるシステムソフトウェアによって生成されるシステムログとユーザが目的とする情報処理を行うために実行されるアプリケーションソフトウェアによって生成されるアプリケーションログである。これらのログはシステム/アプリケーションソフトウェアが正常に動作をしているかを管理するために用いられる。そのため、コンピュータの障害が起きたときにこれらのログをもとに調査を行い、問題の解決が行われる。

昨今では以上の目的だけでなく監査のためにログが活用されている。節 1.1 より、サイバー攻撃に対するセキュリティ対策の重要度は年々増加し続けている。そのため、サイバー攻撃が起きたときに原因を究明するため、ログを保持することは企業にとって必須といえる。これらのログはいつ、誰が、何をしたか、どうやって行ったかを記録している。サイバー攻撃が起きた際に、被害情報を分析するためにログの保存・保持を行うことはフォレンジックと呼ばれている。2015 年 1 月 9 日に施行されたサイバーセキュリティ基本

法第 30 条 [14] より、特定重大事象が起きた際の行政機関におけるサイバーセキュリティ戦略本部への資料提供が義務化されたことからもフォレンジックの必要性が高まっていることがわかる。また、ログには重要度が設定されており、必要に応じてログの出力を調整できる。よってログをもとに障害の調査を行う場合に、原因の特定するための絞込を短時間で行うことができるようになる。しかし、この方法では、あらかじめシステム/アプリケーション側で想定されている障害でなければ、発見することが困難である。

システムログは OS の起動・再起動・終了といった起動管理やユーザによるログイン・ログアウトやファイルへのアクセスといった認証・セキュリティの管理、ハードウェア・BIOS・OS の動作・稼働状況の管理を行うために各システムソフトウェアの動作時のイベントを記録している。そのため、システムログを解析することで、システム/ハードウェア障害の検知やその原因の調査などを行うことができる。

アプリケーションログは、書類作成、メールの読み書きなどのユーザがアプリケーションソフトウェアを介して操作したときの各アプリケーションソフトウェアの動作時のイベントを記録している。そのため、アプリケーションログを解析することにより、ユーザのシステム上での活動を監視し、ログからどんな活動を行ったかを特定することができる。

セキュリティ管理のためのシステムログとして、監査システムログと呼ばれるものがある。監査システムとは、ユーザによる情報処理の操作内容とそれに付随して発生するシステムの動作を記録するシステムである。そのため、監査システムログはシステムログに加えて、アプリケーションソフトウェアによって呼び出されるシステムコールを記録する。特に、監査システムログにおいてファイルアクセスが起きたときに、そのファイルアクセスを実行するシステムコールに関して、いつ、誰が、どんなソフトウェアを使って、何のシステムコールを呼びだしたかを記録したものはファイルアクセスログ（File Access Log：FAL）と呼ばれている。

3.4 FAL におけるエントリの構成と関係性

ファイルアクセスログは、主要 OS において実装されている監査システムによって取得できるシステムログである。Windows では、ローカルセキュリティポリシーが Windows シリーズにおいて特定のエディションで使用可能である。Mac OSX(BSD) と Linux では Open Basic Security Module(OpenBSM)・Linux Audit が標準で使用可能である。これらは共通して、ファイルアクセスの際に発生するシステムコールをフックすることで全てのファイルアクセスログを取得している。

Linux で/home/shadowman から/etc/ssh/sshd_config を cat コマンドで閲覧したときのファイルアクセスログの例は以下のようになる [15]。

Linux Audit Sample

```
・type=SYSCALL msg=audit(1364481363.243:24287): arch=c000003e
syscall=2 success=no exit=-13 a0=7fffd19c5592 a1=0
```

```
a2=7fffd19c4b50 a3=a items=1 ppid=2686 pid=3538 auid=500
uid=500 gid=500 euid=500 suid=500 fsuid=500 egid=500
sgid=500 fsgid=500 tty=pts0 ses=1 comm="cat" exe="/bin/cat"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
key="sshd_config"
・type=CWD msg=audit(1364481363.243:24287):  cwd="/home/sh
adowman"
・type=PATH msg=audit(1364481363.243:24287): item=0
name="/etc/ssh/sshd_config" inode=409248 dev=fd:00 mode=0100
600 ouid=0 ogid=0 rdev=00:00 obj=system_u:object_r:etc_t:s0
```

Linux Audit のログは type=から始まるレコードで構成される。これらのレコードは監査システムが取得可能なコマンド入力時に発生するユーザ情報が全て含まれているため、ユーザの活動の詳細を知るために有用なものである。また、このレコードを 1 つの固まりにしたものがログとなる。そして、複数のログが結合して Audit ログが形成される。以下は Audit ログがそれぞれ意味しているものをまとめたものである。

Linux Audit Sample First Record

・type=SYSCALL
このメッセージの後に続くログはシステムコールのパラメータが入る。
・msg=audit(1364481363.243:24287)
システムコール発生時の timestamp と Audit が生成する ID をコロン：でつなげたものである。この timestamp と ID により同レコードであることが判断される。
・arch=c000003e
CPU のアーキテクチャ情報が入るフィールドである。"c000003e"は CPU のアーキテクチャが x86_64 であることを示している。また Audit ログを検索するコマンド ausearch のオプション-i または--interpret によりこのフィールドの値を人間可読な値に変換することができる。
・syscall=2
カーネルに送られるシステムコールの番号が入り、このアーキテクチャにおいて"2"はシステムコール open を示すものである。システムコール番号とシステムコールの対応はアーキテクチャによって異なり、一般に以下のどちらかの方法で知ることができる。

1. /usr/include/配下の unistd.h or unistd_64.h を見る。
（/usr/include/asm/unisted_64.h or/usr/include/asm-generic/unistd.h)
2. ausyscall コマンドのオプション--dump を用いる。

・success=no
上記のシステムコールの成否を示す。システムコールの失敗した場合は"no"が入り、成功した場合は"yes"が入る。

・exit=-13
システムコールによって返される終了ステータスが入る。
・a0=7fffd19c5592 a1=0 a2=7fffd19c4b50 a3=a
システムコールによって最初に用いられる４つの引数であり、これらの値は
システムコール毎に異なる。
・items=1
このイベントのパスレコード番号が入る。
・ppid=2686
Parent Process idを示す。"2638"はbash processを表している。
・pid=3538
Process idを示す。"3538"はcat processを表している。
・auid=500
Audit User idを示しており、ログインIDと同等である。
よってログインIDを切り替えた場合この番号が変わる。
・uid=500
User id を示しており、Audit process を実行させた user の id である。
ausearch -i --uidによってuser名を知ることができる。
・gid,euid,suid,fluid,egid,sgid,fsgid
これらはそれぞれ、gid(group id),euid(effective user id),
suid(set user id),fluid(file system user id),
egad(effective group id),sgid(set group id),
fsgid(file system group id)であり、Audit processを実行させた
userのidを示している。
・tty=pts0
Audit processを実行させた端末のidを示している。/dev/pts/0に接続しているユーザがこのAudit processを実行している。
・ses=1
Audit processが実行されたセッションidを示している。
・comm="cat"
Audit processが実行された（(auditのイベントのトリガとなった）コマンドライン名が入る 。ここではファイルを連結して標準出力に出力する"cat"コマンドが入る。
・exe="/bin/cat"
Audit processが実行されたコマンドのpathを示している。
・subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
Audit processの実行時にラベルとして記録されたSELinuxのコンテキストを表示する。
・key="sshd-config"

実行者が定義した文字列を示している。この文字列を用いて ausearch で該当するログのみを抽出することが可能となる。

Linux Audit Sample Second Record

・2 つ目のレコード
type="cwd"
"cwd"は Current Working Directory を表し、最初のレコードで実行された作業ディレクトリのパスを記録するために用いる。
msg=audit(1364481363.243:24287)
システムコール発生時の timestamp と Audit が生成する ID をコロン：でつなげたものである。
cwd="/home/shadowman"
システムコールが呼び出されたときの作業ディレクトリのパスである。

Linux Audit Sample Third Record

・type="PATH"
システムコールに含まれる全てのパス型レコードが含まれる。
・msg=audit(1364481363.243:24287)
システムコール発生時の timestamp と Audit が生成する ID をコロン：でつなげたものである。
・item=0
type=SYSCALL によって参照される項目のうち、現在のレコードがある項目を 0 origin で表している。（複数パスが参照される場合の識別子である。）
name="/etc/ssh/sshd_config"
引数としてシステムコールに渡されたファイル・ディレクトリの完全パスを表す。
・inode="409248"
イベントに記録されたファイル・ディレクトリに関連付けられている inode 番号を表す。（find / -inum "huge" -print で inode 番号から full path に変換できる。）
・dev=fd:00
デバイスのメジャー番号とマイナー番号を示している。
・mode=010066
ファイルへのアクセス権を表す。"010066"を Unix パーミッション表記に直すと"-rw-------"であり、root user のみ読み書きができることを示す。（cat コマンドによる open システムコールが失敗した理由はこのファイルにふられた実行権限によるものである）
・ouid=0
owner's user id を示している。

・ogid=0
owner's group id を示している。
・obj=system_u:object_r:etc_t:s0
Audit process の実行時にラベルとして用いた SELinux のコンテキストを表示している。

前述したとおり、FAL はシステムコールが起きたときに発生したイベントを記録するものであるため、あるソフトウェア操作が行われたときに複数のレコードが生成される場合が存在する。監査システム側で同じイベント、つまり単体のソフトウェア操作として扱われた場合は、同じ audit id を持つため判別が可能になるが、監査システム側で異なるイベント、つまり複数のソフトウェア操作として扱われる場合は、生成される複数のレコードがそれぞれ異なる audit id を持つため、audit id のみではレコード間の関係性、つまりソフトウェアの操作とファイルアクセスとの対応を一意に判別することができないという特徴がある。

FAL のレコード間にある関係性を一意に判別する手法について考える。FAL はレコードが時系列順に記録されたログである。そのため、FAL を連続的に解析することにより、FAL における時間的な変化から、ソフトウェア操作とファイルアクセスとの対応を明らかにすることが可能となる。これを実現するために条件が 2 つ必要となる。1 つ目は、ファイルアクセスが発生したときに記録されるレコード同士の関係性の意味付けを行う必要がある。2 つ目は、FAL とソフトウェア操作によって異なるファイルアクセスの傾向や頻度などの定性的な情報を対応させる必要がある。

前述した 2 つの条件は、1 つのシステムコールに対応して記録されるレコードを単体で解析・処理を行うだけでは満たすことが出来ない。そのため、FAL においてレコード間の関係性とファイルアクセスの定性的な特徴の意味付けを行うためには、FAL の構造化が必要となる。

3.5 ユーザによるソフトウェア操作と FAL との関係

情報技術が社会様式や生活様式の基盤として確立された高度情報化社会において、ソフトウェアを用いた情報処理は公私問わず日常的に行われるようになっている。そのため Web の閲覧、メールといった情報サービスだけでなく、行政や銀行などの業務の大多数がソフトウェアの操作によって行われている。

これらの日常的にユーザが使用しているソフトウェアの操作には、3.3 節で示したとおり、システム/アプリケーションログが記録される。ここで、ソフトウェアの操作の際に記録されるファイルアクセスログに着目する。ファイルアクセスログを見ることでユーザによるソフトウェアの操作を追跡することができるからである。その根拠として、ソフトウェアは補助記憶装置にインストールされ、ソフトウェアの実行時に補助記憶装置からインストールされたファイルとキャッシュファイルが読み込まれる。また、ソフトウェアの

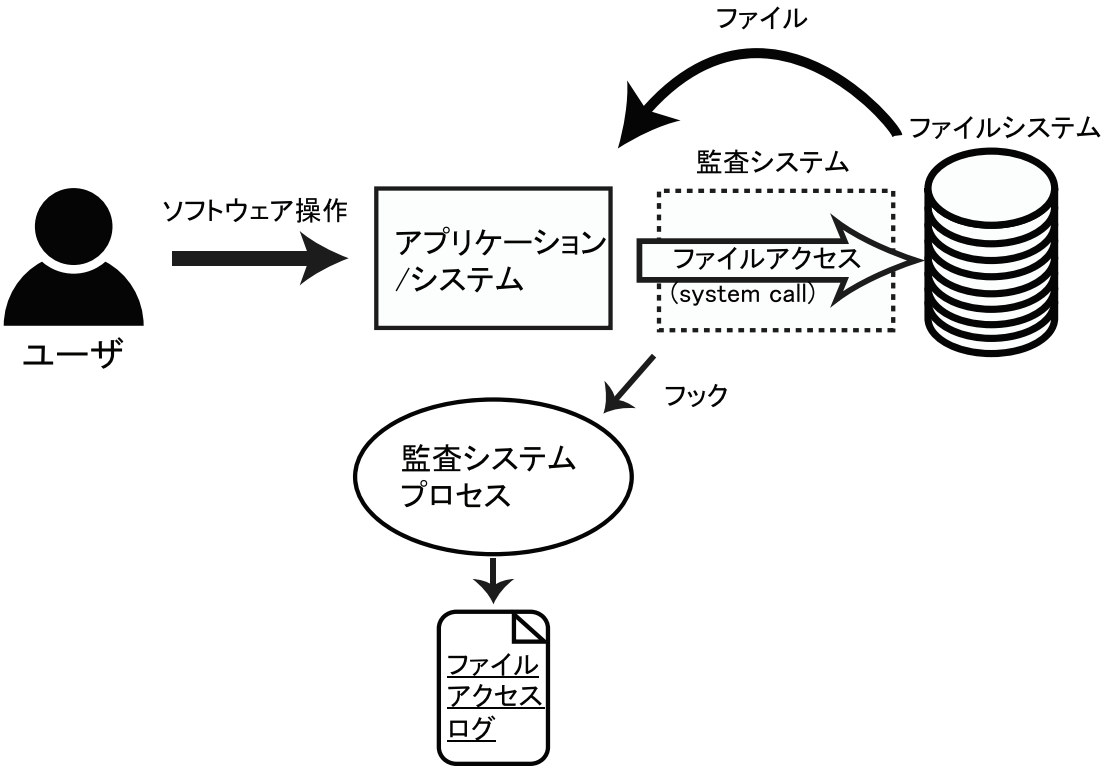

図 3.3: ユーザのソフトウェア操作と FAL

操作時に発生するファイルとキャッシュファイルにおいても、補助記憶装置に記録されるためである。よって、ユーザによるソフトウェアの操作の際にはファイルシステムへのアクセスが発生する。ここで、キャッシュファイルとは、ソフトウェアが実行されるときに一時的な処理のために記録される、または読み込まれるファイルである。そのため、ソフトウェアの操作によって作成されるファイルと同様にユーザのシステム上の活動において密接な関係があるファイルである。

3.2 節より、ファイルシステムへのアクセスつまりファイル/ディレクトリ操作はシステムコールによって実現されている。そして、システムコールがソフトウェアによって呼び出されたとき、それを監査したものが FAL である。 3.4 節の FAL の構成と関係性から、FAL を持続的な記録し解析を行うことでユーザのソフトウェア操作と FAL の対応関係が把握出来るようになる。この対応関係を図示したものが図 3.3 である。また、FAL の時系列によって蓄積されるファイルシステムへのアクセス位置の変移を解析することで、そのユーザのソフトウェア操作によるファイルシステムへのアクセスの傾向がわかる。そのため、ユーザのファイルアクセスの傾向を記録し続けることで、そのアクセス傾向に大きな変化が生じた際にそれを検知することが可能となる。

2.1 節より、システムに侵入した攻撃者はユーザアカウント上でコマンドインタプリタを用いて、ファイルシステムの探索を行う。求めているファイルがどこにあるかわからないためである。そして、業務のためにコンピュータを利用するユーザはその業務に必要

な部分がファイルアクセスの大部分を占めると考えられる。そのため、ファイルシステムにおけるアクセスの範囲は攻撃者の方が広く、ユーザの方が狭いことが想定できる。よって、攻撃者が行うファイルシステムの探索は、侵入を受けたユーザのファイルアクセスの傾向に変化を生じさせる。ゆえに、ユーザのファイルアクセスの傾向からユーザの異常活動を検知するだけでなく、攻撃者の活動を検知することも可能だといえる。

以上より FAL を解析することで、ユーザのソフトウェア操作の特定または、ユーザのソフトウェア操作に紛れて行われる非日常的な攻撃者の活動を検知することができる。

3.6 FAL の持続的な記録と構造化について

前節より、FAL の持続的な記録を行い、それを解析することで、ユーザのソフトウェア操作における異常検知及びユーザのソフトウェア操作に紛れて行われる非日常的な攻撃者の活動の検知を行うことが可能となる。 2.4 節より、攻撃者はシステムへの侵入期間は最短で 1ヶ月未満であり、最長で 28ヶ月、平均値が約 8.5ヶ月であり、中央値が 7ヶ月となっている。また、攻撃者は IDS による検知を回避するために、コマンド入力の間隔を長くとり、探索活動を行う。ファイルシステムへのアクセスの傾向によって、この探索活動を検知する場合、そのシステムをユーザのみが長期的に使用しているログが必要となる。これらを踏まえると、攻撃者の長期的な内部活動を検知するためには、年単位で FAL を記録することを想定しなければならない。

3.4 節で挙げた FAL のログのサイズは 594byte である。このログはソフトウェアの操作におけるファイルアクセス 1 回に対して記録されるものである。筆者が常用しているシステムの read/write の発行回数をもとに 1 日のファイルアクセスを 40 万とした場合、1 年間のデータ量は 87Gbyte に及ぶ。この FAL を 1000 人分年単位で記録する場合、87TByte にも及ぶ巨大なストレージが必要となる。FAL は内部活動を詳細に分析する上で必要不可欠なデータであるが、全てのログを生ログの状態で記録し続けるのは困難であるといえる。

FAL のレコードは全て独立して蓄積されていくものであるため、ログ単体で解析を行うだけでは、ユーザに紛れて行われる攻撃者の長期的な内部活動による潜在的なファイルアクセスの特徴を顕在化させることはできない。そも、ファイルアクセスの特徴とはファイルアクセスに対して、誰がどのファイルにどんな操作を行ったかを定性的または定量的に表したものだといえる。FAL における定量的な特徴とは、ファイルアクセス数・システムコールの種類やユーザ毎のアクセス数の差などである。これらは、レコードのパラメータを解析することで表現することが可能である。FAL における定性的な特徴とは、集中的・分散的・局所的などと表現できるユーザ毎のファイルアクセスの頻度及び範囲や定期的・周期的・非定期的などと表現できるユーザ毎のファイルアクセスの時間間隔などで表現されるファイルアクセスの傾向であると考える。よって、これを表現するためにはレコードのパラメータを用いて解析するだけでなく、レコードの前後関係や全レコードに対するレコード単体の相対的な位置などを定義する必要がある。

以上より、FAL からユーザのファイルアクセスのアノマリ検知を行うためには、FAL を

現実的に持続的な記録が可能となるサイズまで小さくすることと、FAL からユーザのファイルアクセスの傾向をシステム上で計算可能な表現を行うことが必要となる。そのためには、 3.4 節で挙げたとおり、ファイルアクセスが発生したときに記録されるレコード同士の関係性の意味付けと、FAL とソフトウェア操作によって異なるファイルアクセスの傾向や頻度などの定性的な情報を対応させる必要がある。これは FAL を構造化させることによって可能だと考える。

第4章　Tree Structured Log（TSL）の提案とその特徴及び演算

4.1　TSLの概要

3.6節よりファイルアクセスが発生したときに記録されるレコード同士の関係性の意味付けと、FALとソフトウェア操作によって異なるファイルアクセスの傾向や頻度などの定性的な情報を対応させるのためにFALを構造化させる。ここで、FALはソフトウェアの操作によって発生するファイルアクセスがトリガとなって記録されるログであるため、FALの誘因となるファイルアクセスに着目する。ファイルアクセスはファイルシステム上のファイルに対して読込・書込などの操作を行うことである。また、ファイルの管理を行うための構造として、木構造型のファイルシステムが存在する。木構造型のファイルシステムの特徴として3.1節より、パスを用いてファイル/ディレクトリの相対的または絶対的な位置を表現することが可能な点が挙げられる。よって、FALを木構造に写像させることにより、FALのエントリにおけるファイル/ディレクトリパスから、ファイルアクセスにおけるユーザのアクセス範囲を表現することが可能となる。以上より、FALをTree Structured Log（TSL）に写像させることを提案する。

TSLとは、FALのエントリを基にファイルアクセスを表現する木構造型のデータ構造である。以下TSLの構造に関する話：

FALの構造化の例を図4.1で示す。ここで作成されるTSLをTSL:Aとする。FALのパースを行ったあとに ['file path', 'system call' , 'other attribute'] で表現できる [[/foo,read,...], [/foo/bar,write,...], [/foo/baz,creat,...], [/hoge/hogehoge,trunc,...]] という4つのログが得られたとする。このログを順にTSL:Aに記録していく。最初に、木構造におけるルートを作成する。これがTSL:Aの初期状態となる。ルートは親を持たないノードである。また、初期段階ではシステムコールやアクセス数などの名前以外のラベルを持たないノードである。これを空のノードと定義する。空のノードを作成後に、ログからノードを作成していく。処理を行うときは'/' が初期の操作位置となる。

はじめに、[/foo,read,...] の処理を行う。初期位置は'/' である。ファイルパスを/で区切ると'/','foo' の2つのファイル/ディレクトリ名に分かれる。このときに、分割されたファイル/ディレクトリ数をこのファイルパスの深さと定義する。この深さ数の分だけTSL:Aのノードに対象に親から子へと移動しながら処理を行う。'/' は既にTSL:Aに存在しているノードであり、このあとに'foo' というファイルが続くため、この位置にあるノードに操作は行わない。その後'foo' というファイル名の処理に移る。fooはTSL:Aの'/' の子部

分にはないノードであるため、'/' の子部分にファイル名が'foo'、属性がファイルアクセス:1、システムコール:read、etc：... というノードを新しく生成する。

次に [/foo/bar,write,...] の処理を行う。ファイルパスを/で区切ると'/','foo','write' の 3 つのファイル/ディレクトリ名に分かれる。'/','foo' は既に TSL:A に存在しているノードであり、このあとに'write' というファイルが続くため、'/','foo' のノードに操作は行わないが、次の処理のために'foo' にポインタを移動させる。'bar' は TSL:A の'/foo' の子部分にはないノードであるため、'foo' の子部分にファイル名が'bar'、属性がファイルアクセス:1、システムコール:write、etc：... というノードを新しく生成する。[/foo/baz,creat,...] の処理も同様にして、'foo' の子部分にファイル名が'baz'、属性がファイルアクセス:1、システムコール:creat、etc：... というノードを新しく生成する。

最後に、[/hoge/hogehoge,trunc,...] の処理を行う。ファイルパスを/で区切ると'/', 'hoge', 'hogehoge' の 3 つのファイル/ディレクトリ名に分かれる。'/' は既に TSL:A に存在しているノードであり、このあとに'hoge' というファイルが続くため、この位置にあるノードに操作は行わない。'hoge' は TSL:A の'/' の子部分にはないノードである。また'hoge' の子部分に'hogehoge' というファイルが存在する。そのため、'/' の子部分にディレクトリ名が'hoge'、属性なしの空のノードを作成する。その後、ポインタを'/' から'hoge' に移す。'hogehoge' は TSL:A の'/hoge' の子部分にはないノードであるため、'/hoge' の子部分にファイル名が'hogehoge'、属性がファイルアクセス:1、システムコール:trunc、etc：... というノードを新しく生成する。

4.2 TSL の加算

TSL における加算処理とは、ノード及びラベルの結合を行うことである。ここで足される側の TSL を Augend TSL として、足す側の TSL を Addend TSL とおく。このとき、ノードの結合は、Addend TSL にのみ存在するノードを Augend TSL に新しくノードを作成することである。また、ラベルの結合は、両方の TSL のラベルにおいてパラメータであればその値の和を計算し、それが、キャラクタであれば、該当するラベルの属性のデータ列に情報を追加することである。

演算方法の具体例を図 4.2 と図 4.3 を用いて記述する。はじめに、ログ [[/foo/bar,read,...], [/foo/baz,write,...], [/hoge,trunc,...], [/hoge/hogehoge,read,...]] で与えられる TSL が図 4.2 の TSL:B である。このとき、TSL:B を Addend TSL、TSL:A を Augend TSL としたときの TSL:A + TSL:B つまり TSL:A+B を計算する。Addend TSL の各ノードを'/' から順番に足していく。ただし、TSL:A+B の初期状態は TSL:A である。ルートノードである'/' はラベルを持たないノードのため、加算処理は行われない。よって'/' の子ノードにあたる'hoge' ノードの加算処理に移る。'hoge' ノードは TSL:A と TSL:B のどちらにも存在するノードであり、TSL:B の'hoge' ノードのラベルが空でないため、加算を行う。TSL:A の'hoge' ノードは空のノードのため、TSL:B の'hoge' ノードのラベルがそのまま TSL:A+B の'hoge' ノードとなる。その後、'hoge' の子ノードにあたる'huga' ノードの加算処理に移る。'huga'

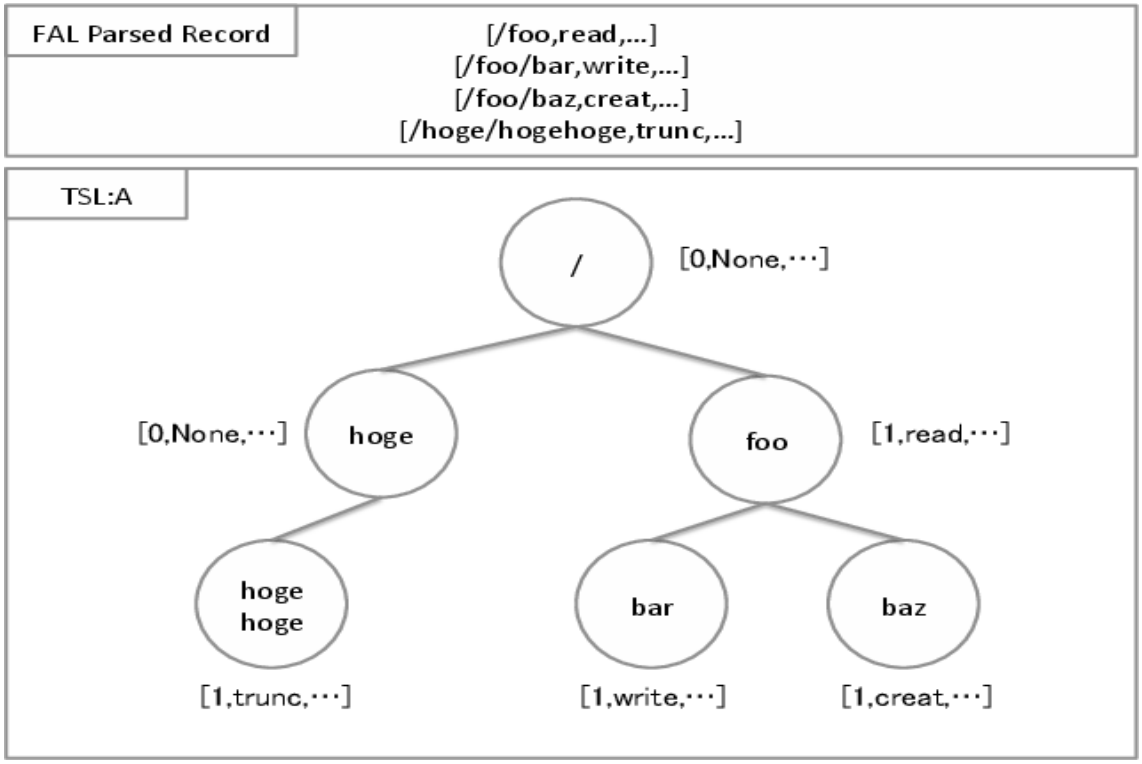

図 4.1: TSL 構造化例

ノードは TSL:A の’hoge’ の子ノードにはないノードである。そのため、’/hoge’ の子部分にファイル名が’huga’、属性がファイルアクセス:1、システムコール:read、etc：... というノードを新しく生成する。TSL:B の’hoge’ ノード以下の全てのノードの加算処理が終わったため、同じ深さの’foo’ ノードの加算処理に移る。TSL:B の’foo’ ノードは空のノードであるため、’foo’ の子ノードである’bar’ と’baz’ ノードの加算処理に移る。’bar’ ノードは TSL:A と TSL:B のどちらにも存在するノードであり、TSL:B の’bar’ ノードのラベルが空でないため、加算を行う。ここで TSL:A の’bar’ ノードのラベルも空でないため、そのラベルである [1,write,...] について順にパラメータの加算と追加を行う。TSL:B の’file access count’ が 1 のため、TSL:A+B の’file access count’ は 1+1=2 となる。TSL:B の’system call’ が’read’ のため、TSL:A+B の’system call’ は’write, read’ となる。他に属性があれば同じように追加していく。また、’baz’ ノードの加算処理も同様にして行う。以上の処理によって作成された TSL:A+B が図 4.3 である。

4.3 TSL の減算

TSL における減算処理とは、加算処理と同様にノード及びラベルの結合を行うことである。ノードの結合は、Addend TSL にのみ存在するノードを Augend TSL に新しくノードを作成することである。また、ラベルの結合は両方の TSL のラベルにおいてパラメー

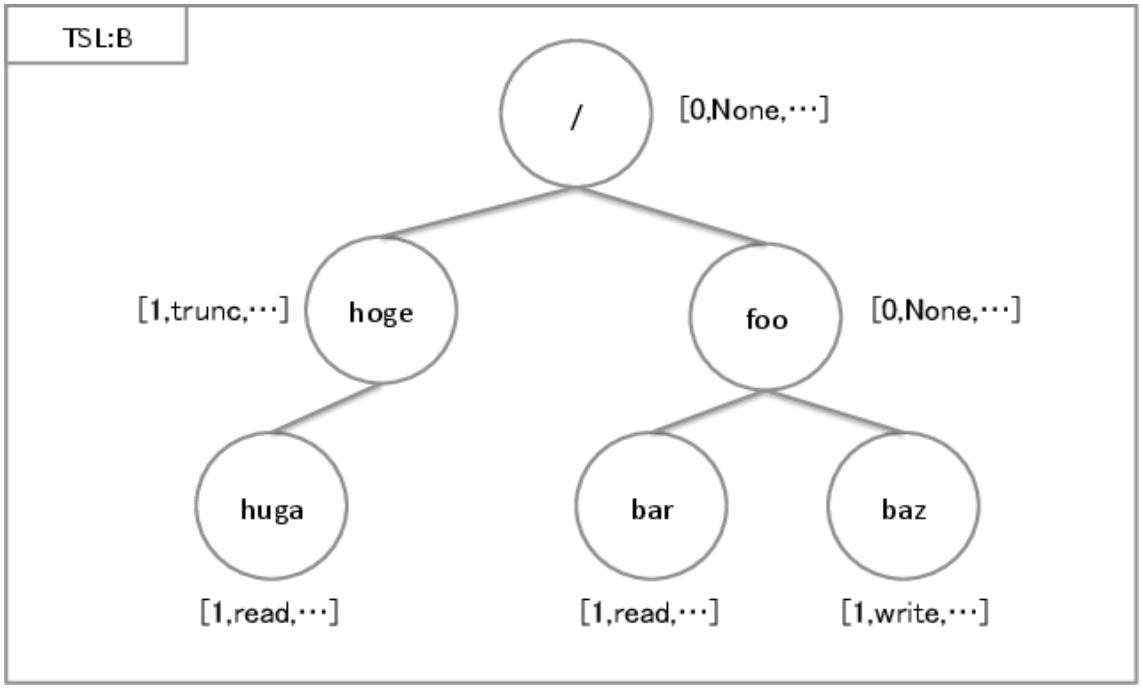

図 4.2: TSL:B

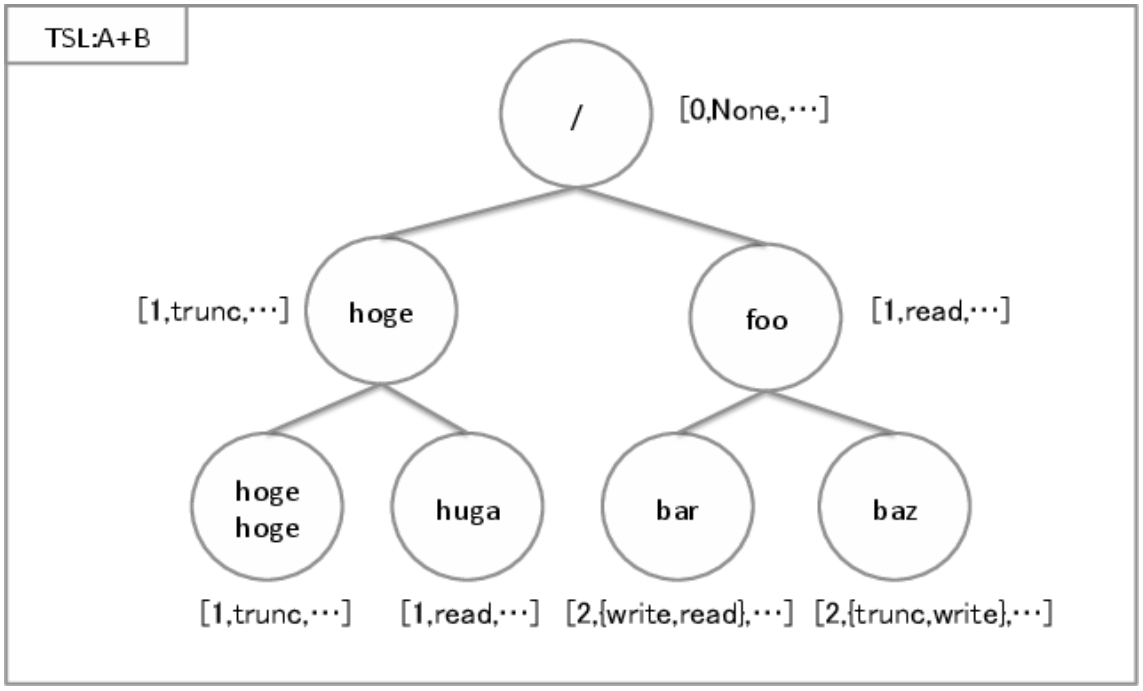

図 4.3: TSL:A+B

タであればその値の差を計算し、それが、キャラクタであれば、該当するラベルの属性のデータ列に情報を追加することである。このとき、パラメータは負の値をとることがある。これを anti node と定義する。

anti node は Addend TSL のあるノードのラベルが持つパラメータが Augend TSL のあるノードのラベルが持つパラメータよりも大きい、または Addend TSL のみが持つノードを表したものである。これは、TSL 同士を比較したときに得られる情報として、ファイルアクセスの傾向を把握する上で必要な概念である。

演算方法の具体例を図 4.4 と図 4.5 を用いて記述する。はじめに、ログ [[/foo,read,...], [/foo/bar,write,...], [/foo/bar,write,...], [/foo/baz,trunc,...], [/hoge/piyo,creat,...]] で与えられる TSL が図 4.4 の TSL:C である。このとき、TSL:C を Addend TSL、TSL:A を Augend TSL としたときの TSL:A - TSL:C つまり TSL:A-C を計算する。Addend TSL の各ノードを'/' から順番に足していく。ただし、TSL:A － C の初期状態は TSL:A である。ルートノードである'/' はラベルを持たないノードのため、減算処理は行われない。よって'/' の子ノードにあたる'hoge' ノードの処理に移る。しかし、'hoge' ノードは、ラベルを持たないノードのため、減算処理は行わず、'hoge' の子ノードにあたる'piyo' ノードの減算処理に移る。'piyo' ノードは TSL:A の'hoge' の子ノードにはないノードである。そのため、'/hoge' の子部分にファイル名が piyo、属性がファイルアクセス:-1、システムコール:read、etc：... というノードを新しく生成する。このとき、ファイルアクセスの値は 1 にマイナスを付けて-1 としている。'/hoge' 以下のノード全ての処理が終わったため、'/foo' ノードの処理に移る。'/foo' はどちらも [read,...] のラベルを持つノードのため、ラベルの値とキャラクタが打ち消されて空のノード（[0,None]）になる。'/foo' の子ノードである'baz' も同じように打ち消されて、空のノードになる。ここで'/foo/baz' は子ノードを持たない葉であり、空のノードである。そのため、このノードを TSL:A-C から削除する。子ノードがあるノードは空のノードであっても削除は行わない。'/foo/bar' ノードは [1,write,...] と [2,write,write,...] の減算を行う。TSL:C の方がアクセスカウントが多いため、[-1,write,...] というラベルになる。

4.4 TSL の抽出

TSL:D（TSL:A+B の read only 部分）の説明 TSL における抽出処理とは、指定したラベルをもつノード以外のノードを削除することである。そのため、指定したラベルの属性をもつノードのパラメータやキャラクタは変化させず、それ以外の子ノードに指定したラベルの属性が存在しないノードを削除する。

特定の属性をもつノードのみで構成された TSL を Simplex TSL と定義する。Simplex TSL を使うことにより、ファイルアクセスによって発生する特定の属性ごとの TSL の傾向を把握することができるようになる。これは TSL の特徴を取得する上で必要となる。

演算方法の具体例を図 4.3 と図 4.6 を用いて記述する。はじめに、 4.2 節より、ログ [[/foo,read,...], [/foo/bar,write,...], [/foo/bar,read,...], [/foo/baz,trunc,...], [/foo/baz,write,...], [/hoge,trunc,...], [/hoge/huga,read,...], [/hoge/hogehoge,creat,...]] で与えられる TSL が図 4.3 の TSL:A+B であ

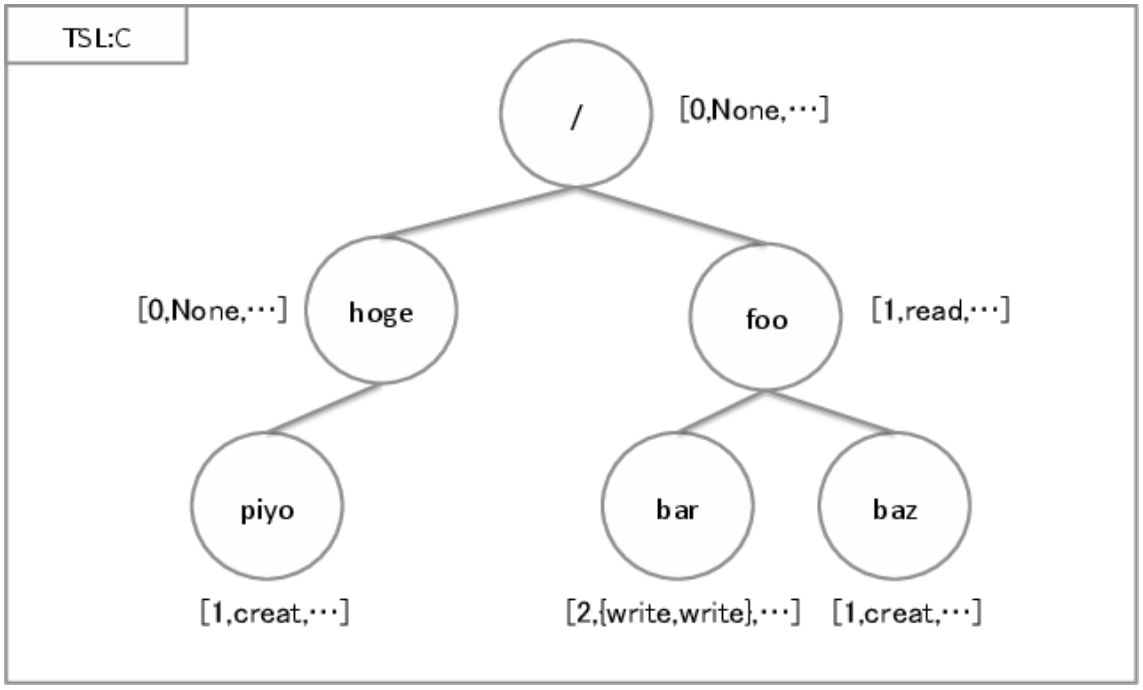

図 4.4: TSL:C

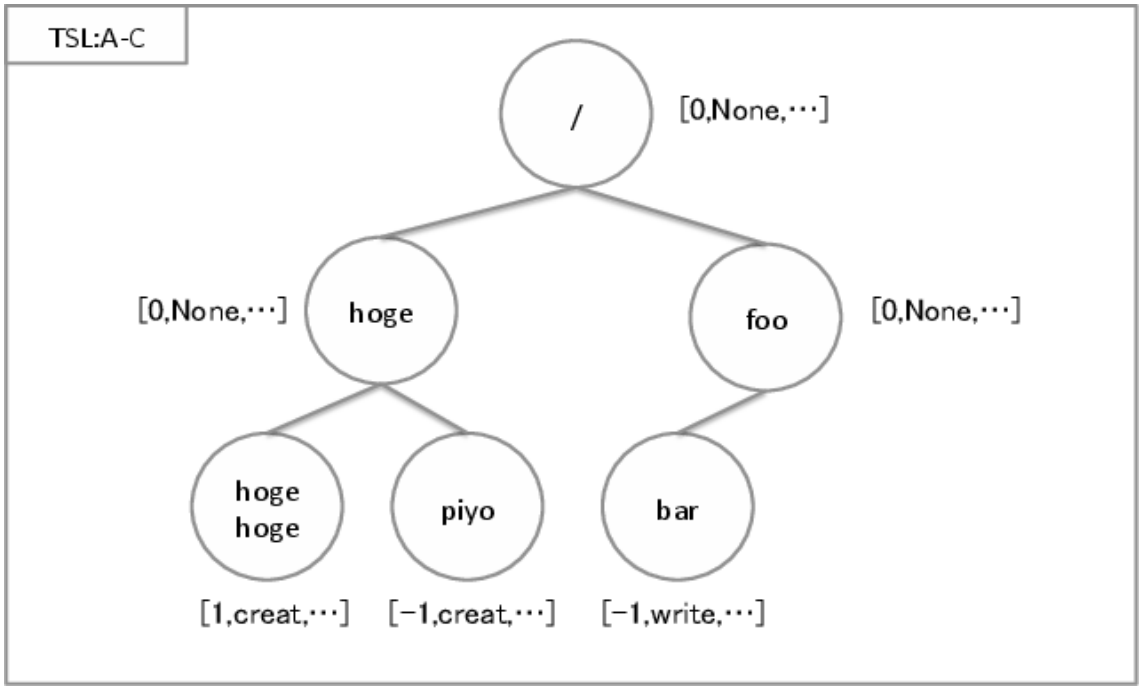

図 4.5: TSL:A-C

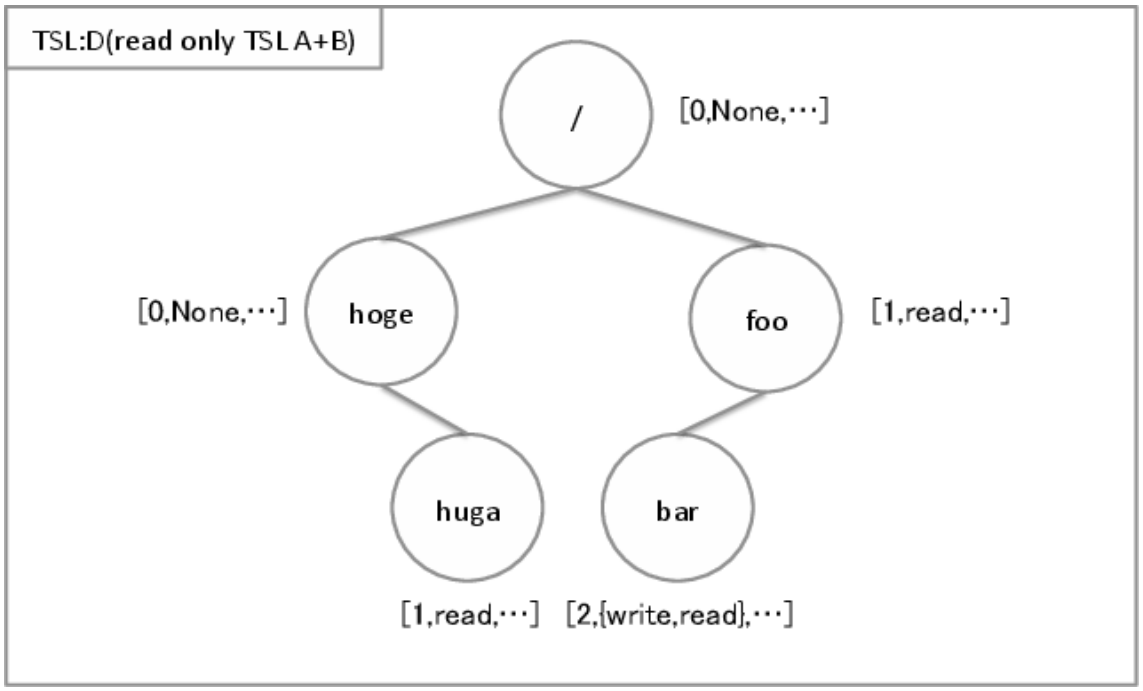

図 4.6: TSL:D(TSL:A+B の read を持つノードによって構成される TSL)

る。このとき、TSL:A+B からラベルの’system call’ 部分で、’read’ を持つノードのみで構成された TSL:D を抽出する。ルートノードである’/’ はラベルを持たないノードのためであるが、子ノードを持つ親ノードのためノードの削除を行わない。よって’/’ の子ノードにあたる’hoge’ ノードの処理に移る。’/hoge’ ノードも、’/’ ノードと同様にラベルを持たない親ノードのため、ノードの削除を行わない。’/hoge’ の子ノードにあたる’/hoge/hogehoge’ ノードの処理に移る。’/hoge/hogehoge’ ノードはラベルに’read’ を持たないノードである。かつ、子ノードを持たないノードであるため、このノードを削除する。もう 1 つの’/hoge’ の子ノードにあたる’/hoge/huga’ ノードの処理に移る。’/hoge/huga’ ノードはラベルに’read’ を持つノードである。よって、このノードの削除は行わない。以上の操作によって、[[/foo,read,...], [/foo/bar,read,...], [/hoge/huga,read,...]] で構成される read を持つノードとその親ノードによって構成される TSL が抽出できる。

第5章　TSLを用いたアノマリ検知

本章では、APT の攻撃者を検知するために必要な要求分析及び関連研究・手法について述べる。

5.1　FALから得られるユーザの活動傾向についての考察

3.5 節より、コンピュータを使用する大多数のユーザは業務を遂行するためのツールとしてソフトウェアを日々利用している。本研究では、前述のユーザを対象としている。また、そのユーザになりすまして情報の取得を狙うものが攻撃者である。よって、ユーザだけでなく、ユーザに扮する攻撃者の活動傾向についても考察を行う必要がある。

ユーザの活動傾向として上記より、ソフトウェア操作によって日常的に業務を行うことが考えられる。そして、3.5 節より、そのソフトウェア操作を行う際にファイルシステム上のファイル/ディレクトリに対してアクセスが発生する。ゆえに、ユーザは特定のファイル/ディレクトリに対して集中的・定期的にアクセスを行うことが考えられる。

攻撃者は 2.1 節より、自身の目的とする情報を入手するために情報が保持しているシステムの内部に侵入し、内部活動を行う。攻撃者の内部活動における特徴として以下の 3 点が挙げられる。

1. 攻撃者は一般ユーザと同じ User ID で活動を行う
2. パターンマッチングなどの検知手法を避けるため長期的に活動を行う
3. 攻撃者は一般ユーザと異なり、目的の情報を探すために探索活動を行う

はじめに、一般にオペレーティングシステムを用いて操作するシステムにログインするためには、アカウントが必要となる。ここでは一時的な利用のために用いられ、アクセスできるリソースが制限されるゲストアカウントは含まない。つまり、攻撃者はシステム内部に存在する User ID を用いてログインを行っている。そのため、システムを一般利用しているユーザに紛れて活動が行われる。また、攻撃者はシステム管理者に発覚されないように活動するため、パターンマッチングで検知される行動を避けて探索を行う。ゆえに、1 日または数日間隔で 1 つのコマンドを入力するといった長期的な視点で内部活動を行うとされている。上記の 2 つの特徴により、シグネチャタイプの検知手法では攻撃者の存在の有無を判定することが難しい。この特徴が APT 攻撃を発見することが困難であることの一因とされている。

しかし、攻撃者の特徴から、攻撃者の内部活動をユーザの FAL を用いてアノマリ検知をすることが可能であると考える。はじめに、攻撃者は目的の情報を取得するために、一般ユーザと同様のコマンドインタプリタまたはインターフェースを用いてシステム内で情報を保持しているストレージにアクセスを行う。ストレージへのアクセスが行われるということはすなわちファイルシステムへアクセスが行われることと同等のため、 3.3 節より入力したコマンドやアクセス先等の情報がファイルシステムへのアクセスとして記録できることを示している。また、攻撃者はどこにあるかもわからない情報の取得するために探索活動を行うため、活動の範囲が一般ユーザと比べて広域で横断的なアクセスを行うことが考えられる。

5.2 想定される FAL のパターンについての考察

5.1 節より、攻撃者のファイルアクセスの傾向として探索的な活動ないし、広域で横断的なアクセスを行うことが挙げられる。このファイルアクセスのパターン例を TSL 上に表現して、考察を行う。はじめに、基準となる TSL のモデルが図 5.1 である。そして、ユーザまたは攻撃者によるファイルアクセスのパターンとして深さ優先アクセス、幅優先アクセス、ランダムアクセス、集中的アクセスの 4 つが考えられる。この 4 つのパターンを TSL 上で表現したものが図 5.2、 5.3、 5.4、 5.5 である。

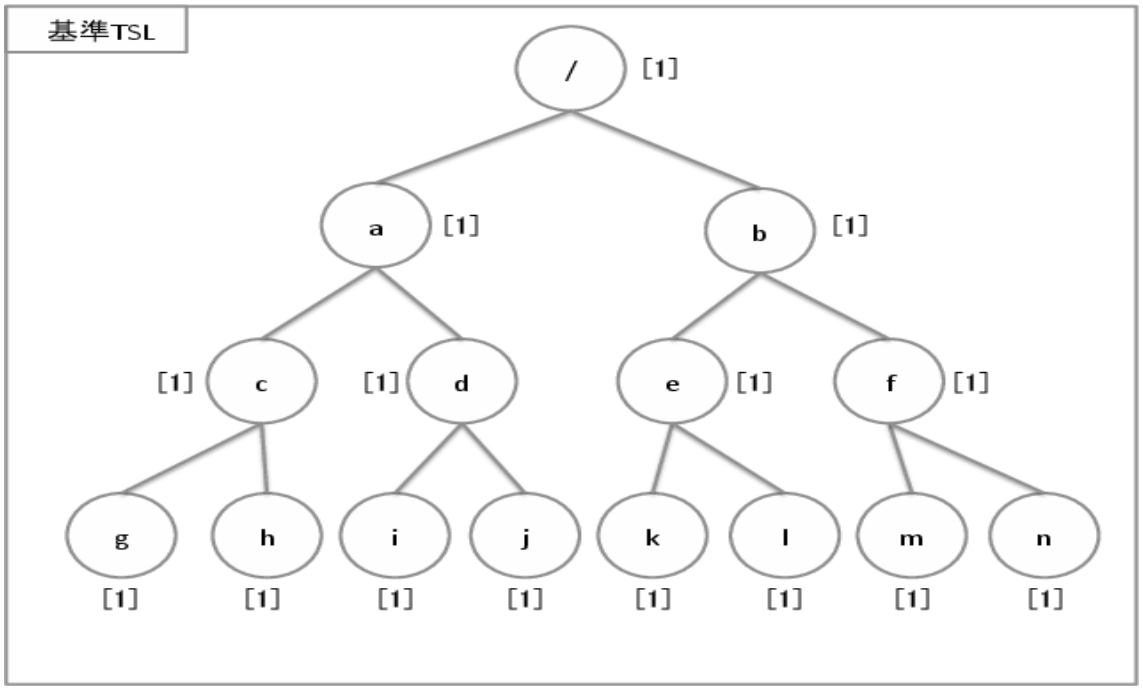

図 5.1: 基準 TSL

・基準 TSL の話を簡単に図 5.1 は、深さ 3 の完全二分木である。各ノードについている

ラベルはファイルアクセスのカウントを示したものである。よって全てのノードに大してファイルアクセスが行われた TSL である。

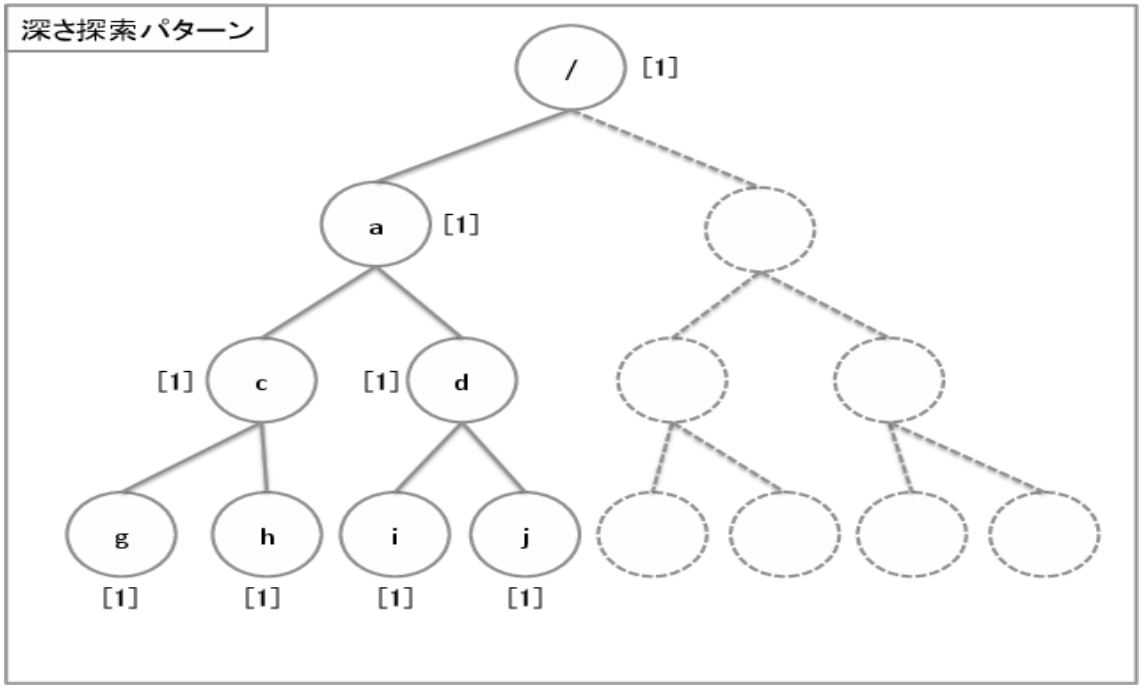

図 5.2: 深さ探索によるアクセス

図 5.2 は基準 TSL（図 5.1）のノードを持つファイルシステム上である深さ探索が行われたときの FAL から作られる TSL である。深さ探索とは、根から始まり子のノードから子のノードへと木の深い部分に降りていく探索方法であり、目的のノードもしくは葉のノードに至るまで降り続ける。葉のノードに達したときは、親のノードに戻り、別の子のノードへと移動していく探索方法である。図 5.2 はノード a の子ノードの探索が終わった段階での TSL である。ゆえに、ノードが'/' , '/a', '/a/c', '/a/d, '/a/c/g', '/a/c/h', '/a/d/i, '/a/d/j' で構成される TSL が形成される。

図 5.3 は基準 TSL のノードを持つファイルシステム上である幅探索が行われたときの FAL から作られる TSL である。幅探索とは、根から始まり根の子のノードを全て探索した後、その子のノードへと移動を行っていく探索方法である。そして、目的のノードもしくは葉のノードに至るまで降り続ける。図 5.3 は深さ 2 までのノードの探索が終わった段階での TSL である。ゆえに、ノードが'/' , '/a', '/a/c', '/a/d, '/b/e', '/b/f' で構成される TSL が形成される。

図 5.4 は基準 TSL のノードを持つファイルシステム上であるランダムアクセスが行われたときの FAL から作られる TSL である。ノード a,b,c,d,i,m,n に対してファイルアクセスが行われたときに'/' , '/a', '/a/c', '/a/d, '/a/d/i, '/b/f/m', '/b/f/n' で構成される TSL が形成される。

図 5.5 は基準 TSL のノードを持つファイルシステム上である集中的なアクセスが行わ

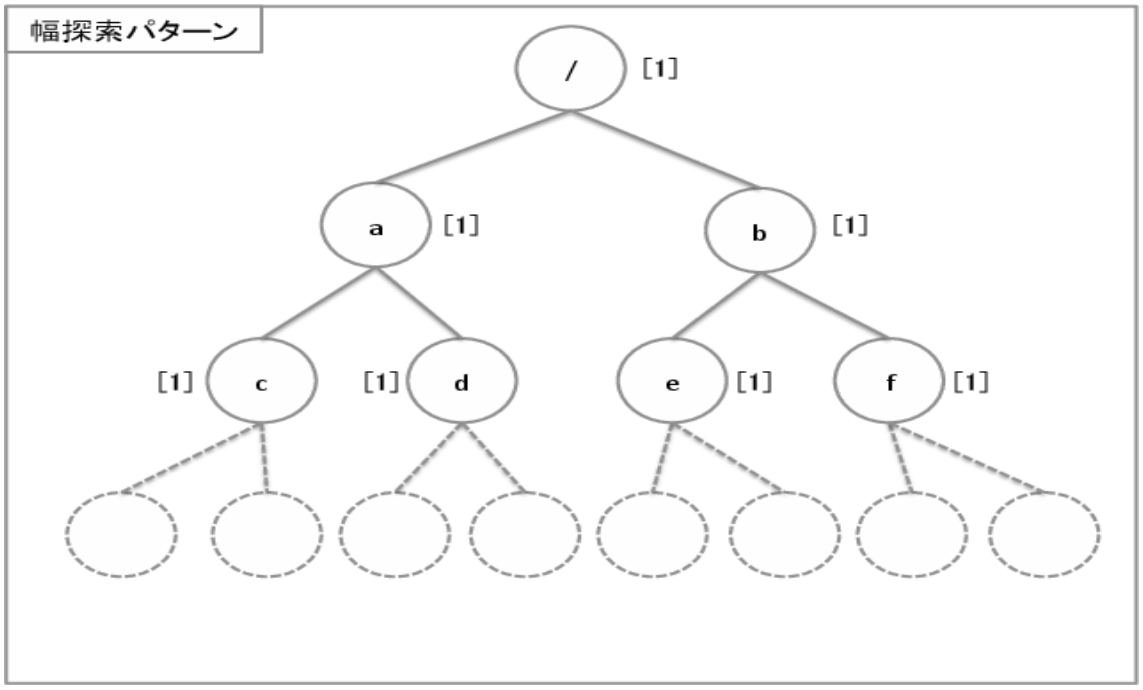

図 5.3: 幅探索によるアクセス

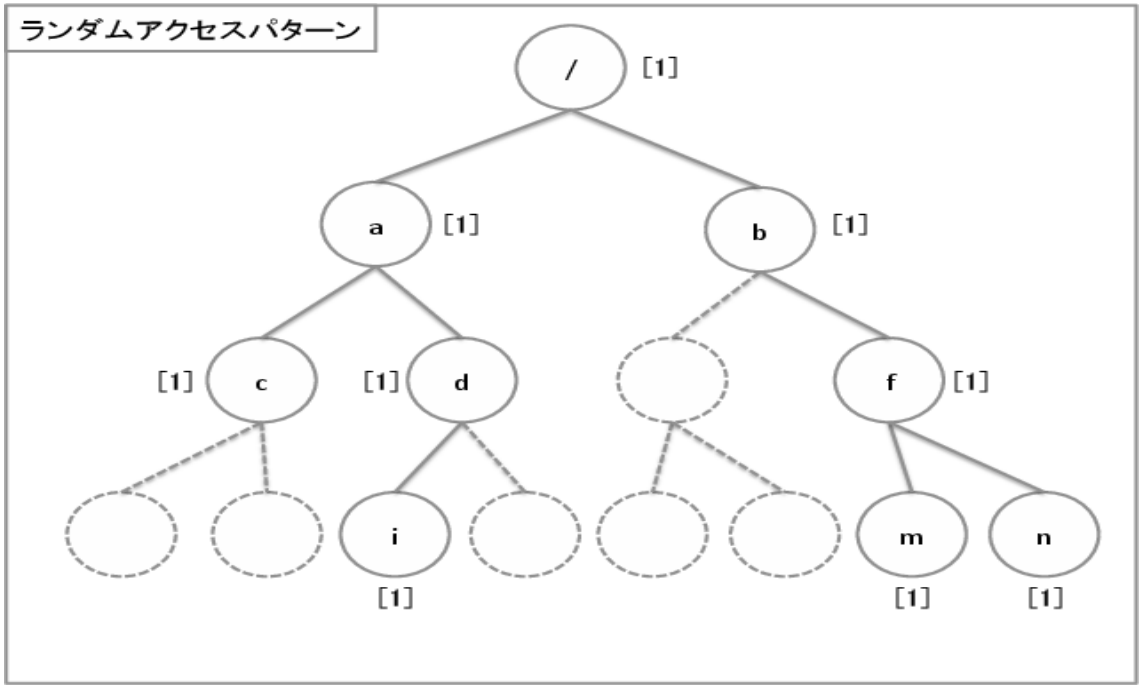

図 5.4: ランダムなアクセス

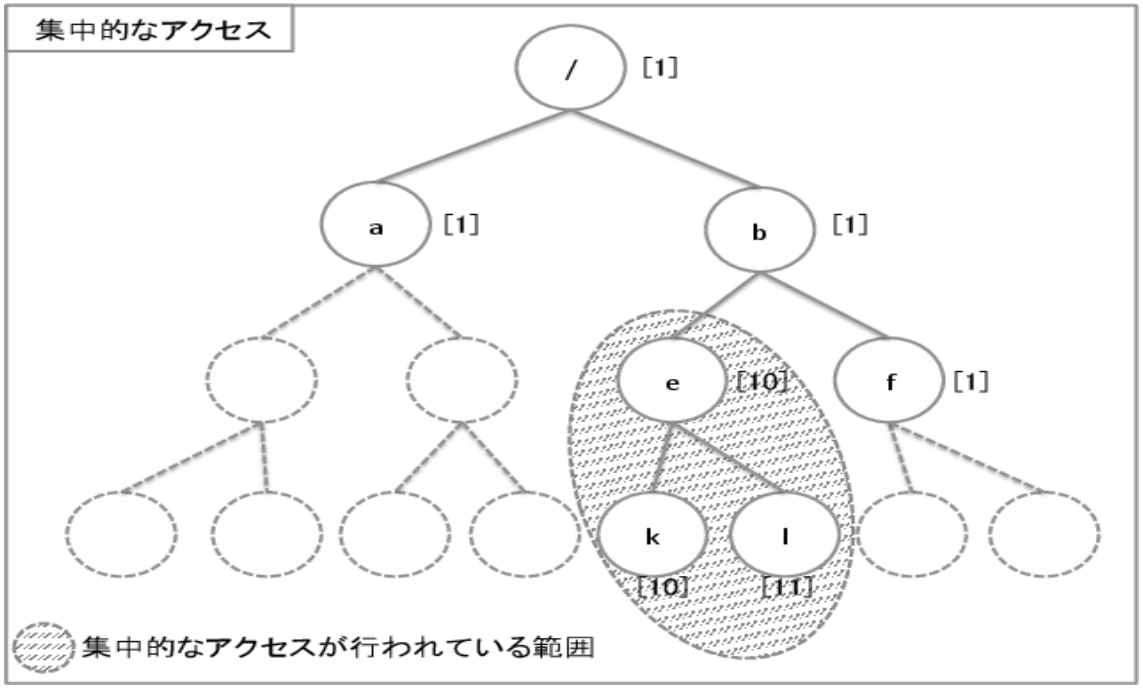

図 5.5: 集中的なアクセス

れたときの FAL から作られる TSL である。ノード a,b,f に対して 1 回のファイルアクセスが行われ、ノード e,k,l に対して 10 回のファイルアクセスが行われたとき、'/' , '/a', '/b/e', '/b/e/k, '/b/e/l', '/b/f' で構成される TSL が形成される。

以上の図 5.2、 5.3、 5.4、 5.5 より、探索的なファイルアクセスと集中的なファイルアクセスによって形成される TSL の概形は異なることがわかる。

5.3 TSL の持続的な記録についての考察

FAL のうち、'file path'、'access time'、'system call'、'user id' の 4 つを 1 つのエントリとして TSL に記録することを考える。FAL の例として、"/Users/m_sonoda/ownCloud/","20151227-160847-326","access(2)","501" が与えられたとき、このエントリのサイズは 67 バイトである。このサイズを基準として、これが 1 つのファイルアクセスに対して記録されるため、1 日のファイルアクセスが 30 万だとすると、1 日に記録されるエントリのサイズは約 20M バイトである。1 年間の場合約 7G バイトとなる。これを 1000 人単位で記録する場合、1 年間で約 7T バイトとなる。そして、TSL はエントリのファイルパスをもとに構造化を行うため、重複するファイルパスが多ければ多いほど、TSL のサイズは小さくなると考えられる。よって、FAL から必要な要素を取り出し TSL の形式で記録を行えば、現実的に持続的な監査システムのログの記録が可能となる。

前述では、'time' をそのまま記録する前提でエントリのサイズを算出している。しかし、TSL では時間の概念をある単位時間である Unit を 1 つの TSL の時間単位として記録を行っ

ていく。これはエントリのサイズを小さくしながら、ある単位時間を組み合わせて柔軟な比較を行うためのものである。 2.4 節より、攻撃者の潜伏期間は 長期間に渡るものであり、期間にばらつきが生じているため、任意の期間の TSL を作成し、その複数の TSL を組み合わせて比較を行うことで攻撃者によるファイルアクセスの検知の精度を向上させることができると考えたためである。

time を記録しない場合、1 つの TSL のサイズはさらに小さくなる。FAL の time の長さとして、例として挙げている BSD audit の場合だと”20151227-160847-326” である。これは 21 バイトなので、例として挙げたエントリのサイズは 44 バイトになる。

5.4 ファイルアクセスにおける活動範囲の比較によるアノマリ検知

TSL は FAL を木構造に写像して記録したものであるため、木構造の一種である。TSL の比較手法を提案するのにあたり、木構造の比較に関する先行研究を挙げて、その先行研究の手法を TSL の比較に適応する場合、どのような結果が得られるかを考察する。

木構造の比較に関する研究として、はじめに Tree Edit Distance [17] による木構造の比較が挙げられる。Tree Edit Distance とは、2 つの木（A,B）があったとき木 A が木 B と同じ木構造になるために、ノードの削除、新しいノードの挿入、ノードの置換という木への処理が何回必要かを尺度として、木構造における構造の差異を定量的に表しているものである。この論文の応用として、Tree Edit Distance に部分木の移動という拡張を加えた LaDiff [18]、MH-Diff [19]、という手法が提案されている。これらの提案手法で得られる情報は、木構造同士の構造上の差異である。TSL の比較の目的は、ユーザの活動傾向に関する情報を定量的に表すことである。よって、ユーザの活動するファイル/ディレクトリによらない比較が必要だと考える。何故ならばユーザの活動を週、月、年単位で捉えた場合、いつも同じファイル/ディレクトリにアクセスすることは考え難いからである。ゆえに、本論では TSL の比較について、Tree Edit Distance [17]、LaDiff [18]、MH-Diff [19] とは異なる手法を用いる。

5.2 節より、構造化によって得られる情報として、定量的な情報と定性的な情報の 2 種類が存在する。定量的な情報として、木構造における’ 深さ’、’ ある深さでの幅’、’ あるノードの子の多さ’、’ 木の面積’、’ ファイルアクセス数’、’ システムコール番号’、’ ユーザ名’ などが挙げられる。また、定性的な情報として、属性に関連してある部分木のファイルアクセスが多い/少ない、ある部分木の特定のシステムコールが多い/少ない、あるユーザの TSL が大きい/小さいなどが挙げられる。

5.1 節より、ユーザと攻撃者のファイルアクセスの傾向の違いとして、ファイルアクセスにおける範囲が異なることが挙げられる。このファイルアクセスにおける範囲に着目して、これを定量的に扱うことを考える。広い/狭いで表現される範囲という定性的な情報を定量的に扱うためには、定量的な情報を組み合わせて、定性的な情報として定義を行う

必要がある。TSL では、FAL におけるファイルアクセスを含むエントリを木構造に写像させて記録している。ゆえに、ファイルアクセスの範囲とは TSL の範囲と同様のものであるといえる。そして、TSL は木構造と同様に定量的な情報として、’ 深さ’、’ 幅’ を持つ。この情報を用いると、深さ毎の幅を計算することが可能であり、その幅に関して和を計算することは、積分計算と同等であるため、この計算はファイルアクセスの範囲の定量的な表現の 1 つといえる。よって、ファイルアクセスの範囲は、木構造の幅と深さを用いて計算できる木構造の面積として定量的に表現することが可能となる。

5.1 節より、ユーザと攻撃者の活動傾向の違いとして、長期的なファイルアクセスにおける範囲が異なることが挙げられる。また、本節のファイルアクセスにおける範囲の数値化よりファイルアクセスにおける活動範囲を定量的に比較することが可能となる。以上より、FAL を構造化させた TSL を記録し、その TSL の比較を行うことにより、ユーザのファイルアクセスに対するアノマリ検知が可能だと考える。

第6章　fspeekの設計と実装

本研究ではファイルアクセスの記録と比較を行う File System Peek(fspeek) を提案する。fspeek は３つの要求を満たすシステムである。ひとつめの要求はユーザのシステム内で常時長期的に動作させることである。これは内部活動をリアルタイムで検知するためである。ふたつめの要求は年単位でファイルアクセスログを蓄積することである。これは長期的な内部活動を検知するためである。みっつめはファイルアクセスログ全体の構造を比較することである。これは横断的または縦断的に行われる侵入者の内部活動の傾向を判別するためである。

上記の要求を満たすための制約条件として２つの条件が挙げられる。ひとつめはファイルアクセスの構造化と集約である。未加工のログではログ同士の関係性を把握出来ないことから、ログの要素間の関係を明らかにしてデータの抽出と比較を行いやすくするために構造化が必要となる。また、１つのログファイルのサイズが大きいため、年単位でログファイルを蓄積するために集約化が必要となる。ふたつめはファイルアクセスの傾向の維持である。集約化を行った場合でも、ファイルアクセスの傾向を維持しなければファイルアクセスログ全体の構造を比較することが出来ないためである。

fspeek の全体構成図は図 6.1 のようになる。fspeek の FAL の操作は FAL の取得を行う FAL reader と FAL の構文解析を行う FAL parser をもつ P-FAL クラスによって実現される。また、fspeek の TSL への構造化とそれの比較によるアノマリ検知は FAL から TSL への構造化を行う parsed FAL constructor と TSL の結合と分離を行う TSL combiner,extractor 及び TSL の比較を行い、アノマリ検知を行う TSL comparer をもつ M-Tree クラスによって実現される。本論では BSD 系 OS である Mac OSX 上で実装を行った。この図をもとに fspeek の設計と実装について述べる。

6.1　FALの取得

本節では FAL reader の設計と実装について述べる。はじめに監査ログの出力プロセスについて記述する。図 6.1 の OS で囲まれた部分は、各 OS がシステムソフトウェアの機能として有しているものである。ソフトウェアの操作によって、ファイルシステム上のファイル/ディレクトリへのアクセスが行われるとき、システムコールによってそれは実現される。このとき、呼び出されたシステムコールをフックしてファイル/ディレクトリに関して、いつ、誰が、何をしたか、どうやって行ったかを記録するものが Audit System である。その Audit System によって記録されたものが File Access Log（FAL）である。

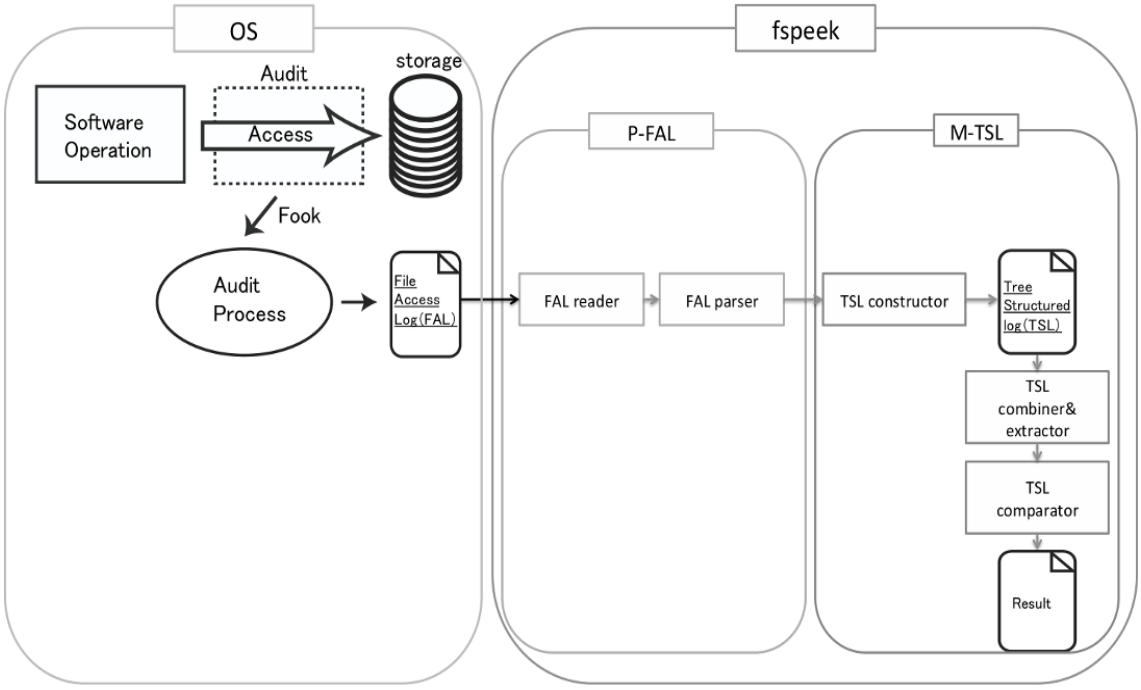

図 6.1: fspeek 構成図

本節では FAL の取得を行う FAL reader の設計と実装について述べる。FAL reader の概要図が図 6.2 である。はじめに、各 OS の Audit System によって出力された FAL が各 OS 毎に固有の場所に記録される。OS によって記録される FAL の最大容量は設定によって変わるが、最大容量に達したときの処理は、別ファイルとして記録されるかまたは同じファイルに上書きされるかのどちらかである。FAL を連続的に読み込みを行うことを考えると、同じファイルに上書きされる場合はそのファイルの差分を取得すれば良い。別ファイルに記録される場合は、そのファイルの差分を取得するだけでなく、別ファイルに記録される FAL を追跡して読み込む必要がある。よって、FAL reader では指定の FAL を対象にログファイルの差分の読み込みを行い、新しい FAL が作成された場合、その新しい FAL を読み込みの対象に移す処理を行う。

ソフトウェアの操作によって新しいファイルが作成されたことをユーザプログラム上で検知するためには、シグナルハンドラが必要となる。Mac OSX ではこのシグナルハンドラを用いてファイル/ディレクトリの操作を監視できる pyinotify [16] を用いる。pyinotify は Linux カーネルの機能であるファイルシステムのイベントの監視を行う inotify を、python 上で実装したアプリケーションである。FAL reader は pyinotify を用いて実装している。

FAL reader は、OS 毎に設定された FAL をもつディレクトリのパスまたは任意の入力した引数をパスとして実行する。この引数または設定されたパスを log file path とおく。対象のディレクトリを監視し、そのディレクトリ下のファイルが新しく作成・移動・削除されたときに、そのファイルのパスを log file path に代入する関数をもつ audit handler というクラスを定義する。また、pyinotify の WatchManager() により監視スレッドを作成し、

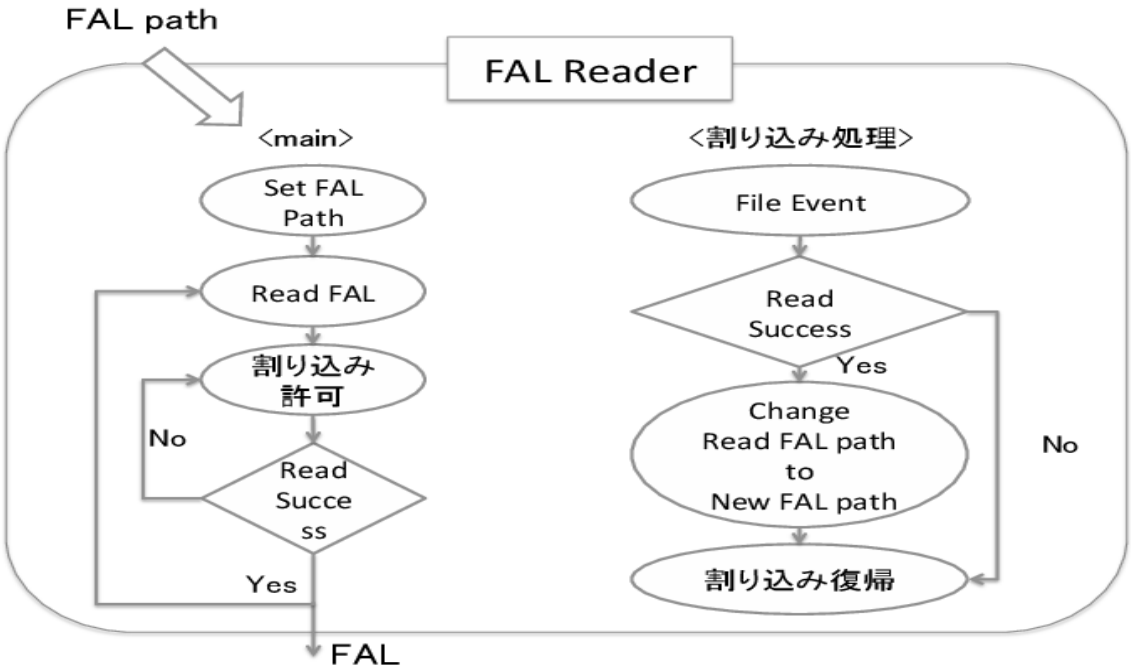

図 6.2: FAL reader 概要図

audit handler を継承してインスタンス化を行う。そして、log file path を用いて監視スレッドに監視するディレクトリのパスを指定する。そして loop 関数によってプログラムを開始し、シグナルハンドラによるプログラム停止処理が発生するまで監視を行う。このときに、読み込まれた差分は標準出力される。

6.2 FAL の構文解析と TSL への構造化

本節では、図 6.1 より FAL の構文解析を行う P-FAL クラスの FAL parser FAL から TSL への構造化を行う M-Tree クラスの parsed FAL constructor の設計と実装について述べる。この FAL parser の概要図が図 6.3 である。

FAL の形式と粒度は OS 毎に異なる。OS 毎に異なる FAL の形式に応じて、FAL の構造化部分の実装を変更するのは、プログラムの煩雑化や煩雑化によるバグを招く要因となるので、M-Tree クラスの処理は OS によるログの形式の違いを考慮せずに行うことが望ましいと考える。よってこの FAL parser によって FAL の形式と粒度の差異を平滑化する。また、 3.4 節より FAL の情報はシステムコールをトリガとするファイルアクセスに関わる情報が一通り記録されている。全ての情報を TSL に使う必要はなく、本論の目的であるユーザに紛れて探索的な活動を行う攻撃者のファイルアクセスを検知するために必要な情報があれば良いと考える。そのため、FAL から任意のエントリの中にある要素を抽出して、それらを 1 つにまとめて出力を行う。

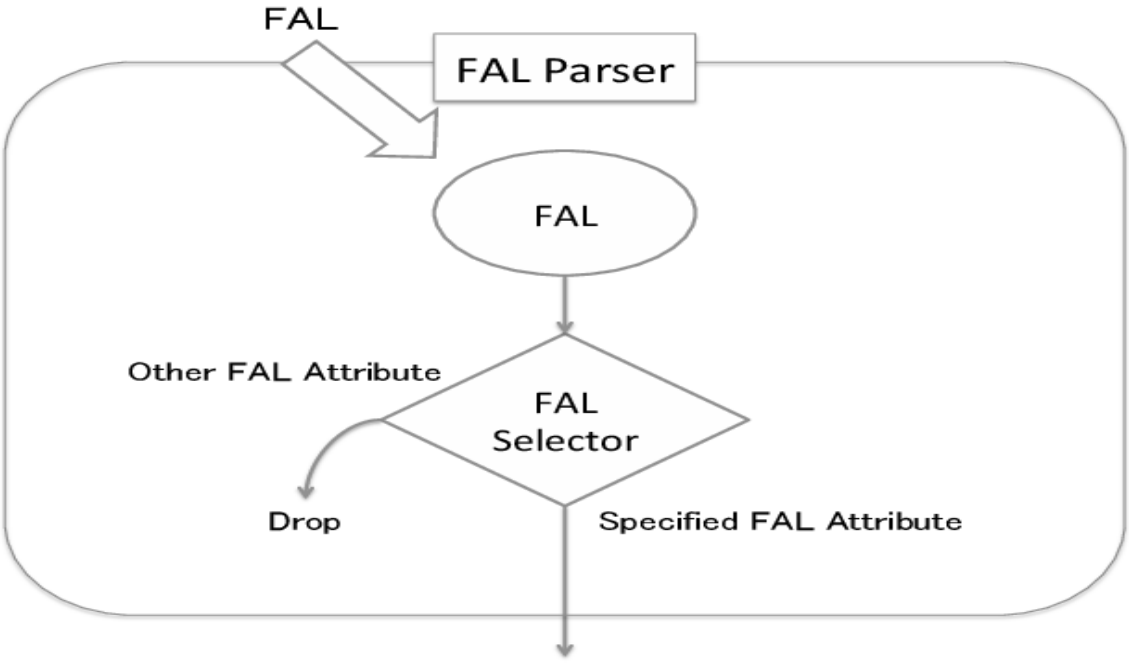

図 6.3: FAL parser 概要図

3.6 節より、ログをそのまま記録した状態だとデータ量の上限とログ単体での有意性が不明瞭という問題が生じることが考えられる。そこで、ログ情報を構造化することでデータ量の削減及びアノマリ検知に適した可読性の高いログの形成を行う。FAL から TSL への構造化を行う FAL constructor の概要図は図 6.4 で示す。本章で述べた要件を踏まえて構造化を行う上で重視する部分は、どんなファイルアクセスが行われたかである。よって FAL をファイルシステムにおいてファイルの位置を相対的に表現することに適している木構造型のデータ構造にする。 3.6 節より、FAL にはファイル/ディレクトリのパスの情報が含まれている。このパス情報を用いて構造化を行う。 3.1、 4.1 節より、ファイル/ディレクトリパスで'/' で区切ることで分かれたファイル/ディレクトリ名を基に、ルート'/' から順に、末尾の葉のファイル/ディレクトリに至るまで木のノードを作成していく。このとき、その葉ノードにはノードの属性を表すラベルをつける。このラベルは、FAL parser で出力された時間以外の情報で構成されている。また、指定した時間を単位時間として TSL を構造化していく。よって単位時間毎に TSL が記録されていくことになる。TSL を記録し続けていく際の問題点として、ソフトウェアの作業ディレクトリやソフトウェアの実行場所は通常変わらず、そのソフトウェアを常用している場合、頻繁に実行されるため、同じファイル/ディレクトリに対して集中的なファイルアクセスが発生する。よって、その場合、TSL のサイズが過剰に大きくなることが考えられる。fspeek の目的はユーザに紛れた攻撃者の探索活動を含む、ファイルアクセスのアノマリ検知を行うことであるため、全てのファイルアクセスの詳細を記録する必要はなく、集中的なファイルアクセスが行われたこととファイルアクセス全体の傾向がわかれば良い。そのため、特定のファイル/ディ

レクトリに対して過剰なファイルアクセスが行われたときにはTSLの集約化を行う。TSLの集約化は、あるノードにラベルの追加が他よりも明らかに高い頻度で発生したときに、そのノードのラベルの情報を削除し、その親のノードに集約化が行われたことを表すフラグを付ける。これにより、TSLのサイズを抑えつつ、TSLからファイルアクセスの傾向を検出することが可能となる。

OSXのFALはXML形式で出力することができるため、このXML形式のログをパースする。parsed FAL constructorはFALのパスを引数として実行される関数である。pythonライブラリのをElementTreeを用いて、XMLの処理を行う。はじめに、XMLのElementTreeのインスタンスを作成し、そのインスタンスに引数からもとに開いたFALを代入して読み込みを開始する。このとき、取得したい属性をイテレータとして設定する。そのイテレータを使って取得した属性を結合し、標準出力を行う。構造化の段階で属性を分割して処理するために、属性の間にはダブルクォーテーション及びカンマを付ける。

FAL parserによって出力されたparsed FALをparsed FAL constructorによってTSLへの構造化を行う。はじめにノードを表現するための構造体M-nodeを定義する。M-nodeはファイル/ディレクトリ名、親のノード、子のノードの3つの引数をもとに構成される。他の要素として、ファイルアクセスのカウント数、時間、システムコール、ユーザidなどのFALの属性と集約化回数を格納することができる。このM-nodeを使って作成されたインスタンスがTSLである。parsed FALを読み込む前に、TSLの初期化・作成を行う。M-nodeの引数を'/'のみでインスタンスを作成すると、これは親も子も属性も持たない空のノードとなる。この'/'のみのTSLに対してparsed FALをもとにノードとラベルを追加していくことがparsed FAL constructorの処理である。

次にparsed FALを行単位で読み込みを行う。読み込んだ1行のログファイルに対して、ダブルクォーテーションとカンマを用いて分割して配列を作成する。そして、配列の要素を引数にして木の追加と集約を行うadd aggre関数を実行する。add aggre関数は、はじめにファイル/ディレクトリパスの分解を行う。ファイル/ディレクトリパスが'/'のときは、'/'のみをもつ配列を作成する。このファイル/ディレクトリパスを格納する配列をpath arrayとする。それ以外のファイル/ディレクトリパスのときは、'/'をキーとして、ファイル/ディレクトリパスを'/'の間にあるファイル/ディレクトリ名に分解する。その分解したファイル/ディレクトリ名をpath arrayに木の根から順に葉が最後になるように追加していく。'/'で区切ると、先頭の'/'が''になるため、path arrayの先頭は'/'にする。ここで、TSLのノードのポインタをseekerとする。add aggreの処理はこのpath arrayとseekerを使って行われる。

path arrayの長さが0のときは、ファイル/ディレクトリパスが入っていないため処理は行わない。path arrayの長さが1以上のときは、path arrayが'/'かつpath arrayの長さが1の場合と、それ以外の場合で処理が分れる。path arrayが'/'かつpath arrayの長さが1の場合は集約化の回数を表す集約化カウンタが1未満のとき、seekerにそのFALの属性を追加する。この処理の後に、ファイルアクセスのカウント数が指定したしきい値を上回った場合、このseekerの集約化カウンタを1にする。それ以外の場合は、前述と異なり、path

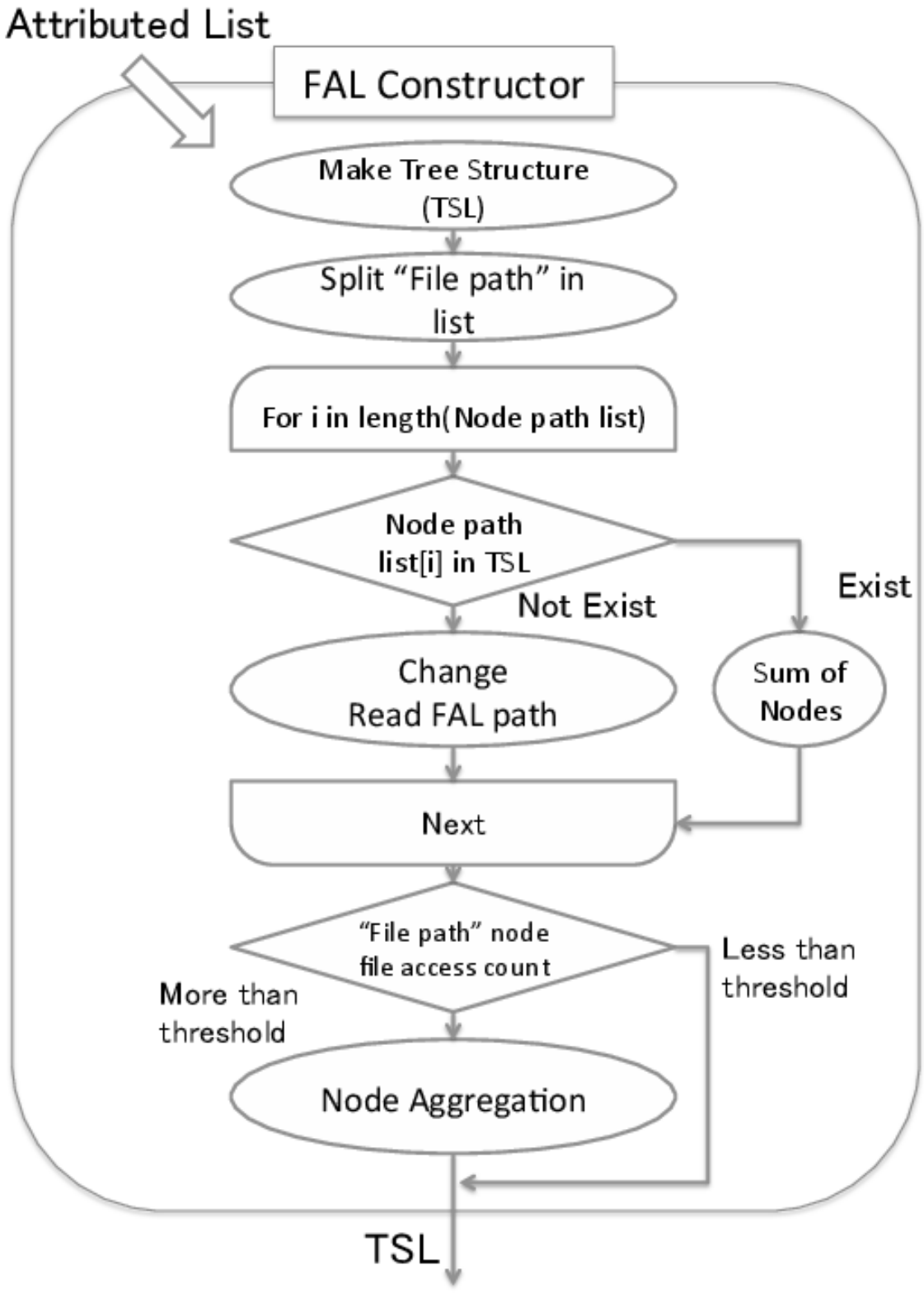

図 6.4: FAL-constructor 概要図

array に含まれるファイル/ディレクトリ名が複数存在しているため、現在の seeker と属性を格納するノードに差異が生じている。よって、葉の部分まで seeker を移動させる必要がある。path array の要素をループ処理の変数として用いて seeker を移動させる。seeker の次のインスタンスつまり、子のノードがあればそこのインスタンスを seeker にする。子のノードがなければそのときの変数をノード名として新しい子のノードを作成する。そして、そのノードのインスタンスを seeker にする。この処理を path array の数と同じだけ行い、ループの最後にその seeker の集約化の回数を表す集約化カウンタが 1 未満のとき、seeker にその FAL の属性を追加する。この処理の後に、ファイルアクセスのカウント数が指定したしきい値を上回った場合、この seeker の集約化カウンタを 1 にする。

6.3 TSL の結合と分離

FAL-combinar は 2 つの TSL が存在するとき、そのノード及びラベルの結合を行う。この FAL-combinar の概要図が図 6.5 である。FAL-combinar はノードの作成とノードの加算に分かれる。ノードの作成とは、Addend TSL にのみ存在するノードを Augend TSL に新しいノードとして作成することである。ノードの加算とは、あるノードの Addend TSL・Augend TSL の両方のラベルのパラメータの和を、その結合後の TSL のノードのパラメータの値とし、ラベルの属性がキャラクタであれば、Augend TSL のラベルに Addend TSL のキャラクタを追加することである。

これらの処理により、2 つの TSL を 1 つの TSL として扱うことができるようになる。6.2 節より、TSL は指定した単位時間で記録される構造体である。よって、この処理を組み合わせることで数ヶ月、数年といった長期的なファイルアクセスの構造化された情報をもつ TSL を作成できるようになる。また、この数ヶ月、数年という TSL は指定した単位時間の粒度の範囲で任意の始点で作成することが可能である。これを用いると長期的なファイルアクセスの比較を柔軟に行うことが可能となる。

TSL extractor は指定したラベルをもたないノードの削除を行う。この TSL extractor の概要図が図 6.6 である。ノードの分離とは、指定した属性を親も子も有していないノードの削除を繰り返し行うことで指定した属性によって構成された TSL を作成することである。TSL のすべてのノードに対して親または自ノードに指定された属性がない場合、そのノードを削除する。または、親または自ノードに指定された属性がある場合、そのノードを削除しないという処理を行う。

これらの処理によって、ある TSL から任意のラベルを持つノードによって構成された TSL を作成することができるようになる。本節より、TSL の加算により長期的なファイルアクセスの構造化された情報を持つ TSL が作成できる。このとき、その TSL に対して、任意のラベルを指定して分離処理を行うことにより、長期的なファイルアクセスの傾向を知ることができるようになる。これはファイルアクセスの比較のために必要となる情報である。

TSL combiner は parsed FAL constructor によって作成された 2 つの TSL を読み込み、そ

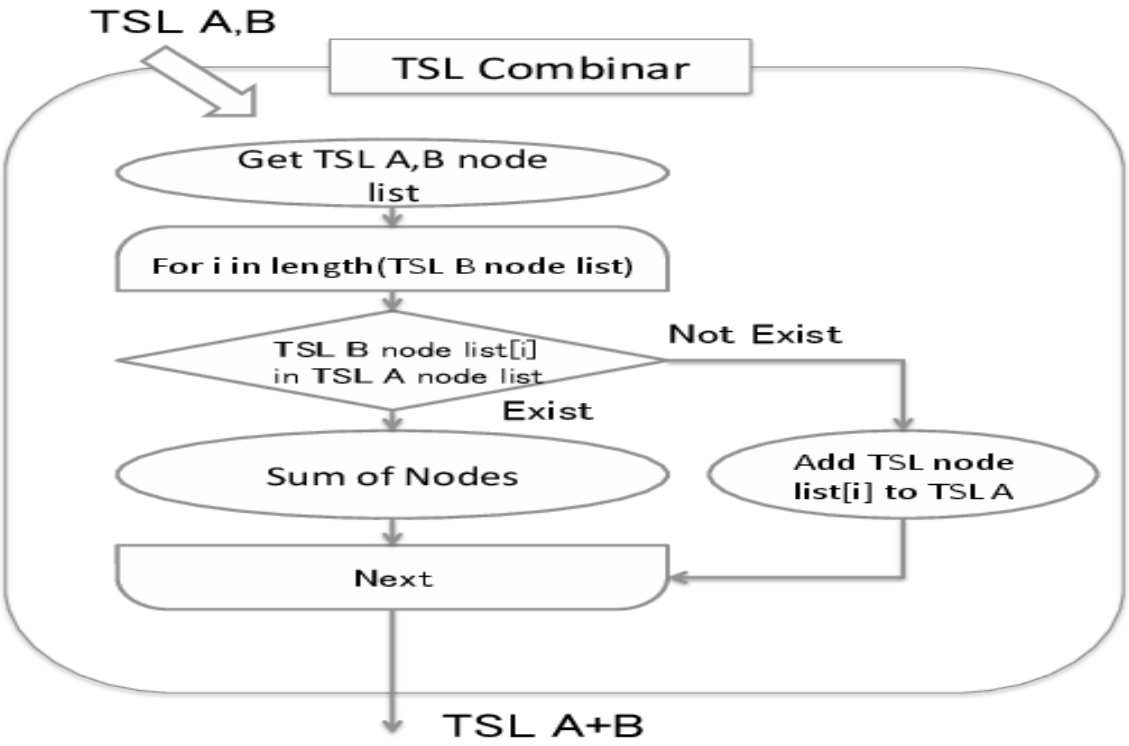

図 6.5: TSL-combinar 概要図

の TSL を結合したものを出力する関数である。足す側の TSL を Addend TSL、足される側の TSL を Augend TSL とする。はじめに、Augend TSL のノードのインスタンスのリストを、深さ優先探索順で取得する。このリストを Augend TSL list とする。次に、Addend TSL のノードのインスタンスのリストを、深さ優先探索順で取得する。これを Addend TSL list とする。変数 i を 0 から Addend TSL list の長さまでとり、for ループ処理を行う。このとき、Addend TSL list[i] のインスタンスが Augend TSL list の中に存在し、かつ Addend TSL list[i] のパス/ディレクトリ名が'/' でないときに、その Augend TSL list[i] のインスタンスが示すノードに、Addend TSL list[i] のインスタンスが示すノードの情報を追加する。または、Addend TSL list[i] のパス/ディレクトリ名が'/' でないとき、Augend TSL に新しく、そのインスタンスが示すノードを追加する。

TSL extractor は parsed FAL constructor によって作成された TSL を読み込み、指定した属性を持つノードで構成された TSL を出力する関数である。TSL のノードのインスタンスのリストを、深さ優先探索順で取得する。そのリストの順番を逆にし、そのリストを TSL re list とする。これは、ノードを削除するときに子のノードを一緒に削除しないための処理である。

最初に各ノードに対して、指定した属性をもたないラベルを数え上げる。次に、数え上げたラベル数を条件にして以下の処理を行う。数え上げたラベルの数とその属性のラベルの数が等しいとき 、つまりラベルのなかに指定した属性がなかったとき、子ノードの状

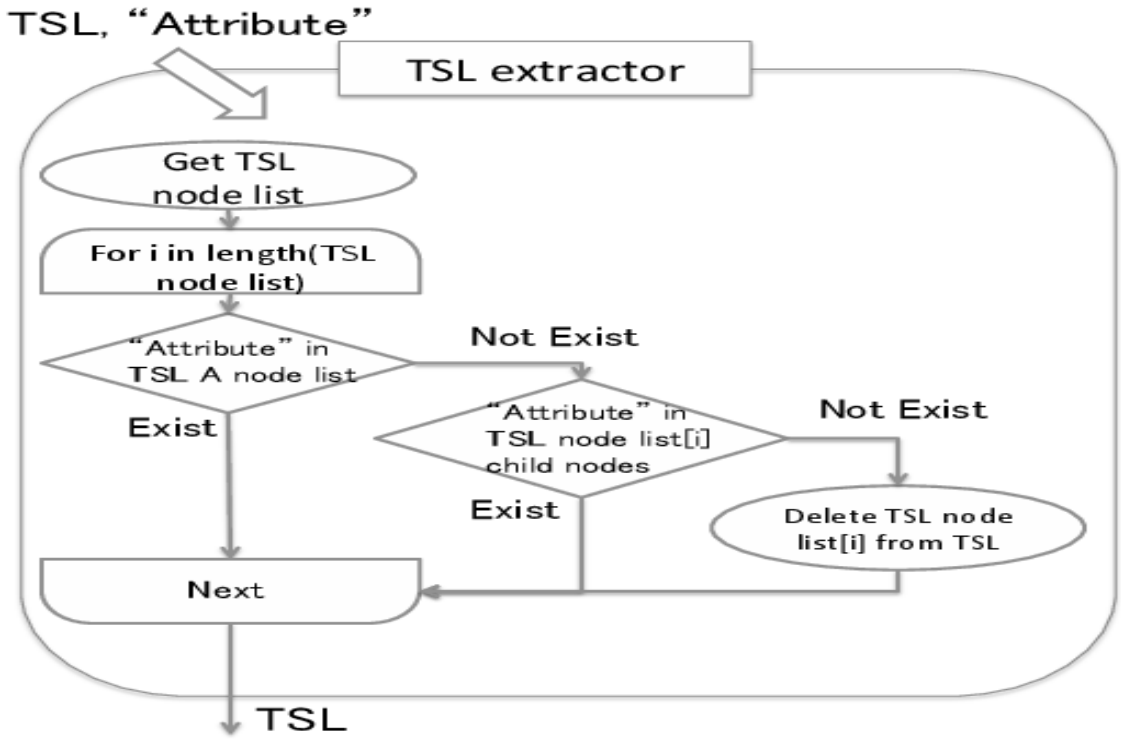

図 6.6: TSL-extractor 概要図

態によって 2 種類の処理を行う。ノードのラベルのなかに指定した属性はないが、子ノードのラベルに指定した属性があるときその子ノードのラベルを全て初期化し、空のノードにする。ノードと子のノード両方のラベルのなかに指定した属性はないとき、その子ノードを削除する。

6.4 TSL の比較によるアノマリ検知

5.4 節より、ファイルアクセスによる活動範囲を木構造の幅と深さを使って、定量的な表現が可能であることを述べた。この特性を用いて、長期的に記録された TSL を使って、TSL 同士を比較することによりファイルアクセスにおけるアノマリ検知を行う File Access Scope T-test（FAST）を提案する。FAST ではユーザが特定のファイル/ディレクトリに対して集中的なファイルアクセスを行うことを前提としている。よって、FAST は集中的でない探索的なファイルアクセスを検知することが目的である。

TSL の比較を行う FAST の実装である TSL comparer は比較のために TSL の定量的なデータを取得する stat TSL とそのデータを使い比較を行う TSL Anomaly Detector によって構成されている。stat TSL は、TSL を引数として、その定量的なデータを出力する関数である。　TSL から木の深さとその深さ毎の幅を取得する。ルートノードの深さを 0 として、その子ノードの深さが 1 となり、その子ノードの数が幅となる。よって、深さ毎に

各ノードの子ノードの数を総和を求めて、それを木の深さとセットにして配列として出力する。TSL Anomaly Detector は TSL から得られた木の深さとその深さ毎の幅を引数として、異常の判定を行う関数である。得られたデータより、深さ毎の幅の総和を求め木の面積を求める。その値を用いて、対応のある t 検定を行う。

第7章　fspeekの評価と考察

自身が実装したシステムの評価を行う。

7.1　実験構成

本実験では、OSX 10.9.5 上で記録された 2015 年 2 月から 2015 年 11 月までの FAL をもとに、TSL の比較実験を行う。FAL を取得するための設定ファイルは以下の通りである。

BSD Audit Config file

```
dir:/var/audit
flags:ex,pc,fw,-fr,fa
minfree:5
naflags:lo,ad,ex
policy:cnt,argv,arge
filesz:100M
expire-after:10G
superuser-set-sflags-mask:has\_authenticated,has\_conso
le\_access
superuser-clear-sflags-mask:has\_authenticated,has\_conso
le\_access
member-set-sflags-mask:
member-clear-sflags-mask:has\_authenticated
```

5.1 節より、システムコールによってファイルアクセスの傾向が異なることがわかった。また、 2.1 節より、検知したい攻撃者の活動はコマンドインタプリタを使った探索的な活動のため、ファイルアクセスにおいてファイルの情報を読み込む read、特に編集を行わない read only 部分に着目する。この read only によって構成される TSL を read only TSL とよび、この read only の TSL に探索的なファイルアクセスを混ぜた TSL を imit TSL とよぶ。read only TSL と imit TSL を比較し、ファイルアクセスにおける活動範囲が広がったときに、それが検知できるかどうかを実験によって確かめる。

予備実験として、取得した FAL のファイルアクセスの活動範囲に関して、TSL 間で差がないことを示す。表 7.1 は各 TSL の深さ及びその深さでの幅をまとめたものである。ま

た、表 7.2 は、TSL から”open read”イベントのみを抽出した TSL（read only TSL）の深さ及びその深さでの幅をまとめたものである。そして、各 read only TSL の面積を計算して、まとめたものが表 7.3 である。このとき、2 月から 6 月の read only TSL をグループ A[219,298,282,272,554]、7 月から 11 月の read only TSL をグループ B[232,260,277,172,134] とおく。このグループ A とグループ B を対象として、帰無仮説をグループ A とグループ B には差がある、対立仮説をグループ A とグループ B には差がないとおき、棄却率 5 %で対応のある t 検定を行う。結果、t 値が 1.695849323 となり、p 値が 0.165156424 となった。そのため、帰無仮説は棄却されるため、グループ A とグループ B には差がないといえる。

深さ	2 月の TSL 幅 (all)	3 月の TSL 幅 (all)	4 月の TSL 幅 (all)	5 月の TSL 幅 (all)	6 月の TSL 幅 (all)	7 月の TSL 幅 (all)	8 月の TSL 幅 (all)	9 月の TSL 幅 (all)	10 月の TSL 幅 (all)	11 月の TSL 幅 (all)
0	1	1	1	1	1	1	1	1	1	1
1	462	1491	1230	313	407	476	512	647	506	2408
2	333	3414	955	420	1178	626	426	3654	303	1263
3	590	2215	1029	3311	2164	3262	2472	1348	2144	3440
4	1220	3432	3169	7859	37601	11201	7237	2345	4176	7367
5	943	2413	6504	4244	2964	2021	1846	5751	757	1874
6	7450	4796	3947	9547	14949	15354	14174	11789	6533	7568
7	12330	14494	12600	23402	29567	33467	27718	19887	13404	17852
8	3091	4893	2648	3240	15669	5016	5389	4893	4446	5779
9	2279	3082	4203	2727	15195	4909	3423	4789	9965	2450
10	66764	102842	85994	113134	131390	183577	294860	149296	135026	221688
11	4260	1323	2797	1221	10388	2173	764	1464	1235	3111
12	177	142	18042	277	4488	654	283	736	660	391
13	55	81	10583	4321	3281	370	279	7443	423	123
14	12	61	4709	715	1825	985	567	1793	1132	50
15	355	96	93	212	1765	1738	1765	1893	1985	327
16		81	148	220	680	584	454	515	786	91
17		24	17	18	477	40	44	25	68	17
18		11	10	9	93	45	108	39	147	9
19		24	10	9	106	36	22	30	52	9
20		13	2		97					
21		3								

表 7.1: raw TSL の深さ毎の幅

深さ	2 月の TSL 幅 (open read)	3 月の TSL 幅 (open read)	4 月の TSL 幅 (open read)	5 月の TSL 幅 (open read)	6 月の TSL 幅 (open read)	7 月の TSL 幅 (open read)	8 月の TSL 幅 (open read)	9 月の TSL 幅 (open read)	10 月の TSL 幅 (open read)	11 月の TSL 幅 (open read)
1	1	1	1	1	1	1	1	1	1	1
2	15	9	10	9	11	9	8	8	9	8
3	23	41	35	24	92	26	29	28	24	17
4	30	49	40	31	100	31	42	31	27	22
5	38	54	40	29	89	67	39	30	24	20
6	42	84	86	85	123	24	86	102	22	18
7	28	20	24	28	69	26	22	25	24	20
8	20	20	22	29	28	24	20	26	21	20
9	13	11	11	24	17	15	9	15	10	4
10	5	5	6	9	6	5	4	7	6	4
11	2	2	3	2	3	2		3	2	
12	2	2	3	1	3	2		1	2	
13			1		2					
14					10					

表 7.2: open read TSL の深さ毎の幅

月	各月の TSL の面積 (open read)	各月の TSL の面積 (open read+1[a/d])	各月の TSL の面積 (open read+2 [a/d])	各月の TSL の面積 (open read+3 a/d)	各月の TSL の面積 (open read+4 a/d)	各月の TSL の面積 (open read+6 a/d)	各月の TSL の面積 (open read+8 a/d)	各月の TSL の面積 (open read+12 a/d)
2	219	244	269	294	318	368	417	516
3	298	325	341	366	396	456	515	635
4	282	310	338	365	393	438	493	604
5	272	301	329	353	383	439	495	607
6	554	584	604	634	655	717	776	893
7	232	264	282	314	320	367	429	555
8	260	289	318	346	374	427	484	595
9	277	309	339	350	381	444	506	632
10	172	203	225	250	283	345	406	528
11	134	161	187	201	227	279	332	436

表 7.3: TSL Square Measure

7.2 TSL の比較結果

2 月から 6 月の read only TSL をグループ A、7 月から 11 月の read only TSL をグループ B とする。そして、2 月から 6 月の read only TSL に探索的なファイルアクセスを混ぜた TSL をグループ A_n とする。また、7 月から 11 月の read only TSL に探索的なファイルアクセスを混ぜた TSL をグループ B_n とする。ここで、B_n 及び A_n の n は read only TSL に混ぜる 1 日当たりのファイルアクセスの回数を示す。n=(1,2,3,4,6,8,12) として各 TSL の面積を計算し、まとめたものが表 7.3 である。

グループ A とグループ B_n を対象として、帰無仮説をグループ A とグループ B_n には差がないとおき、棄却率 1、5、10 %で対応のある t 検定 A を行う。t 検定 A を行った結果が表 7.4 である。表 7.4 より、有意水準 α が 1%で t 検定（A,B_{12}）を行ったとき、また、有意水準 α が 5%で t 検定 (A,B_8),(A,B_{12}) を行ったとき、そして、有意水準 α が 10%で t 検定 (A,B_6),(A,B_8),(A,B_{12}) を行ったとき、帰無仮説は棄却されて対立仮説であるグループ A と B_n には差があるといえる。

t 検定	(A,B)	(A,B_1)	(A,B_2)	(A,B_3)	(A,B_4)	(A,B_6)	(A,B_8)	(A,B_{12})
t 値	1.695849323	0.91277208	0.293923638	-0.243963294	-0.870684481	-2.230325001	-3.557384864	-5.860804847
p 値	0.165156424	0.413000561	0.783436892	0.819261425	0.43306184	0.089578578	0.023641401	0.004230631
有意差（α=1%）	なし	なし	なし	なし	なし	なし	なし	あり
有意差（α=5%）	なし	なし	なし	なし	なし	なし	あり	あり
有意差（α=10%）	なし	なし	なし	なし	なし	あり	あり	あり

表 7.4: t 検定 (グループ A, グループ B_n)

グループ A_n とグループ B を対象として、帰無仮説をグループ A_n とグループ B には差がないとおき、棄却率 1、5、10 %で対応のある t 検定 B を行う。t 検定 B を行った結果が表 7.5 である。表 7.5 より、有意水準 α が 1%で t 検定（B,A_{12}）を行ったとき、また、有意水準 α が 5%で t 検定 (B,A_4),(B,A_6),(B,A_8) を行ったとき、そして、有意水準 α が 10%で t 検定 (B,A_3),(B,A_4),(B,A_6),(B,A_8),(B,A_{12}) を行ったとき、帰無仮説は棄却されて対立仮説であるグループ B と A_n には差があるといえる。

t 検定	(B,A)	(B,A_1)	(B,A_2)	(B,A_3)	(B,A_4)	(B,A_6)	(B,A_8)	(B,A_{12})
t 値	-1.695849323	-1.711472462	-2.020748364	-2.324489603	-2.692622826	-3.290308043	-3.918976015	-5.155595178
p 値	0.165156424	0.162160879	0.113401019	0.080738186	0.054509548	0.030206771	0.017262583	0.006718427
有意差（α=1%）	なし	なし	なし	なし	なし	なし	なし	あり
有意差（α=5%）	なし	なし	なし	なし	なし	あり	あり	あり
有意差（α=10%）	なし	なし	なし	あり	あり	あり	あり	あり

表 7.5: t 検定 (グループ B, グループ A_n)

7.3 実験結果の考察

7.2 節より、10ヶ月の TSL を 5ヶ月ずつに分けて比較を行うことにより、アノマリ検知を行った。よって、攻撃者が 5ヶ月にわたり内部活動を続けた場合に検知ができたことになる。 2.4 節より、攻撃者の潜伏期間は最短で 1ヶ月未満であり、最長で 28ヶ月、平均値が約 8.5ヶ月であり、中央値が 7ヶ月である。よって、攻撃者の潜伏期間中にアノマリ検知することが可能である。また、 7.2 節の t 検定を行うときに用いた TSL の粒度は 1ヶ月である。本実験では、TSL の面積を尺度としてユーザの活動範囲を統計的に見ている。そして、ユーザの活動範囲の粒度を 1 日あたりのファイルアクセス数とおいているため、TSL の粒度を 1 日にしたときも同様の結果が得られることが期待できる。よって、TSL が 1ヶ月分ある場合検定時のサンプル数として約 30 程度確保できる。ただし、TSL の粒度を最小にすると短期的なファイルアクセスによって偏差が生じたときの t 検定への影響が大きくなる。ユーザが所属しシステムの操作を行うことになる組織において、区切りや活動の周期とされている 1 週間または 1ヶ月といった単位が TSL の粒度として妥当である。

t 検定 (グループ A, グループ B_n) とは、2 月から 6 月の read only TSL と 7 月から 11 月の read only TSL に探索的なファイルアクセスを混ぜた TSL との間で t 検定を行っていることを表している。つまり、過去のデータを基準として現在のデータに対してアノマリ検知を行っている。これは、システムの所有者であるユーザのみがシステムを操作していたときに攻撃者が侵入し長期間内部活動を行った場合の t 検定である。表 7.4 から、TSL を蓄積し続けたとき攻撃者による内部活動を検知することが可能であることがわかる。また、t 検定 (グループ B, グループ A_n) とは、7 月から 11 月の read only TSL と 2 月から 6 月の read only TSL に探索的なファイルアクセスを混ぜた TSL との間で t 検定を行っていることを表している。つまり、現在のデータを基準として過去のデータに対してアノマリ検知を行っている。これは、攻撃者がシステムに侵入し長期間内部活動を行い、攻撃者が内部活動を終えた後、システムの所有者であるユーザのみがシステムを操作する場合の t 検定である。表 7.5 から、TSL を蓄積し続けたとき攻撃者による内部活動が行われたことを検知することが可能であることがわかる。「Shady RAT Intrusions in 2006,2007,2008,2009,2010,2011 [4]」より、攻撃者は目的の情報を取得した後、一時的に活動を休止し、再び情報の取得を目的として内部活動を再開することが報告されている。ここで、fspeek を導入した時点でシステムに攻撃者が侵入されていた場合、その攻撃者の活動が休止した時に、蓄積された TSL の比較によって攻撃者を検知することが可能である。ゆえに、一度システムに侵入した攻撃者による 2 回目以降の内部活動による情報取得を未然に防ぐことが可能である。

7.2 節では、有意水準 α に 10%を含めている。統計学において一般に有意水準は 1%または 5%が使われており、10%が用いられることは少ない。有意水準が 10%の場合、10%の誤りを許容することとなるため、有意差への信頼性が低いことが原因である。しかし、fspeek におけるアノマリ検知において重要なことは攻撃者を検知することだと考える。よって、False Positive（α エラー）が上がったとしても検知できる可能性が上がるため、fspeek においては 10%の誤りを許容する。

表 7.2 より、他の月と比べて 6 月の open read の TSL が大きいことが分かる。これは、表 7.1 より TSL 全体のファイルアクセスの幅に依存したものではないことから、ファイルアクセスの傾向によって生じた差である。この 6 月の TSL のパラメータの偏差が大きいことにより、B_n 及び A_n における n=(1,2) のときの比較結果として有意ではないという判定になったと考えられる。これより、ファイルアクセスの範囲に幅がある人物であれば検知することが難しくなる。

7.4 fspeek の性能評価

本節では、fspeek の性能を評価する。評価に用いた PC は以下の構成である。

- ソフトウェア OS X 10.9.5
- プロセッサ 1.8 GHz Intel Core i7（2 コア）
- メモリ 4 GB 1333 MHz DDR3

2015 年 2 月から 11 月の FAL をもとに TSL を作成し、2015 年 2 月から 6 月のグループ A として、2015 年 7 年から 11 月のグループ B として、グループ A とグループ B の TSL の比較を行った。このときのメモリサイズと CPU 使用率と時間は以下のようになった。

- CPU 使用率：平均 100%
- メモリ：平均 2.5GB
- times：real 7m23.422s、user 6m22.476s、sys 0m41.915s

第8章　fspeekの適用性

8.1　監査システムによるFALの違い

本節では監査システムによる FAL の違いについて述べる。Mac OSX(BSD) では Open Basic Security Module(OpenBSM) によって FAL の取得している。以下はユーザ認証が行われたときの BSD Audit の sample である。

BSD Audit Sample

- header,139,11,user authentication,0,Sat Apr 21 22:02:14 2012, + 940 msec
- subject,mako,mako,staff,root,staff,69,100005,69,0.0.0.0
- text,Verify password for record type Users 'mako' node '/Local/Default'
- return,success,0
- trailer,139

BSD audit は最低 header、subject、return の 3 つのトークンで構成される。それに加えて、オプショントークンは"trailer Token" "arbitrary Token" "arg Token" "attr Token" "exit Token" "file Token" "groups Token" "in_addr Token" "ip Token" "ipc Token" "ipc_perm Token" "iport Token" "opaque Token" "path Token" "process Token" "seq Token" "socket Token" "text Token" の 18 種類があり、これらはイベントの種類によって追加される。header から始まり、trailer(option) で終わることが多い。以下は BSD Audit の各属性に関して、何を意味しているのかを記述したものである。

BSD Audit Sample Explanation

header,139,11,user authentication,0,Sat Apr 21 22:02:14 2012, + 940 msec

- "header " = トークン ID
- "139 " = レコード長
- "11 " = レコードのバージョン
- "11 " = レコードのバージョン
- "user authentication " = イベント
- "0 " = イベント ID の修飾子

- ”Sat Apr 21 22:02:14 2012” msec=” + 940 msec ” = 時間

subject,mako,mako,staff,root,staff,69,100005,69,0.0.0.0

- ”subject ” = token ID
- ”mako ” = user audit ID
- ”mako ” = effective user ID
- ”staff ” = effective group ID
- ”root ” = real user ID
- ”staff ” = real group ID
- ”69 ” = process ID
- ”100005 ” = session ID
- ”69 ” = device ID
- ”0.0.0.0 ” = machine ID

text,Verify password for record type Users ’mako’ node ’/Local/Default’

- ”text ” = token ID
- ” Verify password for record type Users ’mako’ node ’/Local/Default ” = text string

return,success,0

- ”return ” = token ID
- ”success ” = error status
- ”0 ” = return value

trailer,139

- ”trailer ” = token ID
- ”139 ” = record length

上記より、linux audit にあって BSD audit にないものは、CPU Architecture、uid、suid(set user id)、fluid(file system user id)、egad(effective group id)、sgid(set group id)、fsgid(file system group id)、key word、ls name、である。そして、書式は異なるが共通項目として以下の項目が挙げられる。タイプ宣言、タイムスタンプ、システムコール、システムコールの成否、exit、引数 parent process、process、tty、session id、command name、command path、current working directory path、inode number、device number、file access mode、である。

5.3 節及び共通項目により、fspeek は BSD または Linux に対応したシステムである。

8.2 SMB/CIFS におけるファイルアクセスログの出力

組織でのシステム運用の形態の 1 つとして、記憶装置がシステム管理者側で一括して管理されている構造であるシンクライアントによる運用がある。本学では、学生と教員用の情報環境をシンクライアントで運用している。本節では、このような実運用されている形態と提案システムとの適合性について述べる。

本研究で提案した fspeek では、 6 章よりユーザが利用しているシステム上で動作させることを前提としており、fspeek をシンクライアント環境の管理側にのみ動作させる環境では正常に FAL を記録することができない。NFS によるファイルシステムへのアクセスの一部が監査システムによって記録されないという問題があるためである。上記の検証のため、クライアント側のファイルシステムをサーバ側にある記憶装置からマウントして使用する環境を構築した。このとき、サーバ側で監査システムを動作させることとする。NFS によるファイルの共有を行ったときに得られた監査システムのログから、結果として、クライアント側でマウントしたファイルシステムへの操作が記録されないことがわかった。つまり、NFS のプロセスによってクライアント側で作成したファイルがサーバ側に共有・記録されたとき、監査システムにおいて、その動作の詳細が FAL に残らない。これは NFS サーバが NFS クライアントからの RPC メッセージを受信したとき、クライアント側のファイルシステムへの操作のために呼び出されるシステムコールが、NFS デーモンによるシステムコールの呼び出しに変更されることで、ファイルアクセスに関する情報の粒度が落ちるためであると考えられる。

第9章　おわりに

はじめに本論では、標的型攻撃の一種の、対象の情報を窃取することを目的とした明確な犯意のある知能的な活動である APT 攻撃による情報漏えい対策について述べた。情報漏えい対策として、入口対策、出口対策、内部対策の 3 つが挙げられる。その中で特に長期的で持続的な内部活動の検知の必要性を提起した。内部活動を含むユーザのファイルシステム上でのソフトウェア操作と記録される FAL との関係を明らかにした。その関係性について考察を行い、FAL を用いたアノマリ検知に必要な要素を列挙し、FAL を持続的に記録するための手法の提案を行った。その提案を実現するために FAL を構造化させたものである TSL を考案した。また、TSL を用いた演算に関して定義を行った。この TSL の演算を用いたアノマリ検知を行うために、ファイルアクセスにおけるユーザの活動傾向や FAL のパターン等に関する考察を行い、結果得られた推論をもとにファイルアクセスに対するアノマリ検知手法の提案を行った。

提案手法を実現するために、BSD 上で FAL から TSL を構築し、TSL を蓄積し、その TSL の比較を行うことでアノマリ検知を行う fspeek の設計と実装を行った。fspeek における APT 攻撃の検出手法として、ファイルアクセスのアクセス範囲を尺度としてアノマリ検知を行う File Access Scope T-test（FAST）を提案した。また、fspeek はユーザが利用しているシステム上で常時動作させることを前提としており、長期的に TSL を記録できるように記録するデータ量は FAL よりも小さく抑えている。

fspeek を利用して、2015 年 2 月から 11 月の 10ヶ月分の FAL に対して提案手法をもとに実験を行った結果、ユーザのソフトウェア操作によるファイルシステムへのアクセスの範囲から、アノマリ検知が可能となった。

APT 攻撃の内部活動における長期的で探索的なファイルアクセスが検知可能となった。機密情報が攻撃者に取得される前に、攻撃者の長期的な内部活動を検知できれば、攻撃者が侵入しているシステムをネットワークから隔離することで、情報漏えいを防ぐことが可能となった。また、システムに fspeek を導入した時点で、そのシステムが攻撃者に侵入されていた場合、その攻撃者の活動が停止したときに、蓄積された TSL の比較によって攻撃者が活動停止以前に行っていた内部活動の検知が可能となった。ゆえに、一度システムに侵入した攻撃者による 2 回目以降の内部活動による情報窃取を未然に防ぐことが可能となった。以上より、APT 攻撃における内部活動を検知することで、情報窃取による情報流出の被害を軽減させることが可能となった。また、本研究の提案した fspeek は、FAL をもとにした TSL を年単位で記録することが可能であり、システムを操作しているユーザに普段行われない活動が含まれていないかどうかを検知するものである。ゆえに、fspeek

を稼働させることで、サイバーセキュリティにおける真正性および責任追及性の向上も期待できる。

参考文献

[1] 「Google」 https://www.google.com/

[2] 「Yahoo!」 https://www.yahoo.com/

[3] McAfee Labs, McAfee Foundstone Professional Service 「Protecting Your Critical Assets -Lesson Learned from ”Operation Aurora”-」 McAfee, 2010

[4] Dmitri Alperovitch, Vice President, Threat Research 「Revealed: Operation Shady RAT」 McAfee, 2011

[5] Information-technology Promotion Agency（IPA） 「情報セキュリティ白書 2015」 https://www.ipa.go.jp/security/publications/hakusyo/2015.html

[6] Matthew J. Schwartz 「RSA SecureID Breach Cost $66 Million」InformationWeek, 2011

[7] Adam Rosenberg「Sony Network Breach to Cost Company $170 million」Digital Trends, 2011

[8] National center of Incident readiness and Strategy for Cybersecurity（NISC） 『「標的型不審メール攻撃訓練」結果の中間報告』 http://www.nisc.go.jp/active/general/pdf/hyoutekigata_120119.pdf

[9] Jeremy Allison 「pwdump」 https://download.samba.org/pub/samba/pwdump/

[10] Arnaud Pilon 「cachedump」 https://packetstormsecurity.com/files/author/3874/

[11] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman「Protecting against unexpected system calls」SSYM’05 Proceeding of the 14th conference on USENIX Security Symposium - Volume 14, 2005

[12] 伊波 靖, 高良 富夫「危険なシステムコールに着目した Windows 向け異常検知手法」情報処理学会 Vol.50 No9 2173-2181,2009

[13] 向井 康貴「ファイルシステムの特性を活かした APT 攻撃検出に関する研究」 北陸先端科学技術大学院大学, 2014

[14] 「サイバーセキュリティ基本法」 http://law.e-gov.go.jp/htmldata/H26/H26HO104.html

[15] redhat 「Understanding Audit Log Files」 https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/sec-Understanding_Audit_Log_Files.html

[16] Sebastien Martini 「pyinotify」 http://seb.dbzteam.org/pyinotify/

[17] Zhang, Kaizhong and Shasha, Dennis 「Simple fast algorithms for the editing distance between trees and related problems」 SIAM journal on computing, 1989

[18] Chawathe, Sudarshan S and Rajaraman, Anand and Garcia-Molina, Hector and Widom, Jennifer 「Change detection in hierarchically structured information」 ACM SIGMOD Record,1996

[19] Chawathe, Sudarshan S and Garcia-Molina, Hector 「Meaningful change detection in structured data」 ACM SIGMOD Record,1997