Computer Security Symposium 2017
23 - 25 October 2017

Linux 上でのホワイトリスト型実行制御機能 WhiteEgret™の開発

小池 正修[†1]　小椋 直樹[†1]　内匠 真也[†1]　花谷 嘉一[†1]　春木 洋美[†1]

概要：さまざまなセンサーやデバイスをクラウドに接続する IoT 化が進んでいる．IoT における課題の 1 つに，マルウェアへの対策がある．その中の有力な方式として，機器上で実行を許可するプログラムを予めホワイトリストとして定義しておき，ホワイトリストにないプログラムの実行をブロックする実行制御方式が知られている．今回我々は，Linux 上でホワイトリスト型の実行制御機能を実現する WhiteEgret™を開発した．WhiteEgret™はプログラムの実行要求を Linux カーネルでフックし，そのプログラムがホワイトリストに含まれているかチェックすることでプログラムの実行を制御する．本稿では，WhiteEgret™の実装方針および実装結果について述べる．

キーワード：ホワイトリスト，実行制御，Linux，IoT，制御システム

Development of WhiteEgret™: A Whitelisting-type Execution Control on Linux

Masanobu Koike[†1]　Naoki Ogura[†1]　Shinya Takumi[†1]　Yoshikazu Hanatani[†1]
Hiroyoshi Haruki[†1]

Abstract: For embedded devices for IoT system, it is an important issue to prevent malware. Whitelisting-type execution control, which uses a whitelist to decide whether executable components are approved to run on a host or not, is known as one of promising solutions. We develop a whitelisting-type execution control WhiteEgret™ on Linux. In this paper, we describe detail of implementation of WhiteEgret™.

Keywords: Whitelist, Execution control, Linux, IoT, Industrial control system

1. はじめに

近年，さまざまなセンサーやデバイスをクラウドに接続する IoT 化が進んでいる．IoT システムにおける課題の 1 つにセキュリティが挙げられる．例えば，IoT システムの一例である制御システムにおいては，核燃料施設を対象とした Stuxnet や，制御用プロトコルを対象とした DragonFly など，多くのサイバー攻撃の事例が報告されている．

IoT システムでは，Intel®[a]，ARM®[b]などのさまざまな CPU と，Windows®[c]/Linux®[d]といった汎用 OS，TCP/IP に代表される汎用プロトコルが採用されているが，計算リソースが限られること，更新が容易でない環境で利用されるなど，一般的な情報システムとは異なる対策が求められる．

このような状況の中で，IoT システムに対する情報セキュリティ対策の 1 つとして，ホワイトリスト型の実行制御技術が考えられている[1][2]．これは機器上で実行を許可するプログラムを予めホワイトリストとして定義しておき，ホワイトリストに存在しないプログラムの実行をブロックする技術である．

表 1　ホワイトリスト型との相性がよい理由

IoT システムの特徴	相性がよい理由
10 年以上の長期に渡って利用される可能性がある	未知（未来）のマルウェアに対しても実行制御ができる
それぞれの機器で動かす機能は多くない	機器の動作に必要なプログラムのリストアップが可能
機能が変更される頻度は多くない	ホワイトリストの更新の頻度が少ないため，保守の手間が少ない
外部ネットワークと接続される環境が貧弱である場合がある	外部からのウィルス定義ファイルのダウンロードは不要

一般に，IoT システムは

- 10 年以上の長期に渡って利用される可能性がある
- それぞれの機器で動かす機能は多くない
- 機能が変更される頻度は多くない
- 外部ネットワークと接続される環境が貧弱である場合がある

という特徴を持っている．これらの特徴から，表 1 にまと

†1 株式会社東芝　研究開発センター
Corporate Research & Development Center, Toshiba Corporation

a) Intel，Xeon は，アメリカ合衆国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です．

b) ARM，Cortex は，ARM Limited（またはその子会社）の EU およびその他の国における登録商標もしくは商標です．

c) Windows は，米国 Microsoft Corporation の米国およびその他の国における登録商標です．

d) Linux は，Linus Torvalds 氏の米国，日本およびその他の国における登録商標または商標です．

その他本稿に掲載の商品，機能等の名称は，それぞれ各社が商標として使用している場合があります．

めた理由により，ウィルス対策ソフトで採用されているブラックリスト型と比較し，IoT システムはホワイトリスト型と相性がよいといえる．

ホワイトリスト型の実行制御を行う製品は，セキュリティ企業からいくつか販売されている[3][4]．これらの製品は，Intel CPU を対象としたものがほとんどで，動作する CPU が限定されているという課題がある．

また，実行制御機能を試作する研究も行われている．文献[1]では，システムごとの仕様要求に応じて機能を作りこめるようにするために，Linux カーネル内にホワイトリスト型の実行制御機能を実装している．

2017 年 8 月現在，Linux カーネルには，強制アクセス制御を実現する SELinux[5]などさまざまなセキュリティ機能が標準搭載されているが，筆者らが知る限り，ホワイトリスト型の実行制御機能を直接提供するものは存在しない．

今回我々は，IoT システムでも多く採用されている Linux 上で動作するホワイトリスト型の実行制御機能 WhiteEgret™を開発した．その特長として，利用環境に依存しない部分を Linux カーネルに実装したことが挙げられる．利用環境に依存する部分はユーザ空間で対応する方針をとっているため，その環境に適した実行制御をユーザ側が柔軟に制御できる．また WhiteEgret™は Linux の標準機能だけを用いて実装しているため，Intel CPU でも ARM CPU でも動作し，幅広い組み込み機器に活用されることが期待される．さらにそれを後押しするために，WhiteEgret™が Linux カーネルに標準搭載されること（メインライン化）を目指している[6]．

本稿では，WhiteEgret™の実装について述べる．2 章では，IoT システムでの利用を想定したときの要件を整理する．それを受けて 3 章では，WhiteEgret™の実装方針を，4 章では実装の詳細を述べる．WhiteEgret™がさまざまな環境で動作することの確認，および処理速度のオーバーヘッドについて，5 章に記載する．6 章で，実装結果から WhiteEgret™が 2 章の要件を満たしていることを確認し，7 章で本稿をむすぶ．

2. 想定要件

本章では，IoT システムでホワイトリスト型の実行制御機能が使われる状況を想定し，WhiteEgret™に求められる要件を整理する．

(1) プラットフォーム

組み込み機器はさまざまな環境で利用されることから，特定のプラットフォームに依存しないことが望まれる．具体的には，CPU および Linux のディストリビューションに依存しない実装とする．特に，情報機器では Intel CPU が使われることが多いが，組み込み機器では ARM CPU が採用されていることも多いため，Intel CPU だけでなく ARM CPU でも動作することを要件とする．

(2) 実行制御の対象

未知のマルウェアの動作をブロックするため，すべての実行ファイルを実行制御の対象とする．ここで実行ファイルとは，プログラム及びプログラムから動的にリンクされることで動作する共有ライブラリを指すものとする．

(3) 実行制御の柔軟性

組み込み機器はさまざまな環境で利用されることから，その環境に適した実行制御を柔軟に実現できることが望ましい．例えば，環境に適したホワイトリストの構成要素を選択できたり，通常運用時と保守時でホワイトリストを切り替えたりできるとよい．後者の例としては文献[2]があげられる．文献[2]では，システムのライフサイクルのフェーズごとに，それぞれ適したホワイトリストを利用することを提案している．

(4) オーバーヘッド

実行制御機能を導入することで，実行ファイルの動作が遅くなることが予想される．IoT システムはリアルタイム性を求められる状況も多いため，実行制御機能によるオーバーヘッドをなるべく小さくする必要がある．

(5) ユーザビリティ

IoT システムの運用者は情報システムに詳しくない場合もあるため，簡単に使えることが望ましい．ホワイトリスト型の実行制御機能を実現するために，複雑な設定ファイルやポリシーファイルが必要であったり，複雑な手順が必要であったりすると，現場で使われないことになりかねない．

3. 実装方針

本章では，前章の想定要件をもとにした実装方針を述べる．

ホワイトリスト型の実行制御機能は，図 1 のステップで実現される．

1	実行ファイルが実行されようとしていることを検知する．
2	その実行ファイルがホワイトリストに存在するかどうか確認する．
3	ホワイトリストに存在する場合は実行処理を継続し，存在しない場合は実行処理を打ち切る．

図 1　実行制御のステップ

(1) **ステップ 1：実行ファイルの実行の検知**

実行ファイルの実行の検知は，ユーザ空間上のプログラムで行う方法，カーネル空間で行う方法がある．仮想化してハイパーバイザの機能で行う方法もあるが，組み込み機器では仮想化環境を前提とするのは現実的とは言えないため，この方法は除外した．

ユーザ空間上のプログラムで行う方法では，ptrace などを用いてシステムコールを捕捉することが考えられるが，オーバーヘッドが大きく，あらゆる実行ファイルの実行を効率良く検知するのは難しい．

一方のカーネル空間では，すでに実行ファイルの実行を効率よく検知してフックする機構が備わっている．以上のことから，実行ファイルの実行の検知は，カーネル空間で行う方法を採用した．

(2) **ステップ 2：ホワイトリストに存在するかの確認**

指定した実行ファイルがホワイトリストに存在するかどうかの確認は，ユーザ空間上の制御プログラムで行う方法とカーネル空間で行う方法がある．

ユーザ空間上の制御プログラムで行う場合は，制御プログラムおよびホワイトリストの作り方に自由度を与えられる．これは実行制御をユーザが柔軟に実現できるメリットとなる．その反面，実行制御機能のセキュリティがその制御プログラムの信頼性に依存するデメリットがある．

一方，カーネル空間で行う場合は，ユーザ空間の信頼性に依存しないというメリットがある．また，処理がカーネル空間で閉じていてユーザ空間との通信が発生しないため，オーバーヘッドが抑えられる点も有利である．しかしながら，実行の制御はどの環境でもカーネルに実装されている通りの挙動しかできないため，環境に応じて柔軟に制御できないというデメリットがある．

どちらの方法でも一長一短があるが，今回我々は，IoT システムの多様性に鑑み，実行制御の柔軟性を重視した．すなわち，指定した実行ファイルがホワイトリストに存在するかどうかの確認を，ユーザ空間上の制御プログラムで行う方法を採用した．

(3) **ステップ 3：実行ファイルの実行可否の判断**

実行ファイルの実行可否の判断処理は，ステップ 1 までの処理を行ったカーネル空間が，ステップ 2 の結果に基づいて行うのが自然な方法であるので，その方法を採用した．

以上の議論をまとめ，実行制御機能の実装の概要をユーザ空間とカーネル空間に分けて図示したものが図 2 である．IoT システムの環境に依存する部分はユーザ空間で，依存しない共通部分はカーネル空間で処理する分担となっている．

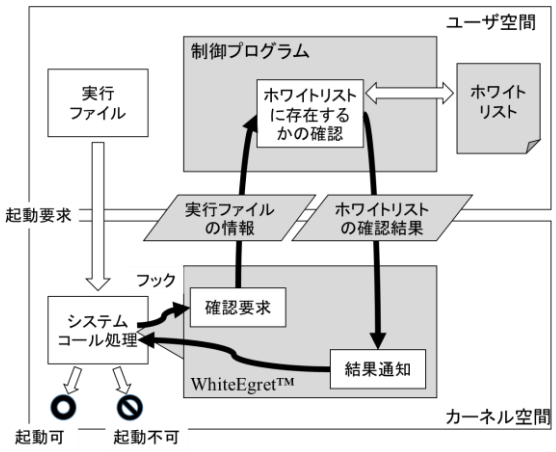

図 2　実行制御機能の実装の概要

4. 実装

本章では，WhiteEgret™の実装の詳細を述べる．

4.1 ホワイトリスト

ホワイトリストはユーザ空間側で利用する情報であり，また実行制御を柔軟に行うために，WhiteEgret™ではホワイトリストの内容やフォーマットは規定しない．しかしながら，ホワイトリスト型の実行制御機能を実装する上で，ホワイトリストの内容についてある程度前提をおく必要があるため，ここでその前提について記載する．

ホワイトリストは，実行を許可する実行ファイルのリストである．ここに含まれると判断された実行ファイルは実行が許可されるため，実行してよいか問い合わせた実行ファイルがリストにある実行ファイルと本当に同じものか，正しく判断する必要がある．そのためには，実行ファイルを一意的に識別でき，かつその実行ファイルがホワイトリストに登録されたものから変化していないことを確認しなければならない．そのために利用できる属性の候補は，例えば，文献[7]にまとめられている．

後者の確認については，実行ファイルのハッシュ値またはデジタル署名を利用する方法が考えられるが，オーバーヘッドを考慮して，ハッシュ値を用いることとする．すなわち，その実行ファイルのハッシュ値をホワイトリストに登録しておき，実行制御時にもう一度ハッシュ値を計算してホワイトリストのハッシュ値と同じかどうか確認するものとする．

一方前者の識別には，実行ファイルのパス名（絶対パス）を使うことが考えられる．ただし，パス名を使う場合は，一意性に厳密さを欠く場合もあることに注意が必要である．例えば，ハードリンクを作成した場合，同じ実行ファイルに対して複数のパス名が存在することになる．この場合，それらすべてをホワイトリストに登録するか，すべてを登録しないかのどちらかでないと，呼び出したパス名に依存

して実行制御機能が矛盾した動きをする可能性がある．また，mount や chroot コマンドを使うと，パス名が変更されるため，ブロックすべき実行ファイルの実行が許可されたり，許可すべき実行ファイルの実行がブロックされたりすることも起こりうる．

上記のことが起こりうる状況の場合は，より厳密に一意性を保証できる属性を使用する必要がある．Linux カーネルの内部では，ファイルシステム内のオブジェクトは i ノード番号とデバイス番号で管理されている．i ノード番号はそのファイルシステム内での固有の番号で，異なるオブジェクトには異なる番号が付与される．ハードリンクで異なるパス名が与えられたとしても，それらは同じ i ノード番号を持つため，パス名に関わらずオブジェクトの実体に対して付番できる特徴がある．また，mount や chroot コマンドに対してもファイルシステムが変わらなければ i ノード番号は不変である．

ただし，異なるファイルシステム上では，異なるオブジェクトに対して同じ i ノード番号が付与されることはありうる．そのファイルシステムが存在するデバイスの固有の番号であるデバイス番号と組にすることで，対象のオブジェクトを一意的に識別可能となる．

上で述べたように，実行ファイルを一意的に識別するためには，（i ノード番号，デバイス番号）の組が必要十分である．しかしながら，ホワイトリストの移植性および視認性という観点では，パス名のほうが優れている場合もある．

例えば，新規に多数の同じ機器を導入する状況を考える．すべての機器のファイル構成が同じだとすると，設定すべきホワイトリストもすべて同じである．ホワイトリストを（i ノード番号，デバイス番号）の組で構成していた場合，異なる機器上では，同じパス名であっても i ノード番号は異なる値となるため，機器ごとにホワイトリストを設定しなければならない．これは大きな導入コストとなる．一方，ホワイトリストをパス名で構成していた場合，すべての機器で同一となるため，ホワイトリストをコピーするだけでよい．

以上の考察から，本稿ではホワイトリストは，実行を許可する実行ファイルの（i ノード番号，デバイス番号，ハッシュ値）をリスト化したもの，または（パス名，ハッシュ値）をリスト化したものと仮定する．

4.2 LSM

カーネル空間で実行ファイルの実行を検知する方法として，LSM (Linux Security Modules) を利用した．LSM は Linux カーネル 2.6 以降で標準搭載されているフレームワークである．LSM は強制アクセス制御を実現する仕組みとして導入されたもので，ユーザ空間側からの内部の重要なカーネルオブジェクトへのアクセスに対し，カーネル内のすべての呼び出し箇所にフックポイントを定義する．事前にコールバック関数を関連付けておくことで，そのフックポイント到達時に関連付けたコールバック関数を呼び出し，その呼び出し処理を継続してよいか判断させることができる．

したがって，ユーザ空間からの実行ファイルの実行を要求するシステムコールをフックするポイントに，その実行ファイルがホワイトリストに存在するかどうか確認する機能を持つコールバック関数を関連付けておけば，ホワイトリストに存在するかどうかで，システムコールの処理を継続するか打ち切るかの実行制御ができる．

LSM を利用する際には，必要なフックポイントにのみコールバック関数を実装すればよい．したがって，カーネルへの実装は最小限に抑えられる．実装していないフックポイントについては，呼び出し処理が継続され，カーネルオブジェクトへのアクセスが許可される．

4.3 フックポイント

WhiteEgret™ で用いた LSM のフックポイントは，bprm_check_security および mmap_file の 2 つである．

bprm_check_security は，プログラムの実行処理を行う execve システムコール内でフックされるポイントである．したがって，すべてのプログラムの実行を検知することができる．

その一方で，Linux でのプログラムの実行時には，共有ライブラリがロードされて実行されることも多い．共有ライブラリのロード時には execve システムコールは発行されないため，bprm_check_security のフックにかからない．そこで WhiteEgret™では，共有ライブラリのロードをフックするために mmap_file フックポイントを利用した．これはロード時に行われる，ファイルをメモリにマッピングする mmap システムコール内でフックされるポイントである．ただし，このフックポイントは共有ライブラリのメモリマッピングだけでなく，その他のファイルのメモリマッピングの際にもフックされる．共有ライブラリのメモリマッピングの場合のみを処理の対象とするために，そのメモリが実行可能であり，かつ execve システムコール経由の呼び出しでない場合に限り，コールバック関数を呼ぶようにした．

4.4 通信データ

カーネル空間でフックした後のコールバック関数内で，ユーザ空間に対してホワイトリストにその実行ファイルが存在するかの問い合わせを行う．その際にカーネル空間からユーザ空間に渡す情報は，その実行ファイルの i ノード番号，デバイス番号，パス名の 3 つとした．この 3 つの情報があれば，4.1 節で仮定したどちらのタイプのホワイトリストにも対応できる．

逆向きの，ユーザ空間からサーバ空間への通信内容は，問い合わせのあった実行ファイルがホワイトリストに含ま

れていたかどうかの情報である．含まれていたという情報を受けたカーネル空間は，フックポイント以降の処理を継続して対象の実行ファイルを実行する．含まれていなかったという情報を受けた場合は，コールバック関数はエラーを返し，そのフックポイントでシステムコール処理を中断することで，ホワイトリストにない実行ファイルの実行をブロックする．

4.5 カーネル空間／ユーザ空間の通信

カーネル空間とユーザ空間の間の通信には，securityfs を利用した. securityfs は RAM ベースのファイルシステムで，ユーザ空間からカーネル空間の LSM のメモリへアクセスするためのインタフェースを提供できる．securityfs は通常 /sys/kernel/security にマウントされる．WhiteEgret™ではここに準備したデバイスファイルにユーザ空間側の制御プログラムから読み書きすることで，ユーザ空間側への通信，ユーザ空間側からの通信を実現している．

5. 実装結果

本章では，2 章で述べた想定要件のうち，プラットフォームとオーバーヘッドについての実装結果を記す．

5.1 プラットフォーム

WhiteEgret™が，CPU および Linux のディストリビューションに依存していないことを確認するために，表 2，表 3，表 4 に示すさまざまな環境で動作させた．

表 2 は，CPU が Intel，OS が CentOS であり，VMM として VMware ESXI を用いた仮想環境である．表 3 は，CPU が Intel，OS が Ubuntu であり，実機（MAGNIA 3600）上の動作環境である．表 4 は，CPU が ARM，OS が Linux raspberrypi であり，ハードウェアは Raspberry Pi 3 Model B である．

いずれの環境においても，正常に動作することを確認した．ここで，正常に動作するとは，ホワイトリストに含まれるプログラム，および共有ライブラリは実行でき，含まれないものは実行がブロックされることを指す．

表 2　測定環境（Intel/CentOS）

項目	内容
CPU	Intel Xeon E5-2690 v3 @ 2.60GHz
メモリ	2GB
OS	CentOS 7.2 (kernel 3.10.0)
VMM	VMware ESXI 6.0

表 3　測定環境（Intel/Ubuntu）

項目	内容
CPU	Intel Xeon CPU E5520 @ 2.27GHz
メモリ	15GB
OS	Ubuntu 16.10 (kernel 4.8.0)
HW	MAGNIA 3600

表 4　測定環境（ARM/Raspbian）

項目	内容
CPU	ARM Cortex-A53 @ 1.2GHz
メモリ	1GB
OS	Linux raspberrypi (kernel 4.9.45)
HW	Raspberry Pi 3 Model B

5.2 オーバーヘッド

ホワイトリスト型の実行制御機能を導入による，処理のオーバーヘッドを測定した．測定環境は表 2，表 3，表 4 のそれぞれの環境である．

ホワイトリストは，測定環境内のすべての実行ファイルの（パス名，ハッシュ値）の組をリスト化したものを用いた．ハッシュ関数は SHA-256 とした．

ユーザ空間側の制御プログラムは，デーモンとして作成した．その動きは以下のとおりである．

- securityfs のデバイスファイルで，カーネル空間からの問い合わせを待ち受ける．
- 問い合わせがあった場合は，受け取った実行ファイルの情報をもとに，それがホワイトリストに存在するか確認する．
- その結果を securityfs のデバイスファイルに書き込み，カーネル空間に通知する．

実行制御対象の実行ファイルとして，何も処理せずにリターンするプログラムを作成した．ライブラリは静的リンクし, 実行時に共有ライブラリをロードしないようにした．したがって，この実行ファイルを実行した際に発生するフックとそれに伴うホワイトリストの確認は 1 回だけとなる．この実行ファイルを 100 回実行したときの平均値として処理時間を測定した．

ホワイトリスト型の実行制御機能の導入による処理のオーバーヘッドを計るため，ユーザ空間の制御プログラムを起動する前，および起動した後の処理速度をそれぞれ測定した．また，制御プログラム起動後では，対象の実行ファイルがホワイトリストに含まれる場合と含まれない場合の 2 通りを測定した．

以上の環境で測定した結果を図 3 に示す. いずれの環境でも, WhiteEgret™を使うことで 4〜16m 秒程度のオーバーヘッドが発生した．

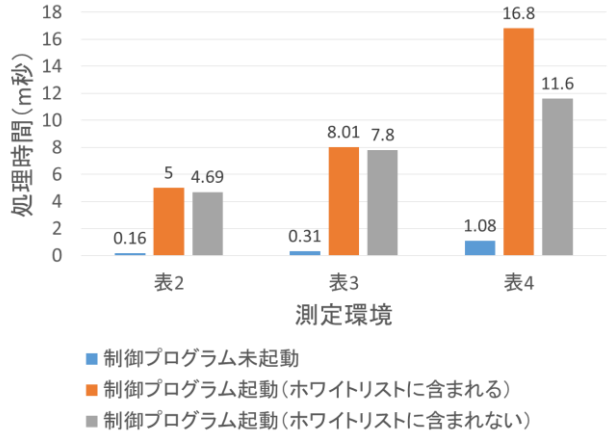

図 3　処理時間

6. 考察

本章では，実装した WhiteEgret™について，2 章で述べた 5 つの要件に照らして考察する．

(1) プラットフォーム

5.1 節で見たように，WhiteEgret™は Intel CPU（表 2, 表 3）だけでなく ARM CPU（表 4）でも動作することが確認できた．また，表 2，表 3 はそれぞれ Red Hat 系，Debian 系ディストリビューションであり，異なる系列の代表的なディストリビューションでも動作を確認できたことから，Linux のディストリビューションにも依存しないといえる．すなわち，WhiteEgret™は CPU や Linux のディストリビューションによらないことが確かめられた．さらに，仮想化環境（表 2），実機環境（表 3），Raspberry Pi 環境（表 4）でも問題なく動作することが確認できた．したがって，WhiteEgret™は特定のプラットフォームに依存せず，幅広い組み込み機器で利用可能である．

(2) 実行制御の対象

WhiteEgret™の動作確認において，ホワイトリストに含まれるプログラム，および共有ライブラリは，実行できることを，含まれないものは実行をブロックされることを確認した．したがって，プログラムだけでなく，それがロードする共有ライブラリの実行制御も可能である．

(3) 実行制御の柔軟性

WhiteEgret™ではホワイトリスト自体，およびホワイトリストに存在するか確認する制御プログラムの内容には依存しておらず，利用環境に応じて自由に作成することができる．5.2 節では（パス名，ハッシュ値）のホワイトリストの場合を扱ったが，（i ノード番号，デバイス番号，ハッシュ値）のホワイトリストでも同様に動作することを確認している．今回動作確認をしていないが，原理的にはハッシュ値の代わりにデジタル署名としてもよいし，また，制御プログラムに現在のライフサイクルを推定する機能を持たせれば，文献[2] で提案されている，システムのライフサイクルのフェーズごとに，それぞれ適したホワイトリストに切り替えることも可能である．

(4) オーバーヘッド

5.2 節の実験環境では，WhiteEgret™を使うことで 4〜16m 秒程度のオーバーヘッドが生じることを確認した．このオーバーヘッドの大部分は，対象の実行ファイルのハッシュ値計算時間である．実際，例えば表 2 の環境では，対象の実行ファイルのサイズは 825K バイトであり，ハッシュ値計算に約 4.2m 秒かかっている．この処理はユーザ空間側の制御プログラムの処理時間であるため，Linux カーネル部分の WhiteEgret™自体のオーバーヘッドは 0.6m 秒程度であり，十分小さいと考えられる．

ハッシュ関数の処理時間は，対象のファイルサイズにほぼ比例するため，実行ファイルのサイズに応じてオーバーヘッドが増減する．巨大な実行ファイルを含む場合は，IoT システムを本稼動する前にオーバーヘッドの影響を検討するなどの対応をとることが望ましい．

(5) ユーザビリティ

WhiteEgret™を利用するために，ユーザ空間側ではホワイトリストおよび制御プログラムを準備する必要がある．これは一般的に，IoT 運用者に対して複雑な設定ファイルやポリシーファイル，あるいは複雑な手順を要求することはないと考えられる．例えば，機器の初期状態がクリーン，すなわち望ましくないファイルが含まれていない，と仮定すると，ホワイトリストは機器内のすべての実行ファイルを機械的にリスト化することで作成できる．また，制御プログラムは一旦開発すれば，運用者はそれを起動するだけで実行制御機能を有効にできる．

以上で考察したように，WhiteEgret™は 2 章の 5 つの想定要件に応える実装となっているといえる．特にプラットフォームに依存せずに動作することは，IoT システムの幅広い機器への適用が期待できる．

7. むすび

本稿では，Linux 上でホワイトリスト型の実行制御機能を実現する WhiteEgret™の実装方針と実装，および実装結果について報告した．

WhiteEgret™は LSM を利用して Linux カーネルに実装している．CPU および Linux ディストリビューションには依存していないため，さまざまな機器へ搭載することが可能である．

また，WhiteEgret™の別の特長として，ホワイトリストを確認する制御を，ユーザ空間で行うことが挙げられる．この特長により，IoT システムを利用する環境に応じて柔軟に実行制御することが可能となる．

我々は WhiteEgret™が Linux カーネルの標準機能として搭載されることを目指して，メインライン化の活動を行っている．多くの IoT 環境で簡単に使えるホワイトリスト型の実行制御機能となるように，活動を続けていく．

参考文献

[1] 伊藤孝之，菅井尚人，藤田淳文，鶴薫．Linux におけるプログラムホワイトリスト化試作．情報処理学会第 77 回全国大会，4E-02，2015.

[2] 秦康祐，佐々木翼，澤田賢治，中井綱人，山口晃由，小林信博．プロセス制御システムのライフサイクルに基づいたホワイトリスト．SCIS 2017，1C2-1，2017.

[3] “McAfee Application Control”．https://www.mcafee.com/jp/products/application-control.aspx, (参照 2017-08-21).

[4] “Trend Micro Safe Lock”．http://www.trendmicro.co.jp/jp/business/products/tmsl/, (参照 2017-08-21).

[5] “SELinux Project”. https://github.com/SELinuxProject, (参照 2017-08-21).

[6] “WhiteEgret LSM module”．https://lkml.org/lkml/2017/5/30/376, (参照 2017-08-21).

[7] Sedgewick, A, Souppaya, M, Scarfone, K. Guide to Application Whitelisting. NIST SP800-167, 2015.