情報処理学会論文誌 Vol.62 No.12 1893–1903 (Dec. 2021)

Windowsにおけるスレッド挿入の起点ファイル情報伝播による長期証拠保全

田中 大樹[1,†1,a)] 川古谷 裕平[2] 岩村 誠[2] 鄭 俊俊[1] 毛利 公一[1,b)]

受付日 2021年3月8日, 採録日 2021年9月9日

概要：マルウェアによる被害状況を解明する手法としてフォレンジックスの重要性は高い．しかし，マルウェアには解析を妨害する機能を持つものが存在するため，解析が難化，高コスト化している．特に，マルウェアを隠蔽することで解析を妨害するスレッド挿入機能は，良性プロセスに悪性挙動を行わせたり，スレッド挿入を繰り返すことで，起点となったマルウェアを特定する大きな障害となる．そこで，本論文では，Windows において，スレッド挿入の起点となったマルウェアを特定可能とする長期証拠保全手法を提案する．具体的には，実行ファイルのハッシュ値を Windows カーネル内のデータ構造に格納し，スレッド挿入が行われると，挿入先にハッシュ値を伝播させるものである．これによって，悪性の挙動がスレッド挿入によるものであった場合，すでにマルウェアの実行ファイルが存在しなくても，フォレンジックス実施時に起点となったマルウェアのハッシュ値を証拠として取得することができる．また，カーネル内にハッシュ値を保持するため，潜伏期間が長期にわたる場合にもハッシュ値を安全に保持できる．フォレンジックス実施者はハッシュ値からマルウェアを検索・取得できる．マルウェアは，その形式として exe 形式や DLL 形式のものがあるため，両者ともに追跡可能とした．評価では，マルウェアに見立てたスレッド挿入を行う exe ファイルと DLL ファイルを動作させ，その実行ファイルを特定できることを確認した．また，本手法を起動時から適用させることを想定し，Windows の起動・シャットダウン時のオーバヘッドの測定とアプリケーション起動におけるオーバヘッドの計測を行った．

キーワード：スレッド挿入，マルウェア追跡，オペレーティングシステム

Long-term Evidence Preservation by Propagating File Information from Starting Point of Thread Injection in Windows

Daiki Tanaka[1,†1,a)] Yuhei Kawakoya[2] Makoto Iwamura[2] Junjun Zheng[1] Koichi Mouri[1,b)]

Received: March 8, 2021, Accepted: September 9, 2021

Abstract: The importance of forensics is high as a method to clarify the damage caused by malware. However, many malware has functions that interfere with analysis, making analysis more complicated and high cost. In particular, the thread injection function, which interferes with analysis by hiding malware, is a significant obstacle to identifying the originating malware by causing benign processes to behave maliciously or by repeatedly inserting threads. This paper proposes a long-term evidence preservation method to identify the originating malware that used thread injection in Windows. Specifically, we store the hash values of executable files to a data structure in the Windows kernel, and when a thread is injected, the hash values are propagated to the injected process or thread. If the malicious behavior is due to thread injection, the hash value of the malware can be obtained as evidence during forensics even if the malware executable does not exist. In addition, since the kernel keeps the hash values, they can be safely kept even when the infected period is long. In addition, since the hash values are stored in the kernel, the hash values are not lost even if the infected period is long. The forensic investigator can search for and retrieve malware from the hash value. Since malware can be in the format of exe or DLL, the proposed method assumes that both types of malware. In the evaluation, we confirmed that we could identify the executable files by running an exe file and a DLL file that injected threads in malware. We also measured the overhead of Windows startup and shutdown, and the overhead of application startup, assuming that our method is applied from startup.

Keywords: thread injection, malware trace, operating system

1. はじめに

マルウェアを利用したサイバー攻撃による被害は年々増加し [1]，企業や政府機関に対する標的型攻撃が確認されている [2], [3]．こうした被害が発生した場合，ログやメモリ，記憶媒体等から関連する情報を抽出して分析をするフォレンジックスが実施される．そして，ここで得られた情報から，どのようなマルウェアが用いられたのか，どのような攻撃が行われたのか，どのような被害があったのか等の原因や被害状況の調査が行われたり，今後のマルウェア対策も行われる．このとき，マルウェア本体のファイルにつながる確実な情報を取得することができれば，調査や対策もよりスムーズに，より確実に実施することができる．しかし，実際には，マルウェア本体のファイルにつながる確実な情報を得ることは難しい．

たとえばRAT（Remote Administration Tool）と呼ばれる遠隔操作マルウェア [3] に代表されるような，継続的な情報の窃取や踏み台確保を目的とするマルウェアが存在する．これらのマルウェアは長期間にわたって潜伏と悪性挙動を行うが，攻撃者にとってはマルウェアの検出や解析，マルウェアに対する対策は活動の妨げとなるため，マルウェアに解析を妨害する機能を組み込むことで解析を困難にしている．特にマルウェアの正確な解析をするうえで課題となるのがスレッド挿入機能である．スレッド挿入機能は，他の動作中のプロセスにプログラムコードを挿入し，注入されたプロセスに属するスレッドとして当該プログラムを実行させる機能を指す．通常のアプリケーションがマルウェアによって悪性コードを挿入されると，悪性挙動であっても通常のアプリケーションの処理に見えるため，被害の発見の遅れ，正常なファイルのマルウェアとの誤認等，マルウェアの特定を困難にすることにつながる．また，長期潜伏を許してしまった場合には，ログファイルのローテーション等によって古い記録が消去されることがあり，同様にマルウェアの特定を困難にする．以上から，マルウェアがスレッド挿入機能を有していたり，長期潜伏しているような場合においても，マルウェアにつながる正確な情報を取得可能とするよう長期の証拠保全が求められる．

マルウェアにつながる情報を取得するものとして，すべてのスレッド生成や他プロセスのメモリ空間へのアクセスを監視し，そのログを取得する方式が提案されている [4], [5]．この方式では，マルウェアが書き込んだメモリ領域を1バイト単位で追跡するテイント解析を行い，より詳細なマルウェアの挙動を観測可能としている．しかし，テイント解析にはメモリアクセスをつねに監視する必要があり，かつ一般的に利用される環境では監視対象が多くなるため，観測時のオーバヘッドが問題となる．また，情報を保存するためのログファイルは，保存期間やログファイルサイズに上限が設けられていたり，上限を超えた場合にはローテーション等によって古い記録が削除されるため，マルウェアの潜伏期間・悪性挙動の期間によってはフォレンジックス実施時にログが残っていない可能性があり，長期の証拠保全には向いていない．

以上から，本論文では，広く利用されているWindowsにおいて，スレッド挿入機能を有するマルウェアに対し，フォレンジックス実施時に有効な情報の取得を可能とする長期証拠保全手法を提案する．具体的には，スレッド挿入をしたスレッドの実行ファイルのハッシュ値を記録し，さらにスレッド挿入やスレッド生成を繰り返した場合にはそのハッシュ値を伝播させる機能を実現する．これによって，スレッド挿入の起点となったプロセスの実行ファイル（以下，起点ファイルと記す）のハッシュ値を容易に取得可能とした．提案手法の貢献は次にあげるとおりである．

- 自己削除・自己改変（以下，自己隠蔽と記す）機能を有しているようなマルウェアであってもVirusTotal [6] 等を用いてマルウェアを検索・取得が可能となるよう，起点ファイルのハッシュ値を証拠として取得可能とした．
- マルウェアの潜伏期間が長期にわたる場合にも適したハッシュ値の保全方式を提案した．
- マルウェアの実体がexe形式であっても，DLL形式であっても追跡可能とした．
- DLLファイルはexeファイルと比較してロードが頻発するため，DLL形式に対する処理についてはオーバヘッド軽減策も含めて提案をし，プロセスが多数動作するような一般的な環境でも利用可能とした．

以下，本論文では，2章でスレッド挿入攻撃と既存の特定手法の課題について述べ，3章で提案手法について，4章でその実装について述べる．また，5章で評価について，6章で提案手法の制限について，7章で関連研究について述べる．最後に8章でまとめとする．

2. スレッド挿入の特定手法における課題

2.1 スレッド挿入攻撃

スレッド挿入攻撃は，マルウェア等悪性プロセスが悪性コードを他の良性プロセスへ挿入し，スレッドの形で実行させる手法である．マルウェアは，良性プロセス内の悪性コードによって，自身のプロセスを終了させたりファイルを削除する等の自己隠蔽機能を有することが多い．自己隠蔽機能により，フォレンジックス実施時にはファイルやメ

[1] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[2] NTT 社会情報研究所
NTT Social Informatics Laboratories, Musashino, Tokyo 180–8585, Japan
[†1] 現在，日本電気株式会社
Presently with NEC Corporation
[a)] dtanaka@asl.cs.ritsumei.ac.jp
[b)] mouri@cs.ritsumei.ac.jp

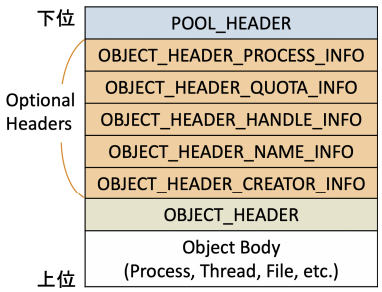


図 1 Windows におけるオブジェクトのデータ構造
Fig. 1 Data structure of object in Windows.

モリ内にマルウェアにつながる情報が残らず，フォレンジックスを困難にする．

2.2 既存のマルウェア特定手法の課題

2.2.1 Windows におけるメモリダンプの解析

フォレンジックスにおける侵害原因の調査段階では，調査対象のコンピュータのメモリダンプを取得し，そのなかの情報から原因を明らかにする手法がとられる．スレッド挿入攻撃を行う悪性プロセスによって生成されたスレッドの特徴は次のとおりであり，両方を満たせば悪性プロセスによるものであると判断できる．

- スレッドを生成したプロセスと，そのスレッドが属するプロセスとが異なる（プロセス内でのスレッド生成ではない）．
- スレッドを生成したプロセスと，そのスレッドが属するプロセスの親プロセスとが異なる（プロセス生成ではない）．

Windows では，スレッド，プロセス，ファイル等あらゆるものをオブジェクトと呼ばれるデータ構造で表現しており，オブジェクトがそれらの実体であるといえる．オブジェクトには，カーネルオブジェクト，ユーザオブジェクト，GUI オブジェクトの 3 種類があり，スレッドやプロセスはカーネルオブジェクトとしてメモリ上で管理される．カーネルオブジェクトのデータ構造を**図 1** に示す [7]．オブジェクトの本体は図 1 の Object Body に格納され，オブジェクトを管理するための情報がその他のヘッダ部分に格納される．

Windows にはカーネルの細かな動作設定をするための Global Flags と呼ばれるフラグがあり，そのうち FLG_MAINTAIN_OBJECT_TYPELIST を有効にすると，オブジェクトの Optional Headers の 1 つである OBJECT_HEADER_CREATOR_INFO 構造体にそのオブジェクトを作成したプロセスの PID（プロセス ID）を格納させることができる．すなわち，スレッド挿入をしたプロセスの PID を取得することができる．しかし，マルウェアのスレッド挿入の目的は自己隠蔽であるため，悪性プロセスは早期に終了してしまったり，実行ファイルが削除されてしまうことが多い．したがって，スレッド挿入したプロセスの PID が得られても，フォレンジックス実施時にマルウェアの有効な情報となりうることは期待できない．また，PID は再利用されるため，他の良性プロセスを悪性プロセスであると誤認する可能性もある．

なお，以下では，スレッドを管理するカーネルオブジェクトをスレッドオブジェクト，プロセスを管理するカーネルオブジェクトをプロセスオブジェクトと呼ぶ．

2.2.2 Sysmon のログの解析

スレッド挿入攻撃に利用される Windows API である CreateRemoteThread を検知するソフトウェアに，Microsoft 社が提供する Sysmon [8] がある．Sysmon は，様々なシステムアクティビティを取得してイベントログに出力する機能を有している．しかし，ログファイルとして出力されるため，その保存期間等が問題となる．文献 [9] によると，サイバー攻撃が発生してから検知されるまでの期間は 2,000 日を超える事例があり，このような長期潜伏が発生するとその間にログが失われてしまう可能性が高い．また，企業等において多数の計算機のログ保管が必要となるような場合は，すべての情報を長期にわたって保管し続けることが難しい場合もある．以上より，ログの保存期間に依存しない証拠の保全手法が求められる．

3. 提案手法

3.1 全体像

2 章で述べた問題を解決すべく，Windows 上で動作するスレッド挿入機能を有するマルウェアに対し，フォレンジックス実施時に有効な情報として起点ファイルであるマルウェアの実行ファイルのハッシュ値を取得可能とする手法を提案する．また，マルウェアの実行ファイルの形式が exe 形式であっても DLL 形式であってもハッシュ値を取得可能とする．これによって，自己隠蔽機能を有しているようなマルウェアであっても VirusTotal 等を用いてマルウェアを検索・取得が可能となる．提案手法の概要を**図 2** に示す．図 2 は，起点ファイルの exe 形式が実行または DLL 形式がロードされた後に，悪性プロセス（図 2 左側）が発生している例である．さらに，悪性プロセスから良性プロセス（図 2 中央）へスレッド挿入攻撃がなされ，さらに別の良性プロセス（図 2 右側）へ多段階のスレッド挿入攻撃がなされている．

提案手法は，起点ファイルが exe であった場合（図 2 exe 手法）を想定して，プロセス生成時に起点ファイルのハッシュ値を記録する．これによって，マルウェアが自己隠蔽をしても実行開始時のハッシュ値を保持することができる．その後，スレッドが生成されるたびにその値を伝播させる処理を行う．

DLL は，プロセスにロードされると，プロセス内のどのスレッドも DLL 内の関数を呼び出すことができる．また，

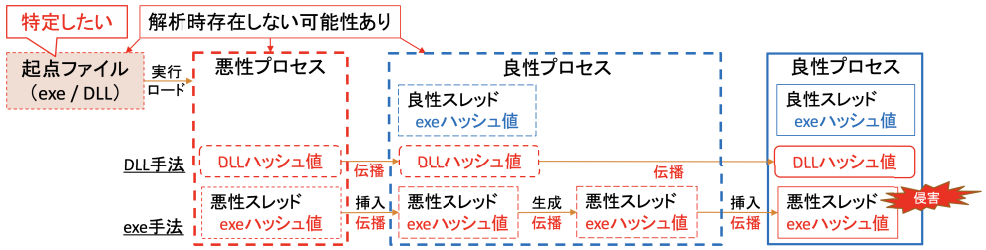


図 2 提案手法の全体像
Fig. 2 Overview of proposed method.

ロードされている DLL のうちスレッド挿入機能を呼び出した関数を含む DLL がどれであるかを区別することは難しいため，ロードされているすべての DLL が悪性である可能性がありうる．よって，提案手法は，起点ファイルが DLL であった場合（図 2 DLL 手法）を想定して，すべての DLL についてロード時にハッシュ値を記録する．これによって，マルウェアが自己隠蔽をしても実行開始時のハッシュ値を保持することができる．なお，DLL のハッシュ値は，プロセス内の全スレッドで共通であるため，プロセス単位で保持すればよい．その後，プロセスをまたがってスレッド挿入がなされるたびにそれらを伝播させる処理を行う．

以上により，フォレンジックス実施時に，起点ファイルや悪性プロセスが存在しない場合においても，悪性スレッドが起点ファイル候補となるファイルのハッシュ値を保持していることから，マルウェアの特定につなげることができる．既知のマルウェアであった場合は検体を VirusTotal 等で検索・取得できる．また，未知のマルウェアであった場合は検体の検索・取得はできないが，フィルタリング等のブラックリスト制御に利用可能である．

この処理を実現するためには，プロセスやスレッドの生成時や DLL のロード時に確実にそれらの生成のイベントを獲得する必要がある．また，ハッシュ値はプロセスやスレッドに対応付けて保持する必要がある．さらに，ハッシュ値等がマルウェアによって改ざんされないようにしなければならない．よって，提案手法では，上述の仕組みを OS カーネル内で実現するとともに，そのハッシュ値をカーネルオブジェクト内に保持することで実現する．

3.2 exe 手法におけるハッシュ値伝播

図 3 に exe 形式の実行ファイルに対するハッシュ値伝播の様子を示す．その動作は以下のようになる．

(1) プロセス（シェル等）が exe 形式の実行ファイルを実行したとき，その処理をフックする．この実行ファイルが起点ファイルとなる．提案手法は，実行ファイルがマルウェアか否かの判定はしないためすべての実行ファイルを対象とする．

(2) フックに対する処理では，起点ファイルのハッシュ値を計算し，生成されたスレッドオブジェクトに記録する．起動時に計算することで，マルウェアが自己隠蔽をした場合でも起動時のハッシュ値を保持できる．

(3) プロセス生成以外のスレッド生成時には，生成元スレッドのスレッドオブジェクトに記録されたハッシュ値を，生成先スレッドに伝播させる．これによって，スレッド生成やスレッド挿入を繰り返す悪性コードであっても起点ファイルのハッシュ値を伝播させることができる．

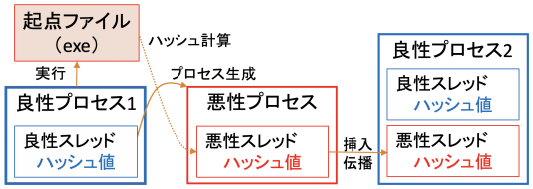


図 3 exe 手法におけるハッシュ値伝播
Fig. 3 Hash value propagation for exe files.

3.3 DLL 手法におけるハッシュ値伝播

本提案方式では，後述のオーバヘッド軽減を目的として，DLL を 2 種類に区別し管理している．1 つは所有者が TrustedInstaller と呼ばれるユーザである DLL（以下，Trusted DLL と記す）であり，もう 1 つはそれ以外の DLL（以下，通常 DLL と記す）である．Trusted DLL は，OS としてインストールされる DLL によく見られ，管理者 administrator を含む他のユーザに書き換え等のアクセス権を与えていない．すなわち，マルウェア等によって改ざんされにくい DLL である．

図 4 に DLL 形式のファイルに対するハッシュ値伝播の様子を示す．DLL はプロセス内の全スレッドで共通であるため，exe 形式とは別のデータ構造を用いて管理をする．具体的には，ハッシュリストと共有ハッシュリスト（図 4 では共有 HL と表記）の 2 種類のデータ構造を用いる．ハッシュリストは，プロセスごとに保持され，当該プロセスがロードしたすべての DLL（Trusted DLL と通常 DLL）のハッシュ値が格納される．共有ハッシュリストは，全プロセスで共有され，全プロセスがこれまでロードした Trusted DLL について，それらのファイル名のハッシュ値（以下，

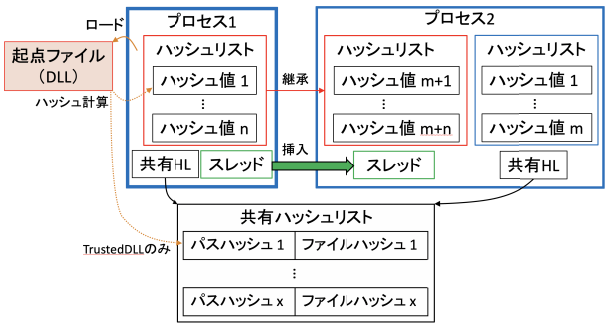


図 4 DLL 手法におけるハッシュ値伝播
Fig. 4 Hash value propagation for DLL files.

パスハッシュと記す）とファイルのハッシュ値（以下，ファイルハッシュと記す．これはハッシュリストのハッシュ値と同等である）が格納される．共有ハッシュリストは，過去に計算した Trusted DLL のハッシュ値のキャッシュとして機能する．パスハッシュは，共有ハッシュリスト内のエントリを高速に検索するために用いる．DLL ファイルは複数読み込まれることが想定されるため，いずれのデータ構造も動的に領域を確保する．以下，動作について示す．

(1) プロセスが DLL をロードしたとき，その処理をフックする．ただし，PE 形式のファイルはすべて DLL としてロードして呼び出し可能であるため，すべてのタイプの PE ファイルを対象とする（以下，このような PE ファイルを含めて DLL と記す）．
(2) フックに対する処理では次のような場合にわけて処理をする．
- Trusted DLL のロードの場合で，共有ハッシュリストに格納されていない場合（初回ロード時），ハッシュ計算し，かつ共有ハッシュリストに格納する．
- Trusted DLL のロードの場合で，共有ハッシュリストに格納されている場合（2 回目以降のロード時），オーバヘッド軽減のためハッシュ計算・格納を省略し，共有ハッシュリスト内のファイルハッシュを取得する．
- 通常 DLL の場合，ハッシュ計算する．

(3) プロセスのハッシュリストについて，手順 (2) で得たハッシュ値が存在しない場合は格納する．
(4) プロセス生成以外のスレッド生成時には，生成元スレッドが属するプロセスのハッシュリストを，生成されたスレッドの属するプロセスに伝播させる．これによって，スレッド生成やスレッド挿入を繰り返す悪性コードに関連する DLL ファイルのハッシュ値を伝播させることができる．

なお，Trusted DLL のハッシュ値は，ハッシュリストと共有ハッシュリストと両方に格納される．共有ハッシュリストに格納することで，広くキャッシュとして活用することができる．ハッシュリストに格納することで，フォレンジックス実施時の調査対象をプロセスに関連するものだけに限定することができる．

共有ハッシュリストはキャッシュとして機能するが，マルウェアが管理者権限を用いて Trusted DLL のアクセス権限を変更したうえでファイルを改ざんする場合が考えられる．この場合に対応するため，ファイルの所有者の変更が行われた場合にもフックし，DLL ファイルの所有者が TrustedInstaller 以外から TrustedInstaller に変更された場合で，共有リスト内に当該ファイルが登録されている場合はその内容を更新する処理を追加している．これによって，改ざん後のハッシュ値を保持することができる．

表 1 評価環境
Table 1 Environment for evaluation.

CPU	Intel Core-i7 7700
メモリ	16 GB
OS	Windows10 64 bit personal build 18363.720

表 2 ハッシュアルゴリズムごとのサイズと計算時間
Table 2 Size and calculation time of each hash algorithm.

ハッシュアルゴリズム	サイズ（bit）	計算時間（ms）
MD5	128	231.08
SHA-1	160	214.72
SHA-256	256	446.65

3.4 ハッシュアルゴリズムについての予備実験と選定

前述のようにハッシュ値を用いて VirusTotal 等を利用したマルウェアの検索・取得を想定している．そのため，利用するハッシュアルゴリズムはそれらの検索サービスで利用可能なものとして，MD5，SHA-1，SHA-256 を候補とした．また，選定には，カーネルオブジェクト内に格納するためハッシュ値のサイズが小さく，計算時間が小さいものであることを条件として検討した．ハッシュの計算時間は，Windows の標準機能である certutil コマンドを利用して 100 MB の実行ファイルのハッシュ値を 100 回計算し，その平均値とした．計測に利用した評価環境を**表 1** に，ハッシュアルゴリズムごとのサイズと計算時間を**表 2** に示す．表 2 より，ハッシュ値のサイズが最小であるのは MD5，ハッシュ計算オーバヘッドが最小であるのは SHA-1 であることが分かった．ただし，MD5 と SHA-1 の計算時間に大きな差はないため，ハッシュ値のサイズが小さい MD5 を採用することとした．

4. 実装

exe 手法におけるスレッド生成時のフックと DLL ロード時のフック，およびそのハッシュ値の管理や伝播処理はすべてカーネルモードドライバを用いて行った．以下，それぞれについて述べる．

```
2: kd> dt nt!_OBJECT_HEADER_CREATOR_INFO ffff808b667e9030
   +0x000 TypeList          : _LIST_ENTRY [ 0xffff808b`667e9030 - 0xffff808b`667e9030 ]
   +0x010 CreatorUniqueProcess : 0x7a636b5b`d5ad2b3f Void
   +0x018 CreatorBackTraceIndex : 0xcbb
   +0x01a Reserved1         : 0x4f41
   +0x01c Reserved2         : 0x8810f9f7
```

図 5 挿入先スレッドの OBJECT_HEADER_CREATOR_INFO 構造体
Fig. 5 OBJECT_HEADER_CREATOR_INFO structure of injected thread.

4.1 exe 手法の実装

スレッドの生成のフックは，カーネル内 API である PsSetCreateThreadNotifyRoutine 関数を用いた．プロセスの生成によるスレッド生成とそれ以外のスレッド生成の区別については，次の 2 つの条件を満たせばプロセス生成のものであると判定した．

- 生成元スレッドが属するプロセスの PID が，生成先スレッドが属するプロセスの親プロセスの PID（EPROCESS 構造体中の InheritedFromUniqueProcessID）と同じである．
- 生成されたプロセスに属するスレッドの数（EPROCESS 構造体中の ActiveThread の値）が 1 つである．

ハッシュ値の格納場所は，スレッドオブジェクトの Optional Headers のうちカーネルが未使用の領域である OBJECT_HEADER_CREATOR_INFO 構造体の領域を流用した．

4.2 DLL 手法の実装

DLL ロード時のフックは，カーネル内 API である PsLoadImageNotifyRoutine 関数を用いた．ただし，この関数は，PE 以外のファイルをロードしたときにもフックをしてしまうため，フック後にファイルのマジックナンバーが “MZ” であることを確認して PE 形式に限定して処理をしている．

また，ファイルの権限変更の検知は，ファイルシステムミニフィルタドライバ（以下，ミニフィルタと記す）を利用した．ミニフィルタでは，ファイル I/O の要求のために Windows 内で送受信される IRP（I/O Request Packet）をフックすることができる．IRP のうち，所有者の変更時に送受信されるものである IRP_MJ_SET_SECURITY をフックし，その内容をチェックすることで実現した．

共有ハッシュリストとハッシュリストの領域は動的にカーネルメモリを確保することとし，プロセスオブジェクトの Optional Headers のうちカーネルが未使用の領域である OBJECT_HEADER_CREATOR_INFO 構造体の領域を流用し，それらへのポインタを格納した．

5. 評価

5.1 exe 手法の評価

5.1.1 機能評価

スレッド挿入を行うマルウェアを模した exe（以下，疑似マルウェアと記す）と，Windows の標準アプリケーションのメモ帳（notepad.exe）を利用し，以下の手順でマルウェアの特定が可能となっていることを確認する．

(1) 疑似マルウェアがメモ帳へ CreateRemoteThread を用いてスレッドを挿入する（挿入先スレッドをスレッド 1 とする）．
(2) メモリダンプをとり，スレッド 1 のスレッドオブジェクト内の OBJECT_HEADER_CREATOR_INFO 構造体の値が疑似マルウェアのハッシュ値と同一か確認し，スレッド挿入時の追跡が機能しているか確認する．
(3) スレッド 1 が，メモ帳プロセス内で新しいスレッドを生成する（生成されたスレッドをスレッド 2 とする）．
(4) 疑似マルウェアとスレッド 1 の実行を終了させる．
(5) メモリダンプをとり，手順 (2) と同様にスレッド 2 のスレッドオブジェクト内の値を確認し，プロセス内でのスレッド生成およびマルウェアの自己隠蔽が行われた場合にも追跡が機能しているかを確認する．

なお，模擬マルウェアのハッシュ値は 3f2badd55b6b637a-bb0c-414f-f7f91088 である．手順 (2) でメモリダンプを解析した結果を**図 5** に示す．図 5 の赤枠内がスレッド 1 の OBJECT_HEADER_CREATOR_INFO 構造体の値である．複数のメンバにまたがっていたりバイトオーダの影響で分かりづらいが，赤枠内 1 行目ではハッシュ値の 3f から 7a までの 8 バイト，2 行目では bb と 0c，3 行目では 41 と 4f，4 行目では f7 から 88 まで 4 バイトが確認できる．

手順 (5) のスレッド 2 の OBJECT_HEADER_CREATOR_INFO 構造体の値についても同様に確認した．以上から，フォレンジックス実施時のメモリダンプからマルウェアの特定が可能となっていることを確認した．

5.1.2 性能評価

提案手法ではプロセス生成時にハッシュ計算を行うため，この部分のオーバヘッドが提案手法の性能に大きく影響する．また，オーバヘッドは実行ファイルのサイズにほぼ比例すると予想されるため，実行ファイルサイズの分布も同様に影響する．そこで，表 1 に示す実環境で，日常的な利用をしたときに発生したハッシュ計算について，対象となる実行ファイルのサイズ，計算時間，計算回数について計測した．日常的な利用とは，著者が 24 時間でブラウザ，オフィスソフト，開発用ソフト等を利用した場合で，明示的には起動を指示していない各種システムアプリケーションの実行も含まれる．なお，本評価では，DLL におけ

表 3 オーバヘッド計測結果
Table 3 Result of performance evaluation.

ファイルサイズ (KB)	平均オーバヘッド (μs) 全ファイル	1 KB あたり	回数
0–100	1,971	31.7	422
100–500	6,584	37.3	1,810
500–1,000	19,009	25.8	69
1,000–20,000	124,754	30.0	32
20,000 以上	–	–	0

表 4 FFRI データセットにおけるクリーンウェアのファイルサイズ分布
Table 4 Distribution of file size of cleanware in FFRI dataset.

実行ファイルサイズ	検体数
30 MB 未満	249,836
30 MB–50 MB	81
50 MB–100 MB	46
100 MB–1 GB	37
1 GB 以上	0

るハッシュ値伝搬機能はオフにしている．

計測の結果を**表 3** に示す．今回の計測では，ファイルサイズが 20 MB を超えるものは観測されず最大は 19.0 MB であった．100–500 KB のものが 77.5%を占めていた．各ファイルサイズ帯における全ファイルを対象とした平均オーバヘッドとしては 125 ms 程度以下となっている．最大値は 19.0 MB のファイルのときの 950 ms であったが，同じファイルが次にアクセスされた際には 222 ms であった．ファイルキャッシュや他のプロセスの動作に影響を受けている可能性がある．各ファイルサイズ帯における 1 KB あたりの平均オーバヘッドは 25.8〜37.3 μs/KB となっている．全ファイルを対象とした場合は 33.3 μs/KB であった．

以上では著者の環境をもとに議論したが，より一般的なケースとして FFRI データセット 2019 [10] に含まれる 25 万件のクリーンウェアのファイルサイズについてその分布を調べた（**表 4** 参照）．クリーンウェアの分布は 99.9%以上が 30 MB 未満となっている．表 1 の環境では，ファイルサイズが 30 MB の場合のオーバヘッドが 1 秒程度と推定できるので，起動時には若干遅くなることもあるが表 3 から頻度としては高くないと推測できる．

5.2 DLL 手法の評価

5.2.1 機能評価

スレッド挿入を行うマルウェアに模した DLL（以下，疑似マルウェアと記す），DLL のロードと指定関数の実行をする Windows 標準ツールの rundll32.exe（以下，rundll と記す），メモ帳を利用し，以下の手順でマルウェアの特定が可能となっていることを確認する．

(1) rundll で疑似マルウェアをロードし，実行する（このプロセスを rundll プロセスと呼ぶ）．

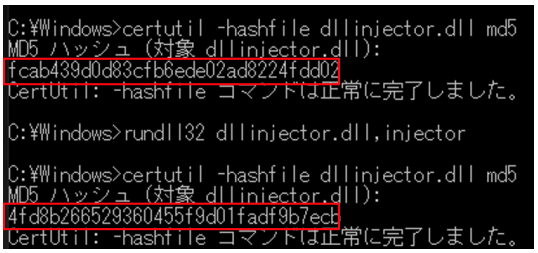


図 6 DLL のハッシュ値
Fig. 6 Hash vaule of DLL.

(2) メモリダンプをとり，rundll のハッシュリスト内に疑似マルウェアのハッシュ値が含まれるか確認し，ロード時の追跡が機能しているか確認する．

(3) プロセス 1 からメモ帳へ CreateRemoteThread を用いてスレッドを挿入する．

(4) メモリダンプをとり，手順 (2) と同様に，メモ帳のハッシュリスト内に疑似マルウェアのハッシュ値が含まれるか確認し，スレッド挿入時の追跡が機能しているか確認する．

また，Trusted DLL が改変された場合に，改変後のハッシュ値が格納されていることを，以下の手順で確認する．

(5) DLL ファイルの所有者を TrustedInstaller に設定し，Trusted DLL を模擬する．

(6) rundll によって当該 DLL をロード・実行する．

(7) 当該 DLL の所有者を変更して DLL を置き換えた後，改変 DLL の所有者を TrustedInstaller に戻し，Trusted DLL の改変を模擬する．

(8) rundll によって改変 DLL をロード・実行する．

(9) メモリダンプをとり，rundll のハッシュリスト内に改変 DLL が含まれるか確認し，ロード時の追跡が機能しているか確認する．また，Trusted DLL については，共有ハッシュリスト内にハッシュ値が格納されているか確認する．

上記の手順を実行した結果，手順 (2) について，メモリダンプからプロセスハッシュリストを探索した結果，rundll プロセスのハッシュリストに疑似マルウェアのハッシュ値が確認された．実験環境ではハッシュリストの 45 番目のエントリに格納されていた．手順 (4) について，メモ帳プロセスのハッシュリストにハッシュ値が伝播していることを確認できた．実験環境ではハッシュリストの 119 番目のエントリに格納されていた．

手順 (5)〜(8) では，パスハッシュが 87397828db9e8632ea0b759f426773a5 の DLL について，ファイルハッシュが**図 6** に示すように fc で始まるハッシュ値（上の赤枠）から 4f で始まるハッシュ値（下の赤枠）に変更した．手順 (9) のときの共有ハッシュリストの内容は**図 7** となっており，当該のパスハッシュ（青枠）のファイルハッシュが変更後のハッシュ値に置き換わっていることが分かる（赤

```
fffff806`5e7391e8  b43c6d84 2280e538 6c0e257b 60e4abb4
fffff806`5e7391f8  6d500662 e5f5cbde 070522dd 0455abd4
fffff806`5e739208  7cb49cd3 c06d9422 453f4795 d995fd5d
fffff806`5e739218  0f2d60ec 6eaad256 ef28cefe 9b371f04
fffff806`5e739228  28783987 32869edb 9f750bea a5736742
fffff806`5e739238  66b2d84f 45609352 fa019d5f cb7e9bdf
fffff806`5e739248  8389b4ef a08dd5fd 3a958253 78ceb8a9
fffff806`5e739258  161e5322 5c685660 a57bd6dd f62d66a3
```

図 7 共有ハッシュリストの内容
Fig. 7 Content of shared hash list.

表 5 OS の起動・シャットダウン時間（ms）
Table 5 Time of boot and shutdown (ms).

	提案手法		オーバヘッド
	なし	あり	
起動	23,770	32,706	8,936（37.6%）
シャットダウン	3,059	3,416	357（11.7%）

枠）．ただし，バイトオーダの影響で，図 7 では 4 バイトごとに下位から読まれたい．

5.2.2 性能評価

DLL に対する手法では，主としてプロセス起動時にロードされる DLL のハッシュ計算におけるオーバヘッドが提案手法の性能に大きく影響する．ただし，Trusted DLL は，OS 標準の DLL であるため多数のプロセスでロードされる一方で，提案手法では初回のみのハッシュ計算となる．よって，以下のような時間を計測することでその性能を評価する．

- OS の起動・シャットダウン時間
- Microsoft Edge（以下，Edge と記す）起動時間

OS の起動・シャットダウンでは，システム全体としてどれくらい性能に影響するかを評価する．特に DLL が多く読み込まれると考えられる起動時は影響が出やすいと推測している．

Edge は一般的に多く利用されるアプリケーションの代表として選択した．なお，Edge の exe と DLL は所有者が TrustedInstaller ではない．一般のアプリケーションを用いて，個々のアプリケーションでどれくらいの影響が出るのかを評価する．

なお，本評価では，exe におけるハッシュ値伝搬機能はオフにしている．

OS の起動・シャットダウン時間

Windows は，起動時間とシャットダウン時間をイベントログに出力する機能を有している．なお，起動時間は，起動後 CPU の利用率が 20%を切るまでの時間とされている．本計測では，Windows 起動時に提案手法のドライバとミニフィルタをロードし，2 分程度待ってからシャットダウンさせた．これを 100 回繰り返し，その平均を算出した（**表 5** 参照）．

表 5 から分かるように，起動時のオーバヘッドは 37%増と小さいとはいえないが，9 秒程度の増加である．シャットダウン時は，基本的にはプロセスの終了が主たる処理であるため，DLL のロードは少なく，その影響は少ない．

表 6 Edge の起動オーバヘッド（ms）
Table 6 Overhead of starting Edge (ms).

提案手法	1 回目	2 回目	3 回目
なし	340	73	75
あり	10,920	1,997	1,986

Edge の起動時間

提案手法は DLL ファイルのロードすべてをフックすることから，アプリケーションの実行の際にもオーバヘッドが発生するため，Edge を対象としてオーバヘッドを計測する．本計測では，アプリケーションの起動時間を，起動後がユーザからの入力を受け付ける状態になるまでとする．本計測では他のアプリケーション等の影響を排除するため OS 起動直後に計測を開始し，アプリケーションの起動を 3 回繰り返した．計測結果から，オーバヘッドにより起動時間が大幅に増加することが明らかとなった（**表 6** 参照）．ただし，2 回目以降の起動では，Trusted DLL のハッシュ計算省略によりオーバヘッドが軽減されていることが分かる．

5.2.3 メモリ使用量の評価

ハッシュリストと共有ハッシュリストは，ロードされる DLL の数によってメモリ使用量が変化する．その数が膨大になるとシステムに影響を及ぼす可能性があるため，メモリ使用量についても評価をした．ここでは，提案手法を適用した環境においてロードされた DLL ファイル数を観測し，そこからメモリ使用量を算出する．プロセスハッシュリストにはロードされた DLL ファイルごとにハッシュ値 1 つが格納され，共有ハッシュリストには DLL ファイルのパスハッシュとファイルハッシュが格納される．MD5 のハッシュ値はいずれも 16 バイトであるので，提案手法のメモリ占有量は，P：プロセス数，S：共有ハッシュリストに格納された PE ファイル情報数，L：ロードされた DLL ファイルの平均数とすると，$16(PL+2S)$ バイトで求められる．

ここで，P，S，L を求めるために，次の手順で実験をした．

(1) Windows を起動
(2) Edge を起動し，10 分待機した後に終了する．
(3) 2 分待機し，Edge の起動が 3 回となるまで手順 (2) に戻る．Edge を 3 回起動したら手順 (4) へ進む．
(4) Windows をシャットダウンする．

本実験の結果，起動からシャットダウンまでのプロセス生成数は 381 回で，DLL のロード状況は**表 7** のようになった．表 7 は，PsLoadImageNotifyRoutine によるフック回数と，その内訳として起動時のロードにとそれ以外の平常時のロードの回数を示している．また，共有ハッシュリスト格納とハッシュリスト格納については，ロードをフック

表 7 ロードされた DLL ファイル数
Table 7 Number of loaded DLL files.

項目			回数
PsLoadImageNotifyRoutine によるフック			22,356
（内訳）	起動時のロード		7,759
	（内訳）	共有ハッシュリスト格納	526
		ハッシュリスト格納	7,279
	平常時のロード		14,597
	（内訳）	共有ハッシュリスト格納	404
		ハッシュリスト格納	13,168

した結果として，それぞれのデータ構造にハッシュ値の格納処理が発生したかを示している．Trusted DLL の場合は，両データ構造にハッシュ値を格納することがあり，その場合は両データ構造でカウントした．一方で，すでに存在するハッシュ値についてはデータ構造への格納をしないため，格納回数としてカウントしない．

Trusted DLL はのべ 930 個がロードされ，通常 DLL はのべ 20,447 個がロードされた．また，平常時における，平均プロセス数は 144.4，共有ハッシュリストのエントリ数は約 830 であった．1 プロセスあたりのハッシュリストの平均エントリ数は 53.7 であった．以上から，平均プロセス数は 144.4 であり，1 プロセスあたり平均 53.7 個の DLL をロードするため，ハッシュリストのメモリ使用量は 124.0 KB と算出できる．一方で，共有ハッシュリストのエントリ数は約 830 個であったため，共有ハッシュリストのメモリ使用量は 26.6 KB と算出できる．よって，本実験では本手法のメモリ使用量が 150.6 KB であったと算出できる．近年のメモリ搭載量と比較すると十分に小さいと考えられ，長期証拠保全に資すると考えられる．

5.3 議論

本章では，CreateRemoteThread を用いてスレッド挿入をする疑似マルウェアを用いて提案手法の機能評価を行った．一方で，FFRI dataset 2013–2017 ではマルウェア 23,138 検体分の Cuckoo sandbox [11] を用いた動的解析ログ分が提供されている．このうち，1,461 検体では CreateRemoteThread が実際に使用され，スレッド挿入が行われていることが確認できる．このようなマルウェアの例としては，リモートアクセス型トロイの木馬である DARKCOMET や，バンキング型トロイの木馬である URSNIF 等が該当する．以上から，疑似マルウェアにおいて CreateRemoteThread の動作を確認したことと，実マルウェアにおいて同一 API が使用されることから，実マルウェアに対しても提案手法を適用可能であるといえる．

6. 提案手法の制限

提案手法は，ハッシュ値の伝播タイミングをスレッド挿入時としているため，悪性コードの伝播にスレッドの生成をともなわないスレッド挿入以外のコード挿入には対応できない．これに該当するものとしては，既存プロセスのコードを入れ替えるプロセスハロウイングがある．

提案手法は，exe と DLL の PE 形式ファイルを対象としている．PowerShell や VBScript 等のスクリプトの場合は，スクリプトをスクリプトエンジンのプロセスやスレッドが実行するため，どのスクリプトエンジンが悪性コードを実行したかは追跡できるが，スクリプト自体のハッシュ値は伝播されない．

フォレンジックス実施時には，スレッドが持つハッシュ値とプロセスが持つハッシュリストに記録されている DLL すべてが起点ファイルの候補となる．特に DLL に関して，スレッドが挿入が繰り返された場合等，プロセスが持つハッシュリストが大きくなる場合が考えられる．

提案手法は，マルウェアの長期潜伏を想定しており，起点となったマルウェアのプロセスおよびそれからスレッド挿入で派生したスレッドのうち，どれか 1 つでも動作している状態でフォレンジックスを実施した場合に有効である．ただし，それらがすべて終了した後にフォレンジックスを実施した場合はマルウェアの情報を取得できない．

7. 関連研究

大月らは，仮想計算機モニタ BitVisor [12] を改良し，マルウェアに検知されにくいマルウェア動的解析環境 Alkanet を実現している [13]．Alkanet では，スレッド生成をすべて記録することができ，その情報からスレッド挿入を追跡することができる．ただし，実利用環境を対象としておらず，取得される動的解析ログが膨大となったり，システムコールが発生するたびにオーバヘッドが発生する．また，コード挿入機能を持つマルウェアへの対応として，山本らは，マルウェアに侵害されていないクリーンな環境のプロセスのメモリイメージと，検査対象の環境のプロセスのメモリイメージとを比較し，挿入されたコードを特定する手法を提案している [14], [15]．ただし，やはり実利用環境を対象としておらず，クリーンな環境を準備して比較する手順が必要であり，適切なプロセス選択をして比較しなければマルウェアの自己隠蔽機能によりその痕跡を検出できない可能性もある．提案手法は，カーネルモードドライバとミニフィルタドライバをロードするだけで実利用環境に適用可能であり，オーバヘッドも比較的小さい．ログに頼らないため長期潜伏したマルウェアにも対応でき，すべてのスレッドについて起点ファイルのハッシュ値を残すことができる．

Lin らは，他プロセスへのメモリ確保，メモリ書き込み，スレッド挿入を検出し，ユーザに対してスレッド挿入が発生したことをユーザへ通知する手法を提案している [16]．また，本手法と同様にカーネルモードドライバのロードのみで適用可能としている．ただし，ユーザへの通知を目的

としているため，悪性挙動に関する情報はプロセスハンドルとプロセスのベースアドレスのみとなっている．このため，自己隠蔽機能を有するようなマルウェアの場合は，その特定は難しい．提案手法は，マルウェアのハッシュ値を伝播させることで，悪性スレッドが生存している限りはマルウェアの特定まで可能としている．

8. おわりに

本論文では，フォレンジックス実施時に有用な情報としてスレッド挿入機能を有するマルウェアのハッシュ値を，マルウェアの潜伏期間が長期にわたる場合にも適した長期証拠保全手法を提案した．具体的には，マルウェアの exe や DLL のハッシュ値をカーネル内に保持し，スレッド挿入による悪性スレッドの伝播と同時にハッシュ値を伝播させる手法を実現した．機能評価ではその動作を確認するとともに，性能評価では一般的な環境では問題とならないオーバヘッドであることを確認した．

参考文献

[1] キヤノンマーケティングジャパン株式会社：2019 年年間マルウェアレポート，入手先 ⟨https://eset-info.canon-its.jp/files/user/malware_info/images/ranking/pdf/MalwareReport_2019.pdf⟩（参照 2020-06-16）．

[2] 朝長秀誠，中村　祐：日本の組織をターゲットにした攻撃キャンペーンの詳細，入手先 ⟨https://www.jpcert.or.jp/present/2015/20151028_codeblue_ja.pdf⟩（参照 2021-01-18）．

[3] 船越絢香，中村　祐，竹田春樹：標的型攻撃で用いられたマルウエアの特徴と攻撃の影響範囲の関係に関する考察，コンピュータセキュリティシンポジウム 2015 論文集，Vol.2015, No.3, pp.963–970 (2015).

[4] 丹田　賢：仮想環境に依存せず詳細な解析能力を持つ動的解析システムの設計と実装，入手先 ⟨http://www.ffri.jp/assets/files/research/research_papers/Dynamic_malware_analyzer_without_virtual_environments_ja.pdf⟩（参照 2021-01-18）．

[5] Kawakoya, Y., Iwamura, M., Shioji, E. and Hariu, T.: API Chaser: Anti-analysis Resistant Malware Analyzer, *RAID 2013: Research in Attacks, Intrusions, and Defenses*, Vol.8145, pp.123–143 (2013).

[6] Google Inc.: Virus Total, available from ⟨https://www.virustotal.com/⟩ (accessed 2021-06-26).

[7] Ligh, M.H., Case, A., Levy, J. and Walters, A.: *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, John Wiley & Sons, Inc. (2014).

[8] Russinovich, M. and Garnier, T.: Sysmon v10.42, available from ⟨https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon⟩ (accessed 2019-12-02).

[9] ファイア・アイ株式会社：M-Trends 2020 レポート，入手先 ⟨https://content.fireeye.com/m-trends-jp/rpt-m-trends-2020-jp⟩（参照 2020-06-16）．

[10] 荒木粧子，笠間貴弘，押場博光，千葉大紀，畑田充弘，寺田真敏：マルウェア対策のための研究用データセット～MWS Datasets 2019，情報処理学会研究報告，Vol.2019-CSEC-86, No.8, pp.1–8 (2019).

[11] Stichting Cuckoo Foundation: Cuckoo Sandbox: Automated Malware Analysis, available from ⟨https://cuckoosandbox.org/⟩ (accessed 2021-06-26).

[12] Shinagawa, T., Eiraku, H., Omote, K., Hasegawa, S., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y. and Kato, K.: BitVisor: A Thin Hypervisor for Enforcing I/O Device Security, *Proc. 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp.121–130 (2009).

[13] Otsuki, Y., Takimoto, E., Kashiyama, T., Saito, S., Cooper, E.W. and Mouri, K.: Tracing Malicious Injected Threads Using Alkanet Malware Analyzer, *IAENG Trans. Engineering Technologies*, Vol.LNEE 247, pp.283–299 (2014).

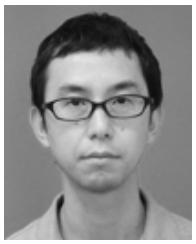
[14] 山本　匠，河内清人，桜井鐘治：不審プロセス特定手法の提案，コンピュータセキュリティシンポジウム 2013 論文集，Vol.2013, No.4, pp.634–641 (2013).

[15] 山本　匠，河内清人，桜井鐘治：不審プロセス特定手法の実装及び評価，情報処理学会研究報告，Vol.2014-DPS-158, No.33, pp.1–8 (2014).

[16] Lin, Y.-H., Lin, Y.-C., Sun, H.-M., Tseng, Y. and Chiang, T.-C.: Detecting the code injection by hooking system calls in windows kernel mode, *Proc. International Computer Symposium 2006*, pp.862–867 (2006).

田中 大樹　（正会員）

1996 年生．2019 年立命館大学情報理工学部情報システム学科卒業，2021 年同大学大学院情報理工学研究科博士前期課程情報理工学専攻修了．同年日本電気株式会社入社，現在に至る．セキュリティの研究に従事．



川古谷 裕平　（正会員）

2003 年早稲田大学理工学部情報学科卒業，2005 年早稲田大学大学院理工学研究科修士課程修了，2019 年早稲田大学博士（工学）．2005 年日本電信電話株式会社に入社，社会情報研究所勤務（現職）．マルウェア対策の研究開発に従事．電子情報通信学会会員．

岩村 誠　（正会員）

2000 年早稲田大学理工学部情報学科卒業，2002 年早稲田大学大学院理工学研究科修士課程修了，2012 年早稲田大学博士（工学）．2002 年日本電信電話株式会社に入社，社会情報研究所勤務（現職）．サイバー攻撃対策の研究開発に従事．電子情報通信学会会員．

鄭 俊俊

1987 年生．2010 年福建師範大学（中国）工学部ソフトウェア工学学科卒業，2013 年広島大学大学院工学研究科情報工学専攻博士課程前期修了，2016 年同大学大学院工学研究科情報工学専攻博士課程後期修了．同年同大学大学院工学研究科外国人客員研究員，2018 年立命館大学情報理工学部助教となり，現在に至る．博士（工学）．ソフトウェア信頼性，モデリングと性能評価，応用確率論，ソフトウェアフォールトトレラント技術等の研究に従事．電子情報通信学会，日本オペレーションズ・リサーチ学会，日本信頼性学会，IEEE 各会員．

毛利 公一 （正会員）

1972 年生．1994 年立命館大学理工学部情報工学科卒業，1996 年同大学大学院理工学研究科修士課程情報システム学専攻修了，1999 年同大学大学院理工学研究科博士課程後期課程総合理工学専攻修了．同年東京農工大学工学部情報コミュニケーション工学科助手，2002 年立命館大学理工学部情報学科講師，2004 年同大学情報理工学部情報システム学科講師，2008 年同准教授，2014 年同教授となり，現在に至る．博士（工学）．オペレーティングシステム，仮想化技術，コンピュータセキュリティ，コンピュータネットワーク等の研究に従事．電子情報通信学会，ACM，IEEE-CS，USENIX 各会員．