ファイルイベント監視とホワイトリストに基づく攻撃プロセス検知

山下　慧† HieuHanh Le† 横田　治夫†

† 東京工業大学情報理工学院　〒 152-8550 東京都目黒区大岡山 2-12-1
E-mail: †{yamashita,hanhlh}@de.cs.titech.ac.jp, ††yokota@cs.titech.ac.jp

あらまし　IoT デバイスの増加により IoT デバイスのマルウェア対策の重要性が増している．しかし，IoT デバイスにはリソースに制限がありその中で保護を提供する必要がある．マルウェアの検出手法の一つとしてはプロセスによるファイルシステムイベントを監視し判定を行うことでマルウェアの活動を検出するものが存在し，その判定方式の一つにホワイトリスト方式が存在する．ホワイトリストの短所となる更新は IoT では少なく，その長所の軽量性を活かせることから IoT デバイスに適していると考えられる．本稿では IoT デバイスにおける制限されたリソースの下でホワイトリスト方式とファイルシステムイベント監視を利用した性能の良い攻撃プロセスの検知を行うことを目的として，プロセス ID のキャッシュとハッシュインデックスへの最小完全ハッシュ関数の適用を提案し評価を行った．

キーワード　改ざん検出, ホワイトリスト, ファイルイベント監視

1. はじめに

1.1 研究背景

近年，マルウェアの数は急速に増加を続けている [1]．また，IoT デバイスの増加によりリソースに制限のある IoT デバイスにおけるマルウェア対策も重要性を増している．

マルウェアの検出の手法としては従来の手法であるブラックリスト方式ではなく，ホワイトリスト方式を利用した検出方法も提案されている [2]．ブラックリスト方式ではマルウェアをリストの要素とし，一致したものを拒否，一致しなかったものは検査を行ったうえで許可をする．ホワイトリスト方式ではアクセスを許可するアプリケーションをリストの要素とし，一致したものを許可，一致しなかったものは拒否をする．ブラックリスト方式の長所としてはソフトウェアの追加，更新時のリストの更新が不要かつリストの自動更新が可能であることがある．短所としては定期的なリストの更新や膨大になったリストサイズがデバイスへのオーバーヘッドとなることやブラックリストのみではゼロデイ攻撃を検出できない点が挙げられる．特に IoT デバイスにおいてはオーバーヘッドの大きさが問題になると考えられる．対してホワイトリストの長所としては，定期的なリストの更新が不要のためオーバーヘッドが少なく，ゼロデイ攻撃を検出可能であることが挙げられる．特に，オーバーヘッドの少なさがリソースに制限のある IoT デバイスに適している．短所としてソフトウェアの追加，更新の度に管理者によるリストへの追加，削除，変更が必要となるが，IoT デバイスでは動作するアプリケーションの少なさと更新頻度の低さから通常の利用環境と比較して影響を抑えることができると考えられる．よって，IoT デバイスにおけるマルウェア対策としてホワイトリストは適していると考えられる [3]．

マルウェアに対する対策のための手法は数多く提案されており，その中の一つとしてプロセスによるファイルイベントを監視し，マルウェアの活動を検出するものが存在する．しかし，既存のファイルイベント監視を利用したマルウェア対策手法では、ブラックリスト方式を採用しており [4]，我々の知る範囲ではホワイトリストと組み合わせて攻撃プロセスの検知を行っているものは存在していない．これは，一般の環境では，ホワイトリストの更新コストが大きいことが影響していると考えられるが，前述したように IoT デバイスではそのコストが小さくなるため，IoT デバイスに特化した場合には有効であると考えられる．

1.2 研究の目的

本研究では IoT デバイス，特に Linux を採用した IoT デバイスにおける制限されたリソースの下で性能の良い攻撃プロセスの検知を行うことを目的とする．特に，アクセス判定速度の向上や CPU 使用率，メモリ使用量の削減を目的とする．

本研究では，保護のアプローチとしてホワイトリスト方式を採用する．また，ホワイトリストの探索コストを削減するために二度目以降のアクセス許可処理の高速化のためのプロセス ID のキャッシュと最小完全ハッシュ関数のハッシュインデックスへの利用をすることを提案する．その理由としてはプロセス ID のキャッシュに関しては同一プロセスからの 2 回目以降のアクセスについて判定速度の向上、CPU 使用率の軽減が期待できること，最小完全ハッシュ関数に関しては衝突が生じないことと最低限のインデックス数で動作することからメモリ効率や計算速度の向上が期待できることが挙げられる．その効果を確認するため提案した手法について評価を行う．

1.3 本稿の構成

本稿は以下の通りに構成される．2 節では背景となる知識について解説を行い，3 節は関連する研究について述べる．4 節では，本研究における攻撃プロセスの検知手法とファイルシステムイベントとホワイトリストの照合を高速化する手法を提案する．そして，5 節で実装にかかわる部分について説明を行う．6 節では，提案手法の実験と考察を行う．最後に 7 節で本稿の結論を述べる．

```
struct fanotify_event_metadata {
    __u32 event_len;
    __u8 vers;
    __u8 reserved;
    __u16 metadata_len;
    __aligned_u64 mask;
    __s32 fd;
    __s32 pid;
};
```

図 1　構造体 fanotify_event_metadata

```
struct inotify_event {
    int      wd;
    uint32_t mask;
    uint32_t cookie;

    uint32_t len;
    char     name[];
};
```

図 2　構造体 inotify_event

2. 背 景 知 識

本節では，提案手法と実装にて用いるハッシュ関数と Linux 上でのファイルシステムイベントの監視機能を提供する API についての説明を行う．

2.1 最小完全ハッシュ関数

最小完全ハッシュ関数（MPHF）は衝突が生じず，空間効率に優れたハッシュ関数であり，ハッシュインデックスに利用した際にサイズとアクセス時間を大きく抑えることができる [5]．キーを S の要素とすると，完全ハッシュ関数は $S \subseteqq U$ を $[0, m-1]$ の範囲の一意の値へと写像を行う関数である．この時，$n = |S|$ と $m = |U|$ について，$m \geqq n$ となる．MPHF は，$m = n$ となる完全ハッシュ関数である．この定義より MPHF を利用したハッシュインデックスは衝突が生じず，最低限のインデックス数で動作を行うことができる．欠点として，MPHF は事前に定義されたキーに基づいてハッシュ関数を生成するため，キーに変更が生じた際にはハッシュ関数を再生成する必要がある．ハッシュインデックスのキーとしてホワイトリストの保護対象ファイルを指定した場合，IoT デバイスでは環境の更新頻度が低いと考えられるためキーの更新頻度も低くなると考えられる．したがって，IoT デバイスの保護を想定した状況の下では MPHF の再作成頻度も抑えられると考えられる．

2.2 ファイルシステムイベント監視機能

2.2.1 fanotify

fanotify は Linux 上に実装されたファイルシステムイベントの監視機能を提供する API である [6]．`fanotify_init()` システムコールで fanotify 通知グループを作成・初期化後 `fanotify_mark()` システムコールでファイル・ディレクトリを監視対象として追加することができる．監視対象のファイルにイベントが発生すると，`fanotify_init()` の返り値であるファイルディスクリプタからイベントの内容を構造体として読み出すことができる．図 1 に示した読み出された構造体 `fanotify_event_metadata` には，発生したイベントの内容，アクセスされたオブジェクトに対するオープンされたファイルディスクリプタ，アクセスを行ったプロセス ID などの情報が含まれている．

また，`fanotify_mark()` での追加時に引数としてオプションを指定することで，イベントの発生時にアクセスの許可の判定を行うことができる．アクセス許可の判定を行う場合，通知の発生後アクセスの許可・不許可の情報を含む構造体を `fanotify_init()` の返り値のファイルディスクリプタに書き込むことで指示を出すことができる．

fanotify の短所として監視対象ディレクトリ内でのファイルの作成・削除，監視対象ファイル・ディレクトリ自身の消去・移動・名称の変更を始めとした一部のイベントの監視をサポートしていない点が存在する．

2.2.2 inotify

inotify は fanotify と同じく Linux 上に実装されたファイルシステムイベントの監視機能を提供する API である [7]．`inotify_init()` システムコールで inotify インスタンスを作成後，`inotify_add_watch()` システムコールで監視対象リストを追加することができる．inotify がファイルシステムイベントを通知すると `inotify_init()` の返り値であるファイルディスクリプタから図 2 イベントの内容を表す構造体 `inotify_event` を読み出すことができる．

inotify は fanotify と異なり，監視対象にアクセスを行ったプロセス ID を取得することができない．しかし，fanotify では検知することができない監視対象ディレクトリ内でのファイルの作成・削除，監視対象ファイル・ディレクトリ自身の消去・移動・名称の変更・メタデータの変更を検知することができる．

3. 関 連 研 究

マルウェアの検出にホワイトリストを適用した研究の例としては，文献 [8] が存在する．この研究では米国空軍の地上支援装置の保護を目的としてアプリケーションの実行の制御にホワイトリスト方式を採用している．この研究では，ソフトウェアの実行時に読み込んだコードからハッシュを生成し、ホワイトリストのエントリと比較することでソフトウェアの実行を許可するかどうかを決定している．また，ホワイトリストの動的な更新を安全に行うための手法としてデジタル署名を利用した手法を提案している．組込みシステムのマルウェア対策についてホワイトリストの利用を考察した文献 [3] においても同様にデジタル署名を利用した更新方法が提案されている．これらの研究ではアプリケーションの実行の制御にホワイトリストを利用している．ファイルシステムイベントを監視することで検出を試みる本研究とは差異が存在する．

また，文献 [9] ではアプリケーションホワイトリストの設計と実装のアプローチに関する提案を行っている．この中で，アプリケーションホワイトリストデータベースのエントリについ

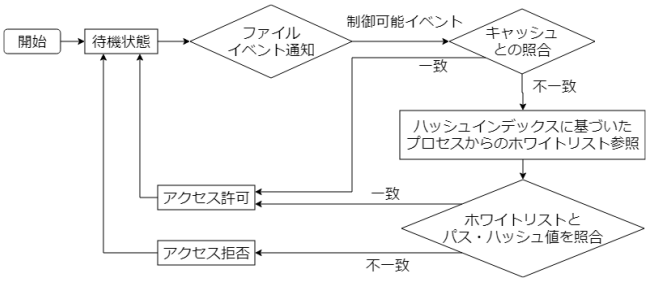

図 3　改ざん検出プログラムの概略のフローチャート

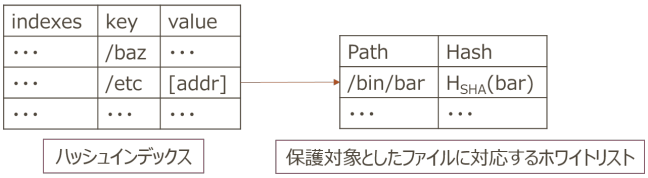

図 4　ハッシュインデックスと対応するホワイトリスト

てアプリケーションのファイル名・ファイルパスだけでなく、アプリケーションファイルのハッシュ値や発行元といったパラメータを保持することで安全性を高めることができるとしている．しかしながらこの研究では提案にとどまっており，実装と評価は行っていない．

文献 [10] では IoT デバイスに対する脅威を分類し，分析を行っている．その中で，IoT を構成するデバイスは外部や内部からの攻撃に対して脆弱であり保護する必要があるにもかかわらず，計算能力，メモリ，電源に関してリソースの制約があるために複雑なセキュリティメカニズムの実装，運用が困難であることを指摘している．

ファイルシステムイベント監視に利用し悪意のあるプロセスのアクセスを検出するアンチウイルスソフトウェアとしては F-secure が存在する [4]．F-secure では fanotify によるファイルシステムイベント監視とブラックリスト方式を利用して，ファイルにアクセスしようとするプロセスを検出しそのプロセスがマルウェアであるかどうかの検査を行っている．F-secure ではブラックリストを採用しているが，動作するアプリケーションが限られており動作のリソースも限られている IoT デバイスにおいてはホワイトリストが適していると考えられる．

4. 提 案 手 法

本節では，ファイルシステムイベント監視とホワイトリストに基づいた改ざん検出手法とその照合処理の高速化手法，そしてホワイトリストの安全な更新手法について説明する．まず判定手法としては，軽量であることや IoT デバイスではホワイトリストの設定，更新コストが軽減されることを考慮してホワイトリスト方式を利用することとする．そして，ホワイトリストの探索コストを軽減するためにハッシュインデックスとプロセス ID のキャッシュを行う．

4.1 ファイルシステムイベント監視とホワイトリストに基づいた改ざん検出手法

図 3 に改ざん検出プログラムの概略を示している．ホワイトリストに基づいて保護対象をファイルシステムイベント監視機能に登録し、保護対象のパスと対応するホワイトリストのエントリからハッシュインデックスを生成する．図 4 にホワイトリストから生成されるハッシュインデックスと対応するエントリの例を示している．ハッシュインデックスでは保護対象となるファイルパスをキー，保護対象ファイルパスに対応するエントリへのアドレスを値とする．エントリではアクセスを許可するアプリケーションの実行可能ファイルパスとそのハッシュ値を格納している．

ハッシュインデックスとエントリの作成が終了すると，プログラムはファイルシステムイベントの検出待ち状態へと移行する．ファイルシステムイベントが通知されると最初にキャッシュされたプロセス ID との照合を行う．(4.2.1 節と 5.3 にて詳細を説明）キャッシュされていなかった場合は，ファイルシステムイベントの情報を基にハッシュインデックスを利用してホワイトリストを参照し，アクセスされたファイルに対応するエントリを取得する．そして，アクセスしようとするプロセスのパスとハッシュ値に対して取得したエントリを照合しアクセスを許可するかどうかの判定を行う．(5.3 節にて詳細を説明）

4.2 ホワイトリストの探索コストの軽減手法

4.2.1 プロセス ID のキャッシュ

一度アクセスを許可されたプロセスは再度同一のファイルにアクセスを要求した際に高速に処理を行うことができるようキャッシュを行う．キャッシュはメモリ上にのみ置かれ他のプロセスからの書き換えの対象にならないと仮定する．このキャッシュのパラメータにはプロセス ID の再利用が発生した場合にキャッシュ内のプロセスとアクセスを行おうとするプロセスが同一であることを確認するためにプロセスの起動時刻を加える．キャッシュがされた状態で再度同一のプロセス ID かつ同一の実行可能ファイルパスからアクセスの要求が発生すると，プログラムはキャッシュされたホワイトリスト内のプロセス起動時刻と現在のプロセス ID に対応するプロセスの起動時刻が一致するかを検証し，一致すればアクセスを許可する．

4.2.2 最小完全ハッシュ関数のハッシュインデックスへの適用

ハッシュインデックスの検索を高速化するために最小完全ハッシュ関数（MPHF）をハッシュインデックスに適用する．最小完全ハッシュ関数は 2.1 節で説明した通り，ハッシュ関数が単射かつキーの数とインデックス数が一致するので衝突が起きない．処理速度の面では衝突が起きないことによる検索速度の向上が期待できる．また，メモリ効率の面では通常のハッシュ関数が load factor が一定値以上になるとリハッシュを行う必要があるのに対し，MPHF では load factor が 100 %の状態で運用をすることができるため空間効率に優れておりメモリ効率の向上が期待できる．事前に定義されたキーに基づいて MPHF を生成するためにキーに変更が生じると MPHF を再生成する必要があるという欠点も IoT デバイスにおけるホワイトリストの更新頻度の低さから抑えられると考えられる．

ハッシュインデックスの作成時に，ホワイトリストに基づいて MPHF を作成する．4.1 節のハッシュインデックスに対応

するように，キーを保護対象となるファイルパス，エントリを保護対象ファイルパスに対応するエントリへのアドレスとしている．

ホワイトリストが動的に更新された際の対応処理は更新部分によって変化する．エントリ部分のみに変更が生じた場合にはハッシュインデックスは更新前の MPHF をそのまま利用することができ，エントリのみを更新することでアップデートができる．しかし，監視対象とするファイルの記述に変更が生じた際にはキーに変更が生じるため MPHF を再作成する必要がある．しかし，前述したように IoT デバイスにおけるホワイトリストの更新頻度の低さからこの欠点の影響は抑制できると考えられる．

4.3 ホワイトリストの更新手法

ホワイトリストの更新には文献 [3] [8] と同様にデジタル署名を利用する．利用する管理者は公開鍵と秘密鍵を用意して作成したホワイトリストに署名を行い，秘密鍵は管理者が保護対象のデバイスとは異なる場所に保存する．改ざん検出プログラムがホワイトリストを読み込む際に公開鍵と合わせてデジタル署名を検証しホワイトリストが正常な状態に保たれていることを確認する．デジタル署名を利用することでファイルシステムイベント監視を停止すること無くホワイトリストのアップデートを行うことができる．それにより，更新のためにファイルシステムイベント監視を停止している間にデバイスが脅威にさらされることを防ぐことができる．

5. 実　　装

この節ではプロセス検知プログラムの実装の細部についての説明を行う．

5.1 ファイルシステムイベントの通知

ファイルシステムイベントの通知手法としては fanotify と inotify を利用する．fanotify と inotify において定義されているイベントの中でマルウェア対策としてのファイルシステムイベント監視において重要性が高いものは以下が考えられる．

- OPEN : ファイルやディレクトリのオープン
- ACCESS : read() などによるファイルへのアクセス
- MODIFY : write() などによるファイルへの変更
- DELETE : 監視対象ディレクトリ・ファイルや監視対象ディレクトリ内でのファイル・ディレクトリの削除
- MOVE : ディレクトリ・ファイルの移動やファイル名の変更
- ATTRIB : アクセス許可などのメタデータの変更

このうち，fanotify でアクセス制御をすることができるのは OPEN, ACCESS であり，OPEN に対して制御を行うことによって OPEN，ACCESS，MODIFY に対して制御を行うことができる．しかし，DELETE，MOVE，ATTRIB については fanotify ではアクセスをイベントの検出を行うことはできず，よってイベントの制御をすることもできない．この問題を解決するために本研究では fanotify と inotify を併用し，fanotify に検出，制御ができないファイルシステムイベントについては inotify を利用して検知，ログ出力のみを行う．

表 1　キャッシュに格納されるパラメータの例

プロセス ID	被アクセスパス	アプリケーションの実行可能ファイルパス	プロセス開始時刻
10	/etc/foo	/bin/bar	1000000

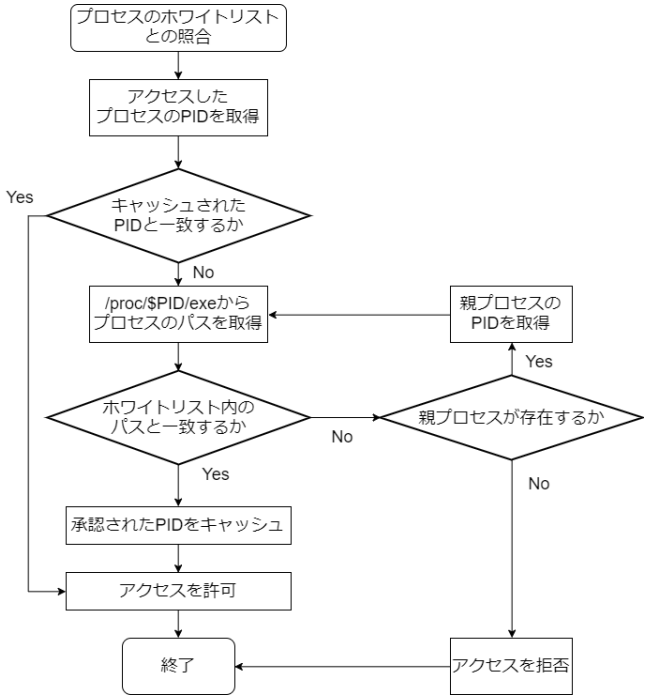

図 5　プロセスとホワイトリストの照合処理のフローチャート

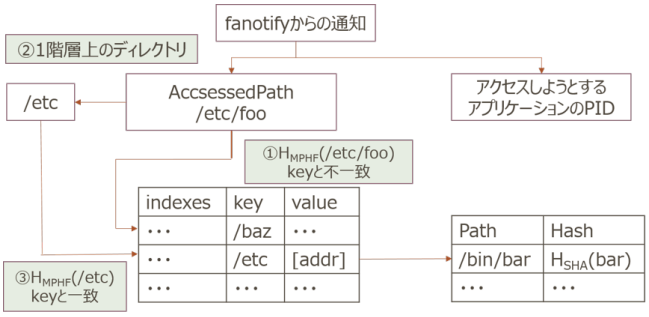

図 6　監視対象に対応するアプリケーションホワイトリストの取得

5.2 キャッシュのパラメータ

4.2.1 節で説明したキャッシュの実装において，表 1 に実際に格納するパラメータの例を示している．キャッシュデータはプロセス ID を key とした B+ tree で管理される．プロセスの開始時刻は/proc/$PID/stat ファイル内の情報から取得をする．キャッシュ内のリストは定期的にプロセスが生存しているかを確認し，プロセスが終了していた場合はキャッシュから消去する．

5.3 アクセス制御の判定

図 5 にプロセスとホワイトリストの照合処理のフローチャートを示している．プロセスのホワイトリストとの照合処理が開始すると，fanotify から通知されたプロセス ID を取得する．そして，以前に同じプロセス ID がアクセスを行いキャッシュされているかどうかを照合する．キャッシュされていた場合は，4.2.1 節の処理へと移行する．

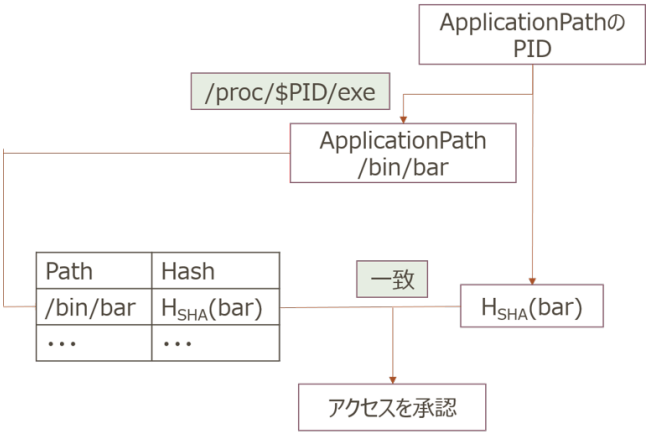

図 7　アプリケーションホワイトリストとアプリケーションの照合

キャッシュがされていなかった場合，メモリ上のホワイトリストとの照合を行う．図 6 に以下の例における監視対象に対応するアプリケーションホワイトリストを取得するまでの流れを示している．

- 保護対象とするディレクトリ：/etc
- アクセスされたファイルのパス（便宜上 AccessedPath と呼ぶ）：/etc/foo
- 保護対象のファイルにアクセスしたアプリケーションのパス（便宜上 ApplicationPath と呼ぶ）：/bin/bar

まず，AccessedPath について MPHF によりハッシュ値を求め，AccessedPath の MPHF ハッシュ値とハッシュインデックスのキーを照合する．一致しなかった場合は AccessedPath の一つ上の階層のパスについて MPHF のハッシュ値を求めファイルパスの MPHF ハッシュ値とキーを照合し，一致しなければさらに一つ上の階層について同様の処理を一致するまで繰り返す．一致した場合は，対応する value に格納されたアドレスから対応するアプリケーションホワイトリストの一覧を取得する．proc ファイルシステムは Linux システム上の情報を含む疑似ファイルシステムである [11]．/proc/$PID/exe はプロセス ID に対応する実行可能コマンドのパス名へのシンボリックリンクとなっている．また，/proc/$PID/stat にはプロセスの状態に関する情報を含んでおり，親プロセスの ID を取得することができる．図 7 に取得したアプリケーションホワイトリストとアクセスしようとするアプリケーションの照合処理を示している．fanotify から通知された監視対象にアクセスしようとするアプリケーションのプロセス ID について，/proc/$PID/exe から ApplicationPath を取得する．そして，取得した ApplicationPath とアプリケーションホワイトリスト内のパスに一致するものがあるかを照合する．一致した場合，ホワイトリスト内のその他のパラメータと ApplicationPath から算出したパラメータが一致するかを確認し，一致すればアクセスを許可する．一致するものがなければ親プロセス ID を/proc/$PID/stat から取得し，親プロセス ID について同様にアプリケーションホワイトリストとの照合を一致するか親プロセスが取得できなくなるまで繰り返す．親プロセスが取得できなかった場合，ホワイトリストに登録されていないプロセスがアクセスを行おうとしているものとしてアクセスを拒否する．

表 2　実験用プログラム

	CHD（MPHF）	libghthash（通常）
キャッシュ有	プログラム 1 ■	プログラム 2 ■
キャッシュ無	プログラム 3 ■	プログラム 4 ■

6. 実験と考察

6.1 実験目的

提案手法の適用による処理速度，CPU 使用率，メモリ使用量の差異を確認し評価する．

6.2 実装プログラム

実験に用いるプログラムとして表 2 に示すようにハッシュテーブルに利用する関数の種類とプロセス ID キャッシュの有無の 2 つの条件に基づいて 4 つのプログラムを実装した．MPHF を利用したハッシュインデックスには CMPH ライブラリ内の CHD アルゴリズム [12] [13] を，通常のハッシュインデックスには libghthash ライブラリを利用して実装を行った [14]．デジタル署名の方式としては SHA256 with ECDSA を利用した．

また，実装におけるハッシュ値の計算には文献 [15] において衝突耐性を必要とするアプリケーションにおいて SHA-2 以上の強度を持つハッシュ関数の利用が推奨されていることを考慮して SHA-256 を利用している．

6.3 実験方法

実験を行う環境を以下に示す．

- デバイス：Raspberry Pi 3 Model B
- OS：Raspbian 4.9.77-v7+
- CPU：Broadcom BCM2837 1.2GHz 64-bit quad-core ARMv8 Cortex-A53
- RAM：1GB

また，raspbian のデフォルト設定では fanotify によるアクセス制御が許可されていない．よって，カーネルコンフィグを操作して有効化している．

また、プログラムのホワイトリストの設定として表 3 に示すように 3 つのホワイトリストを用意した．

実験においては以下の項目において，提案手法と比較対象の実行時間の 10 回の平均値を計測した．

- プロセスからのアクセスイベントの拒否の処理時間
  - 拒否の処理時間の測定時にアクセスさせたアプリケーションはホワイトリスト内のファイルパスとアクセスを試みるプロセスの実行可能ファイルのパスが一致しないアプリケーションである．よって，ホワイトリストとの照合処理においてはホワイトリスト内のファイルパスとの比較の時点でホワイトリストに含まれていないアプリケーションであることが判明するため，実行可能ファイルのハッシュ値の計算は行われない．
- プロセスからのアクセスイベントの許可の処理時間（実験結果においては括弧内に SHA-256 の計算時間を除いた処理時間を示す）
- 提案手法に基づいてキャッシュに登録されたプロセスと同一のプロセスからのアクセスイベントの許可の処理時間

表 3　実験用ホワイトリスト

ホワイトリスト	保護対象パス	説明	ハッシュテーブルインデックス数	保護対象ファイル数	ホワイトリストテーブル要素数
保護対象縮小（W1）	/bin，/sbin ディレクトリのコピー，/boot，アクセス制御テスト用ファイル	基本とする環境から保護対象ファイル数を絞った状況を想定	4	628	7
想定保護対象（W2）	/bin，/sbin，/usr/bin，/usr/s-bin，/etc ディレクトリのコピー，/boot，アクセス制御テスト用ファイル	IoT デバイスにおいて基本となる動作環境を想定	7	7356	13
保護指定粒度細分化（W3）	/bin，/sbin，/usr/bin，/usr/sbin ディレクトリのコピー，/etc ディレクトリのコピーの直下の全てのディレクトリと/boot，アクセス制御テスト用ファイル	基本とする環境からホワイトリストについてより詳細な指定を行った環境を想定	205	7356	409

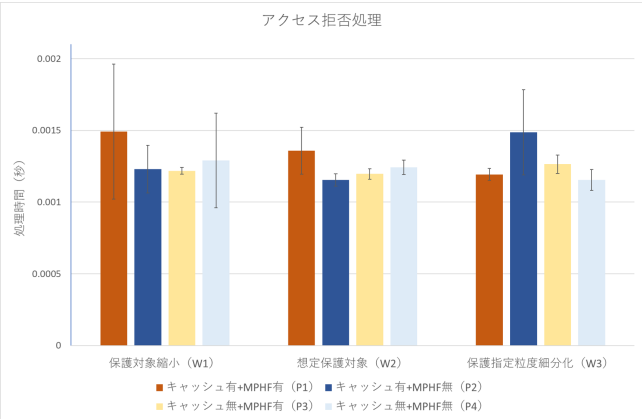

図 8　アクセスイベントの拒否処理時間

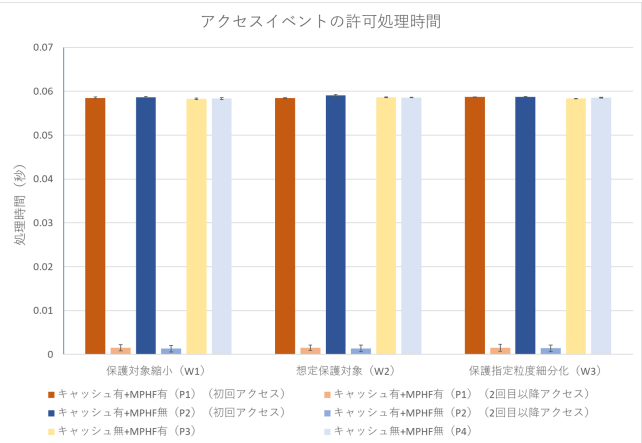

図 9　アクセスイベントの許可処理時間

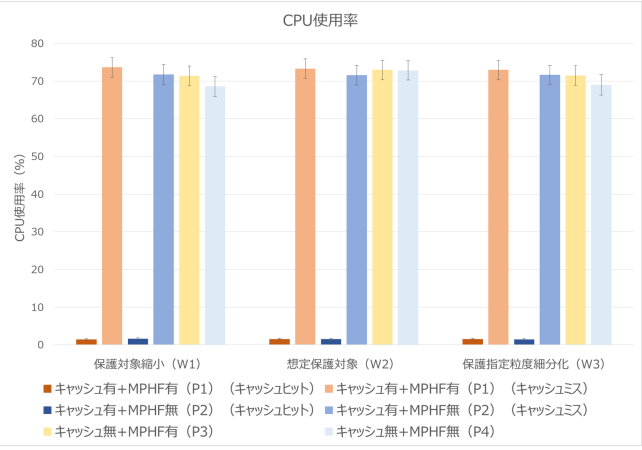

図 10　許可処理中の CPU 使用率

- プログラム開始時からホワイトリストのロードと inotify，fanotify への登録を終えて待機状態に入るまでの処理時間
- ホワイトリストのアップデートの処理時間（ハッシュテーブルは更新せずエントリのみを更新する）
- ホワイトリストのアップデートの処理時間（ハッシュテーブルの更新も行う）

また，待機状態における物理メモリの使用量の 10 回の測定の平均値をそれぞれの条件に対して計測した．

そして，提案手法における CPU 使用率を測定するため同一プロセスから 1 秒毎に 54 回の書き込みアクセスを発生させるプログラムを作成した．このプログラムから保護対象としたファイルに対して書き込みを行い動作中の CPU 使用率を測定した．0.1 秒毎の計測を 100 回行い，これを 10 回繰り返し合計 1000 回の測定の平均値を計測した．また，キャッシュを実装しているプログラム 1，2 についてはキャッシュの実装による負荷への影響を確認するため，キャッシュへの格納とアクセス時のキャッシュの検索は行うがヒットした場合においてもキャッシュを利用せずハッシュインデックスへの照合へ移行することでキャッシュミスを再現したプログラムを作成し測定を行った．

6.4　実 験 結 果

実験結果を図 8〜13 に示す．これらのグラフでは，エラーバーで平均値に対する 95 %の信頼区間を示している．図 8 より、アクセスイベントの拒否処理時間についてはまれに外れ値と見られる大きい値が生じるが、プログラムの差異が原因と考えられる差は生じなかった．

図 9 にはアクセスイベントの許可処理時間をグラフ化したものを示している。通常のアクセスイベントの許可処理時間にお

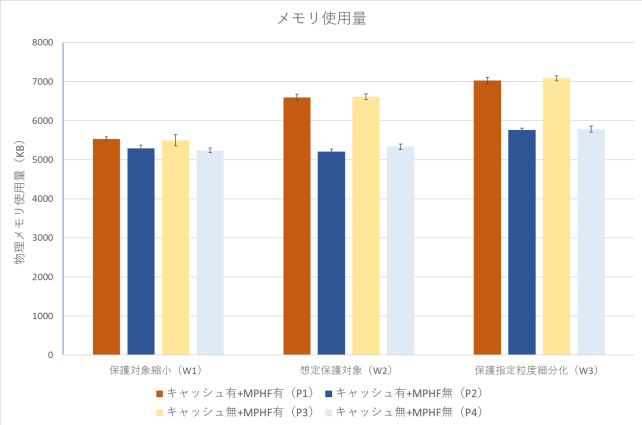

図 11　物理メモリの使用量

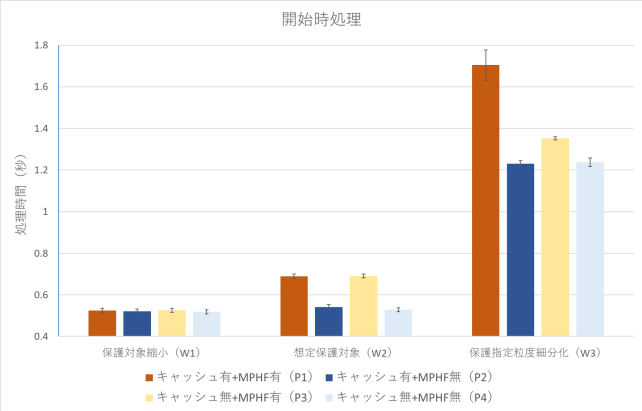

図 12　プログラム開始時の処理時間

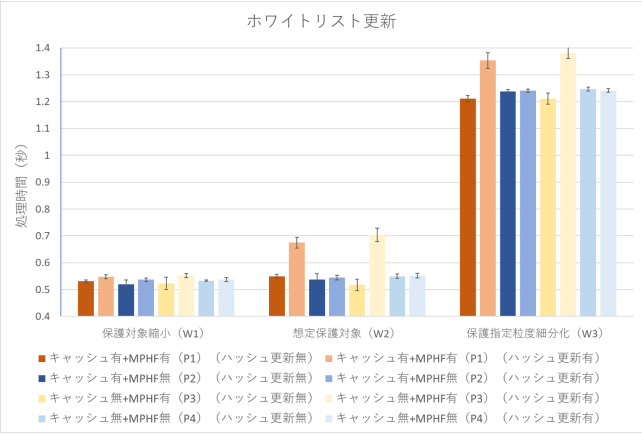

図 13　ホワイトリスト更新時の処理時間

いてはプログラム 1〜4 の間には有意な差は見られなかった．また，通常のアクセスイベントの許可処理時間において SHA-256 ハッシュ値の計算時間はすべてのプログラムで 99 %以上を占めていた．プロセスの起動時刻をパラメータとしたキャッシュではキャッシュを利用しないアクセス許可イベントよりも約 40 倍の速度の約 0.0015 秒まで処理時間を短縮することに成功している．

図 10 には同一プロセスからの連続した許可処理中の CPU 使用率の平均値を示している．MPHF の有無による有意な差は生じなかったが，キャッシュを採用したプログラムでは平均が約 1.5 %と採用しなかった場合の約 70 %から大きく減少している．また，プログラム 1，2 のキャッシュ無効の要素から分かるようにキャッシュ処理の実装による負荷の変化もキャッシュ無（プログラム 3，4）と比較して誤差範囲に収まっている．

図 11 に結果を示した物理メモリ使用量については MPHF 無（プログラム 2，4）より MPHF 有（プログラム 1，3）の方が多く，その差は最大約 27 %増と特にハッシュインデックス数が増えるホワイトリスト 2，3 について大きくなっている．

図 12，図 13 に示したように、プログラムの開始時の処理とハッシュインデックスの更新を含むホワイトリストのアップデートの処理時間については，MPHF 無（プログラム 2，4）より MPHF 有（プログラム 1，3）の方が総じて増加している．

6.5　考　　察

処理速度，メモリ使用量の両面から評価するとキャッシュ有，MPHF 無のプログラム 2 が最も優れていることがわかった．

プロセス ID のキャッシュはアクセスの許可処理の処理時間を約 1/40，CPU 使用率についても約 1/47 と大幅に減少させることに成功したが，この要因は通常のアクセスイベントの許可の処理時間のほとんどを占めるハッシュ値の計算を省略できたことにあると考えられる．キャッシュの実装による負荷への影響もほぼ生じていないことが分かった．

提案手法の一つであった MPHF を実装したプログラム 1，3 については，処理速度と CPU 使用率，メモリ使用量の全てにおいて有効な結果を得ることができなかった．プログラム 2，4 の実装で利用したハッシュインデックスのライブラリにおいては競合の発生時に連鎖法を用いて実装している．また，このライブラリの実装においてはハッシュテーブルへの挿入時に malloc でメモリを割り当て，割り当てたメモリへのポインタを value として格納している．そのため，事前の予測ではハッシュテーブルのメモリ使用量においては必要な分だけメモリの割り当てが生じるためメモリ使用量の減少量は小幅になるものの，計算時間においては衝突が生じることが無く必ず 1 回のハッシュ値の計算で対応する key を得ることができる MPHF の方が高速になるのではないかと予測していた．

メモリ使用量が MPHF 有（プログラム 1，3）の方が多くなっている点については，ハッシュインデックス数が増加するホワイトリスト 2，3 について差が広がっていることからハッシュインデックスに用いる関数の違いに原因があると思われる．推測としては提案手法の実装においてはホワイトリストに対応した MPHF を生成後メモリにロードして利用しており，このロードした MPHF のサイズがメモリ使用量の差につながっている可能性がある．また，処理速度に有意差が生じなかった原因としては文献 [13] 内の実験で CHD アルゴリズムがキー数が 500 万個の環境下で実験を行っていることから，CHD アルゴリズムが高速に動作する環境として想定しているハッシュテーブルインデックス数が今回の提案手法で想定していたインデックス数よりも遥かに多いことが原因と考えられる．この点についてインデックス数を 7399 まで大幅に増加させたホワイトリストについても実験を行ったもののやはり処理速度に有意な差は生じなかった．

プログラムの開始時の処理とハッシュテーブルの更新を含むホワイトリスト更新時の処理時間において MPHF を実装したプログラム 1，3 の方がプログラム 2，4 よりも処理時間が長い点は事前の予想通りである．これは開始時の処理とハッシュテーブルの更新時の処理においてホワイトリストから MPHF を作成する処理が含まれていることが原因と考えられる．

また，実験の結果よりプロセス ID のキャッシュにより一度アクセスを許可されたプロセスについて，今回の検査によるディレイは約 1.4ms 秒未満，CPU 使用率については約 1.5 %，そして物理メモリ使用量は最大でも 7MB 程度と IoT デバイスにおいても十分適用可能なオーバーヘッドであることが確認できた．

7. まとめと今後の課題

本研究では IoT デバイスにおける制限されたリソースのもとで性能の良い攻撃プロセスの検知を行うことを目的として研究を行った．そして，アプローチとしてホワイトリスト方式を採用し，ホワイトリストの探索コストを削減することとした．探索コスト削減のために最小完全ハッシュ関数を利用したハッシュインデックスと一度許可されたプロセス ID のキャッシュにより照合を高速化する方法を提案し，実験による評価を行った．

提案手法について実験を行った結果，プロセス ID のキャッシュについては，プロセス開始時刻をパラメータとしたキャッシュを利用することで通常のアクセス許可処理よりも大幅に高速化し，かつ CPU 使用率を減少させることができると示された．しかし，最小完全ハッシュ関数については従来手法に対しての優位性を確認することができなかった．

今後の課題としては現在実装プログラムに保護対象ディレクトリの指定に依存してプログラムが途中でハングアップすることがあり，実装プログラムが利用するファイルを特定して別途保護する手法を検討する必要がある．また，現在の提案手法では fanotify でアクセス制御を行うことができないイベントについては，inotify を利用して通知を行うのみでありアクセス制御を行うことができない．よって，fanotify で制御を行うことができないイベントに対するアクセス制御手法の研究が必要である．

また，SELinux [16] や Yama [17] といった異なるアプローチから保護を提供するセキュリティモジュールについて本提案手法との比較も行う必要がある．

そして，今回有効性が確認できたプロセス ID のキャッシュについて，IoT 環境に適したキャッシュ置換アルゴリズムの研究を始めとしてより実用的にキャッシュを運用するための研究が必要である．

謝　　辞

本研究の一部は JST 国際科学技術協力基盤整備事業の支援により行われた．

文　　献

[1] AV-TEST. Malware Statistics & Trends Report. http://www.av-test.org/en/statistics/malware/
[2] Adam Sedgewick, Murugiah Souppaya, Karen Scarfone, “Guide to Application Whitelisting,” National Institute of Standards and Technology, NIST Special Publication 800-167, 2015.
[3] J. Powers, R. Smith, Z. Korkmaz and H. Ahmed, “Whitelist malware defense for embedded control system devices,” Saudi Arabia Smart Grid (SASG), Jeddah, pp. 1-6, 2015.
[4] F-Secure LINUX SECURITY. https://www.f-secure.com/ja_JP/web/business_jp/linux-security
[5] Botelho F.C., Pagh R., Ziviani N. “Simple and Space-Efficient Minimal Perfect Hash Functions.” Algorithms and Data Structures. WADS 2007. Lecture Notes in Computer Science, vol 4619. Springer, Berlin, Heidelberg, 2007.
[6] Man page of FANOTIFY. https://linuxjm.osdn.jp/html/LDP_man-pages/man7/fanotify.7.html
[7] Man page of INOTIFY. https://linuxjm.osdn.jp/html/LDP_man-pages/man7/inotify.7.html
[8] S. A. C. DeCato, “Increasing the security on non-networked ground support equipment: Analyzing the implementation of whitelisting protection,” 2016 IEEE AUTOTESTCON, Anaheim, CA, pp. 1-5, 2016.
[9] Hyderabad, India. Himanshu Pareek, Sandeep Romana, Lakshmi Eswari, “Application Whitelisting: Approaches and Challenges.” International Journal of Computer Science, Engineering and Information Technology (IJCSEIT), Vol.2, No.5, 2012.
[10] Mohamed Abomhara and Geir M. Kien, “Cyber Security and the Internet of Things: Vulnerabilities, Threats, Intruders and Attacks,” Journal of Cyber Security and Mobility, 65-88, 2015.
[11] Man page of PROC. https://linuxjm.osdn.jp/html/LDP_man-pages/man5/proc.5.html
[12] CMPH - C Minimal Perfect Hashing Library. http://cmph.sourceforge.net/
[13] Djamal Belazzougui, Fabiano C. Botelho, Martin Dietzfelbinger, “Hash, Displace, and Compress” Computer Research Repository, 2009.
[14] libghthash. http://www.bth.se/people/ska/sim_home/libghthash.html
[15] NIST Policy on Hash Functions. https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions
[16] Security-Enhanced Linux. https://www.nsa.gov/what-we-do/research/selinux/
[17] Yama. https://www.kernel.org/doc/Documentation/security/Yama.txt