修士論文

回避型マルウェア解析のための
回避コード抽出に関する研究

西田 雄亮
奈良先端科学技術大学院大学
先端科学技術研究科
情報理工学プログラム

主指導教員: 門林 雄基 教授
サイバーレジリエンス構成学研究室（情報科学領域）

令和 2 年 3 月 13 日提出

本論文は奈良先端科学技術大学院大学先端科学技術研究科に
修士 (工学) 授与の要件として提出した修士論文である.

西田 雄亮

審査委員：

門林 雄基 教授	（主指導教員，情報科学領域）
林 優一 教授	（副指導教員，情報科学領域）
中島 康彦 教授	（副指導教員，情報科学領域）
妙中 雄三 准教授	（副指導教員，情報科学領域）

回避型マルウェア解析のための回避コード抽出に関する研究*

西田 雄亮

内容梗概

マルウェアの増加に伴い，手動によるマルウェア分析は難しくなっている．そこで，検体を解析環境上で実行し，その振る舞いから解析する動的解析が広く用いられている．しかし，解析環境上では無害な挙動をとることで，動的解析を回避する回避型マルウェアも増加している．回避型マルウェアの解析手法は数多く提案されている一方で，マルウェアによる解析回避方法も複雑さが増している．このため，マルウェアが実行する回避コードを分析し，回避型マルウェア解析技術を継続的に高度化していくことが必要不可欠である．本研究では，回避型マルウェアの多様な回避技術を特定することを目的として，動的解析・静的解析を組み合わせた回避コードの抽出手法を提案する．本提案手法は，回避型マルウェアの「解析環境では無害挙動をとる」特性に注目し，動的解析で無害関数の情報を記録し，静的解析で動的解析から得られた情報を用いて，プログラム実行時に回避コードの疑いがある基本ブロックを自動で抽出する．提案手法に基づいたプロトタイプを実装し，擬似検体を用いて評価した結果，回避コードを 50.0〜60.0% 抽出することができた．

キーワード

回避型マルウェア, マルウェア, 動的解析, 静的解析, セキュリティ

*奈良先端科学技術大学院大学 先端科学技術研究科 修士論文, 令和 2 年 3 月 13 日.

Evasive Code Extraction Technique for Evasive Malware Analysis*

Yusuke Nishida

Abstract

As nowadays, the variety of malware are increasing, it's getting more and more difficult to analyse each single one manually. Therefore, dynamic analysis where malware samples run on an analysis environment is used as a standard. However, a type of malware that conceal their harmful behaviour in the analysis environment is also increasing, called evasive malware. There are many research attempts to detect to evasive malware, but evasive malware is also be strengthen accordingly. Therefore, we need to improve the analysis environment to overcome evasive malware tricks. In this research, we propose a methodology to extract suspect evasive basic block by the combination of both dynamic and static analysis for identifying various evasive method. Our method automatically extracts basic block where suspected evasive code is by using evasive malware's characters which execute harmless code on analysis environment. We evaluated our idea by using by using synthetic malwares and be able to get hidden basic block which is suspect to evasive code extracted.

Keywords:

Evasive Malware, Malware, Dynamic Analysis, Static Analysis, Security

*Master's Thesis, Graduate School of Science and Technology, Nara Institute of Science and Technology, March 13, 2020.

目 次

図 目 次

表 目 次

アルゴリズム一覧

コード一覧

1. はじめに

1.1 研究背景

マルウェア（Malicious Software; Malware）とは，悪意のあるソフトウェアのことを指す．マルウェアの活動は，感染マシンの破壊や情報漏えいといった被害を与えるものだけでなく，感染マシンから他のマシンへ攻撃を行うものまで存在する [1]．感染マシンが被害者になるだけでなく加害者になる場合まであるため，マルウェアを対策することは重要であるといえる．マルウェアの対策・駆除を行うためには，そのマルウェアがどのような被害を与えるのか，またどのような特徴を持っているのかを知る必要があるため，マルウェアの解析を行う必要がある．しかし，マルウェアを手動で解析することは，専門の知識と多くの時間を必要とする．そこで，人的コストと時間コストを削減するために，静的解析や動的解析を用いたマルウェア解析の自動化が行われるようになった [2]．

静的解析とは，マルウェアであることが疑われる検体のバイナリコードから解析する手法である [2]．そのため，検体のコードセグメント全体を解析することができる [3]．静的解析では，シグネチャ検知や，逆アセンブリ，制御フローグラフ（Control Flow Graph; CFG）などを用いて解析する [4]．CFG とは，基本ブロック（Basic Block）同士の制御関係を表したグラフを指す．基本ブロックとは，分岐命令を含まない命令の集合を指す [5]．一方で，多くの回避型マルウェアは，静的解析を回避する手法を取り入れている．しかし，アンパッキングやメモリダンプの情報を用いることで，オリジナルのコードを復元できる場合がある [2, 3, 4]．

動的解析とは，検体を解析環境上で実行し，解析環境にどのような影響を与えたかを追跡することで解析する手法である [2]．静的解析と比べ，得られる情報が多く，静的解析の回避手法に影響されないため，効果的な手法である [2, 3]．しかし，解析できる範囲は，解析環境で実行された範囲に絞られる．したがって，解析環境では無害な振る舞いを示す回避型マルウェアでは，解析者に誤った結果を与えてしまうため，対策の遅れに伴うマルウェアの流行や，脅威の過小評価へとつながる恐れがある [2, 3, 4]．

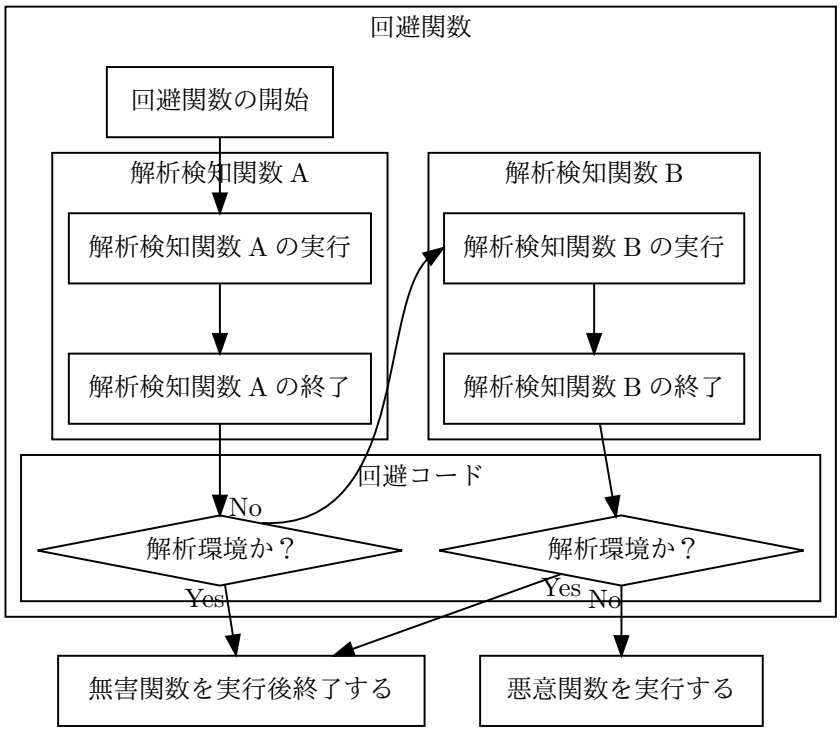

図 1　回避関数の構成例

回避型マルウェアは，解析環境検出のために，解析環境と実際の環境で区別可能な環境属性を取得する．例えば，Web ブラウザの履歴や，ユーザー名などが挙げられる [6]．従来の研究では，環境属性から解析環境の検出を難しくする手法が数多く提案されてきた．しかし，環境属性は無数に存在するため，未知のマルウェアが，従来の研究で考慮されていない環境属性を採用する可能性は十分に考えられる．そのようなケースに対応するためには，検体がどのような環境属性を採用しているのかを知る必要がある．

環境属性は，解析環境を検出するために用いるので，解析環境か否かを判断する回避関数で用いられる．その構成は，図 1 のようになっていることが考えられる．この場合，環境属性に関わらず，解析されていると判断したときに実行される無害関数と，実際の環境であると判断したときに実行される悪意関数へ分岐する制御構造がコードセグメントに含まれるということになる．以後，回避関数内において，この制御構造を持つ基本ブロックを，回避コードと呼ぶことにする．

この関数には環境属性の取得が含まれているため，回避コードの特定によって，環境属性も特定することができることが考えられる．しかし，回避コードを特定するために，制御構造を手動で網羅的に分析することは，マルウェアの手動解析と同様に，人的コストと時間コストが大きい作業である．

1.2 研究目的

本研究では，動的解析を妨害する回避型マルウェアを対象に，環境属性を取得する場所の抽出を目的とする．1.1 節で述べたように，回避コードを特定することによって，その関数から環境属性を取得することができる．したがって，回避コードの特定により，環境属性を取得する場所を抽出する．

1.3 本論文の構成

本論文では，提案手法のプロトタイプを実装した上で，疑似検体を用いて提案手法に対する評価を行う．

本論文の構成は以下のとおりである．2 章で関連研究を述べ，3 章では，本研究で想定する回避型マルウェアについて説明する．その後，4 章で提案手法を述べ，5 章で，擬似検体を用いた提案手法の評価を行う．6 章では，5 章の結果をもとに，提案手法の有効性について議論を行う．最後に，7 章で，本論文の結論を述べる．

2. 関連研究

本章では，まず環境属性に応じた対策手法を取り上げる．その後，環境属性に依存しない対策手法も取り上げ，最後に本研究の位置づけを述べる．

2.1 仮想環境の環境属性を用いた解析環境検知

解析環境は，検体解析後に元の状態へ復元するために，スナップショット機能を必要とする．その機能を実現するために，VMware [7] や VirtualBox [8] といった，仮想化ソフトウェアを用いる場合がある．仮想化ソフトウェア上で動くゲスト OS は，様々な環境属性が存在する．したがって，ゲスト OS 上のプログラムは，自身がゲスト OS 上で動作しているかどうかを検知することができる [9]．回避型マルウェアの中には，自身がゲスト OS 上で動作していれば，解析環境と判断するものも存在する．この対策として，様々な環境で検体を実行し，各環境における振る舞いの違いから回避型マルウェアを解析する手法が提案されている [6]．この手法は，用意した環境のうち，少なくとも 1 つの環境では，悪意挙動が示される環境である必要がある．しかし，用意した環境がすべて仮想環境であれば，すべての環境で回避型マルウェアが仮想環境を検知し，悪意挙動が示されない可能性も考えられる．

そこで，仮想環境を用いない解析環境が提案された．論文 [10] では，仮想環境を用いずにスナップショット機能を実現する手法が提案された．論文 [11] では，回避型マルウェアの回避挙動を抑えるために，プロセッサーの条件分岐監視機能を用いたフレームワークが開発された．このフレームワークを用いて，条件分岐のトレースを振る舞いの違いとして比較することで，回避コードを抽出する手法を提案している．しかし，フレームワークの導入にあたり，監視システムとして，カーネルドライバとクライアントを解析環境に導入する必要がある．そのため，監視システムが環境属性となってしまい，回避型マルウェアによって検知される可能性がある．このような監視システムを導入しない場合，観測できる情報は，検体の起動前後におけるレジストリやファイルの変化やネットワークパケットの取得に限定される [12]．したがって，環境属性を消すことは，観測できる振る舞

いに制限を与えるものといえる [6].

2.2 仮想環境以外の環境属性を用いた解析環境検知

横山ら [13] は，解析環境の特性により環境属性が生じることを示した．例えば，システムの起動時間や，最後にログインされた時間などは，解析環境のスナップショットが使い回しであれば，解析環境ごとに環境属性が現れる．この環境属性を収集するプログラムである SandPrint を，20 個のマルウェア分析サービスに検体として投稿した結果，2,666 個の収集レポートから 76 個の解析環境に分類できたことが報告されている [13].

Miramirkhani ら [14] は，ユーザがシステムを使用することによって生じる変化が環境属性となることを示した．それらの環境属性は，システムが利用された日数と相関関係を持つため，それから環境属性の特徴量を予測するモデルが提案されている．

本節で述べた環境属性は，先行研究が選択したもの以外にも多数存在することが示唆されている．未知の環境属性は，仮想環境に関連するものであれば，2.1 節のように，仮想環境を用いないことで対策することができた．しかし，解析環境に関連するものである場合，解析環境を用いないという選択は不可能である．この課題を克服するための一つのアプローチとして，検体を実行するごとに，実際の環境を完全にクローンしたスナップショットを作成した解析環境を用いる方法が考えられる．しかし，クローンされる実際の環境を複数用意する必要があること，情報流出を目的としたマルウェアの場合，実際の環境を用いることにはプライバシーの観点から問題があること，また標的型攻撃の場合，攻撃対象特有の環境属性を用いる可能性があるという観点で課題が残る．

2.3 環境属性に依存しない対策手法

論文 [15] では，条件分岐命令が実行されたときにスナップショットを記録し，非実行経路が動的解析できるように，解析環境上の入力値を変更して再実行する手法を提案した．なお，すべての条件分岐を対象とすると対象数が非常に多くな

るため，システムコールの戻り値が条件式に用いられている条件分岐命令のみに限定されている．しかし，システムコールの戻り値を条件式に用いる条件分岐を大量に設けることで，対象数を強制的に増加できてしまう課題も存在する [15]．

2.4 本研究の位置づけ

本章をまとめると，回避型マルウェア解析は，以下のような課題が潜んでいるものと考える．

- 解析環境に関連する環境属性の対策が難しいこと
- 解析環境と実際の環境の境界点となる環境属性の特徴量が不明であること
- 検体が持つ条件分岐のうち，悪意関数へ到達する条件分岐がどれであるか不明なこと

今後は，未知の環境属性を採用した回避型マルウェアが登場することは十分に考えられる．そのような回避型マルウェアを解析するためには，手動で検体を解析する必要があるものと考える．しかし，手動での解析は人的コストと時間的コストが大きい作業である．

そこで，動的解析と静的解析を用いて，回避コードを自動で特定する手法を検討する．これにより，解析すべき範囲を絞り込むことができるため，時間的コストを小さくできることが期待される．

3. 回避型マルウェアの想定モデル

本章では，本研究における提案手法や評価の際に用いる仮定事項を示すために，想定する回避型マルウェアのモデルについて述べる．

3.1 事前調査

本節では，回避型マルウェアにおける回避関数が，どのようにして実装されているのかを調べる．

Cuckoo Sandbox [16] とは，オープンソースで開発されている動的解析システムの一つであり，マルウェアデータセットの一つである FFRI Dataset 2013〜2017 や Soliton Dataset などに用いられている [17]．このシステムは，動的解析で検出する振る舞いをシグネチャとして定義することができ，様々なシグネチャがコミュニティによって提供されている．サンドボックスや VM の検知を検出するシグネチャも含まれているため，よく知られた回避手法を採用した回避型マルウェアであれば検出することができる．大山ら [18] は，FFRI Dataset 2016 より，コミュニティが提供したシグネチャによって検知された結果と Windows API のコールシーケンスから， 回避型マルウェアが持つ耐解析手法の傾向を報告した．

Paranoid Fish（Pafish） [9] とは，オープンソースで開発されている解析環境検知システムであり，マルウェアが使用するいくつかの環境属性の取得が実装されている．このプログラムを，Cuckoo Sandbox で解析すると，コミュニティによって提供されたシグネチャによって，以下の振る舞いが検知される．

- Looks for known filepaths where sandboxes execute samples (4 events)
- Checks the version of Bios, possibly for anti-virtualization (2 events)
- Attempts to detect a virtual machine by the use of a pseudo device (2 events)
- Detects Joe or Anubis Sandboxes through the presence of a file (1 event)
- Detects VirtualBox through the presence of a device (4 events)

- Detects VirtualBox through the presence of a file (16 events)
- Detects VirtualBox through the presence of a registry key (4 events)
- Detects VirtualBox using WNetGetProviderName trick (1 event)
- Detects VirtualBox through the presence of a window (2 events)
- Detects VMWare through the presence of various files (4 events)
- Detects VMWare through the presence of a registry key (1 event)
- Detects the presence of Wine emulator (2 events)

この結果より，Pafish は，回避型マルウェアと同様に，Cuckoo Sandbox のシグネチャによって回避挙動を検知されることが分かる．Pafish 自体は，ある解析環境検知手法を用いた結果，解析環境の検知に成功したか否かを標準出力するものである．一方で，回避型マルウェアでは，標準出力の代わりに，無害関数と悪意関数の実行に置き換わっているものと考えられる．したがって，Pafish の構造は回避型マルウェアの回避関数と類似しているものと考え，Pafish のソースコードから，回避型マルウェアにおける回避関数の構造を推定することにした．

Pafish のソースコードは，検知関数と出力関数の 2 つに分かれている．

検知関数

環境属性を取得し，解析環境を検知できたか否かをブール値で返す．

出力関数

検知関数を引数で受け取り，その返り値を if-else 文を用いて標準出力を制御する．

したがって，Pafish は，if-else 文を用いた制御構造を用いて，解析環境の検知に成功したか否かを標準出力することが見られる．

3.2 想定モデル

Pafish には， 2.2 節で述べたような，仮想環境以外の環境属性を用いた解析環境の検知手法はほとんど含まれていない．そのため，仮想環境以外の環境属性を用いた解析環境の検知では，どのようにして回避関数が実装されているのかを議論する必要がある．仮想環境以外の環境属性を用いた解析環境の検知手法について記した論文 [13, 14] では，ある環境属性が，ユーザ環境に由来するものなのか否かを，機械学習を用いて分類した．その分類結果を回避型マルウェアに実装するためには，決定境界を制御構造として実装するものと考えられる．3.1 節で述べたように，Pafish における制御構造の実装は，if-else 文を用いていたため，仮想環境以外の環境属性でも同様であると考える．

したがって，回避型マルウェアの回避関数は，if-else 文を用いた制御構造を用いているものと考える．if-else 文を用いた制御構造は，分岐命令として出力される．このことより，本研究で扱う回避型マルウェアの想定モデルは，環境属性の結果によって，実行経路が無害関数または悪意関数に分岐する分岐命令が含まれているものとする．

4. 提案手法

本章では，回避型マルウェアが解析環境上では無害挙動をとることを活用して，回避コードを特定する手法について説明する．なお，本手法は，OS に関係なく適用できる手法ではあるが，マルウェアの対象に Windows が多いことを踏まえ，Windows を対象として検討する．

図 2 に提案手法の概要図を示す．白抜き矢印で構成される図形は，回避型マルウェアの回避関数におけるフローチャートを示している．黒矢印は，情報の流れを，そして破線の黒矢印は，静的解析が調査する箇所を示している．提案手法は，動的解析と静的解析の 2 つの解析手法から成り立っている．

動的解析では，回避型マルウェアによって検知され，無害な振る舞いが示される解析環境をあえて用いる．解析環境上では，実行中の検体に関して，以下の情報を収集する．これにより，動的解析では，検体の無害関数に関する情報を得ることができる．収集した情報は，静的解析に渡される．

- 検体がコードセグメント中で実行したアドレス
- 条件分岐命令の実行可否とそのアドレス
- 解析環境にマッピングされている検体モジュールの基底アドレス

静的解析では，動的解析から渡された情報を用いて，回避コードを抽出する．そのため，検体の回避コードは，静的解析を回避する手法が取り除かれた状態であることを前提とする．回避コードの抽出にあたり，以下の手法を用いる．

- 無害関数への到達可能性による回避コード検知
- Windows API の呼び出し有無によるカットオフ処理

本手法の利点は，悪意ある挙動が解析環境で再現されるようにする必要がない点にある．そのため，解析環境が検知されないように工夫する必要がない．

動的解析の詳細については，4.1 節で述べ，静的解析の詳細については，4.2 節で述べる．そして，実装については，4.3 節で述べる．

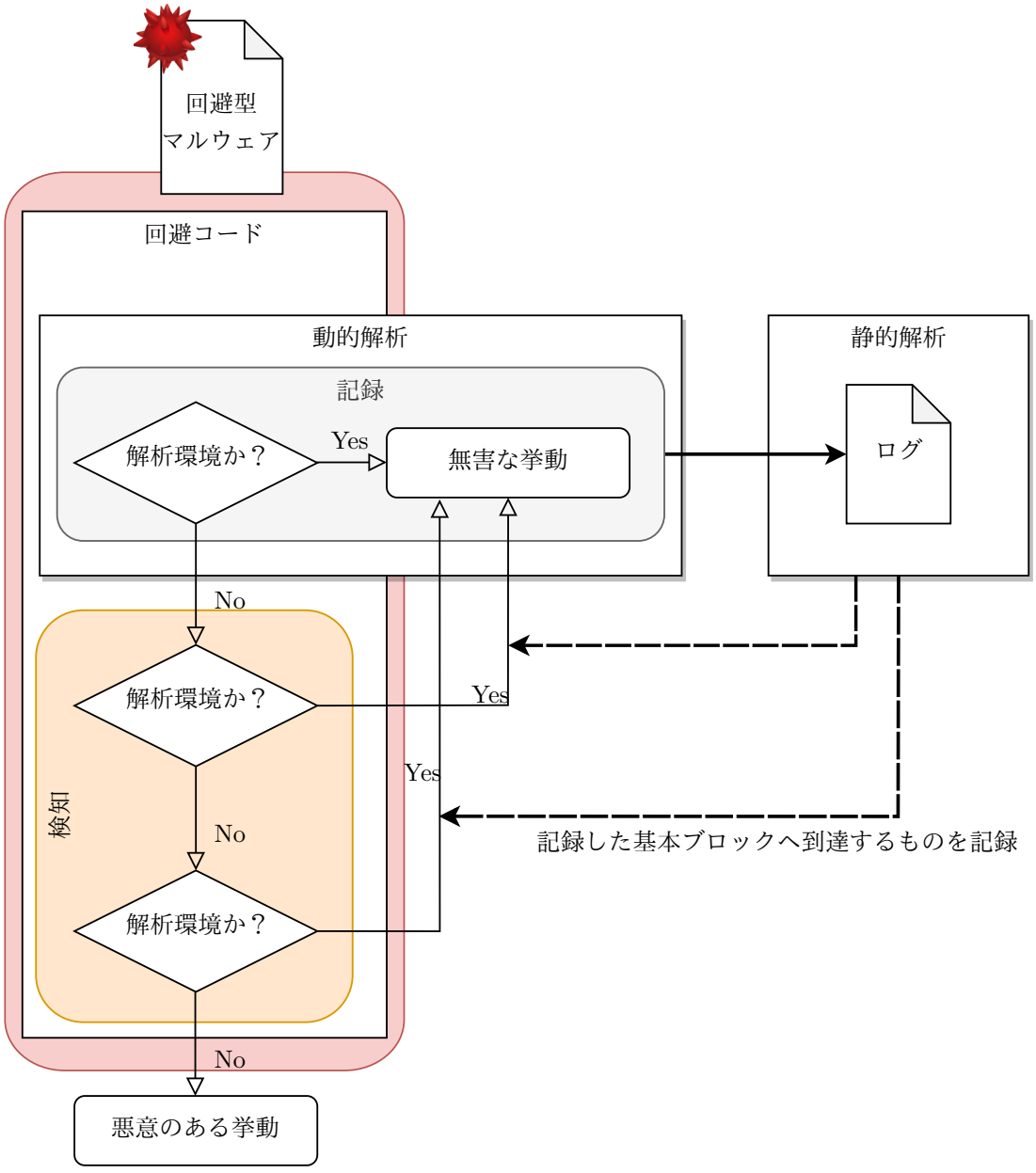
回避型
マルウェア
回避コード
動的解析
記録
解析環境か？
Yes
無害な挙動
静的解析
ログ
No
解析環境か？
Yes
Yes
検知
No
記録した基本ブロックへ到達するものを記録
解析環境か？
No
悪意のある挙動

図 2　提案手法の概要図

アルゴリズム 1: 検体実行基本ブロック記録のアルゴリズム

Input: 実行する検体の基本ブロック X

```
for i ∈ X の命令 do
    命令 i のアドレスを記録
```

4.1 動的解析を用いた無害関数の情報収集

動的解析で用いる検体は，マルウェアであることが疑われるのにも関わらず，動的解析で不審な振る舞いを示さなかった検体であるとする．このような検体は，解析環境上で無害な振る舞いを示したことが考えられる．そのような検体において，以下の情報を記録する．

検体がコードセグメント中で実行したアドレス

無害な振る舞いを示す基本ブロックを特定するために記録する．

解析環境で検体が無害な振る舞いを示すということは，無害関数が実行されたということである．無害関数がどのようにして実装されているかは検体によって異なるため，無害関数のみを抽出することは難しい．

したがって，このフェーズでは，検体が実行した基本ブロックすべてを，無害関数とみなすことにする．無害関数とみなす基本ブロックの特定が目的であるため，実行された順番は重要ではない．

検体が実行した基本ブロックのアドレスを取得するアルゴリズムを，アルゴリズム 1 に示す．入力する基本ブロックは，検体がこれから実行しようとする基本ブロックである．

条件分岐命令の実行可否とそのアドレス

解析環境で実行されなかった基本ブロックを特定するために記録する．前述したように，解析環境で実行された基本ブロックは無害関数とみなしているため，実行されなかった方から回避コードを抽出する必要があるためである．

アルゴリズム 2: 条件分岐命令記録アルゴリズム

Input: 実行する検体の基本ブロック X

```
1 N ⟵ 基本ブロックの最後の命令 (X)
2 if N は条件分岐命令 then
3     S ⟵ i のジャンプ先が指すアドレス
4     P ⟵ 次の命令のアドレス
5     S と P を記録
6     S の基本ブロックに実行記録関数を追加
7     P の基本ブロックに実行記録関数を追加
```

プログラムは，分岐命令によって振る舞いに変化が生じる．分岐命令には，直接ジャンプ (Direct Jump) と，間接ジャンプ (Indirect Jump) の 2 つが存在する．直接ジャンプは，分岐先のアドレスが絶対値または相対値で指定されているため，静的解析から一意に決定することができる．一方で，間接ジャンプ (Indirect Jump) は，分岐先のアドレスがレジスタまたはアドレスの先に記されている値が対象となる．したがって，ジャンプ先を動的に変更することが可能であり，静的解析から特定することが難しい．

条件分岐命令は，x86 アーキテクチャであれば直接ジャンプである．これを踏まえ，解析環境で実行された条件分岐命令から，実行されなかった方のアドレスを取得するアプローチをとる．

解析環境上で条件分岐命令が実行されるときに，その条件が真だった場合にジャンプするアドレス，偽の場合にジャンプするアドレスを記録することができる．それに加えて，解析環境でどちらにジャンプしたかを記録する．

条件分岐命令の実行可否とそのアドレスを取得するアルゴリズムを，アルゴリズム 2 に示す．入力する基本ブロックは，検体がこれから実行しようとする基本ブロックである．アルゴリズム中における実行記録関数とは，その基本ブロックが実行されたことを記録する関数である．条件分岐命令が実行される場合とされない場合の基本ブロックにそれぞれ実行記録関数を追加することで，その条件分岐が実行されたのかどうかを判別することができる．

解析環境にマッピングされている検体モジュールの基底アドレス

前述したアドレスは，すべて検体の実行中に記録するため，仮想アドレス (Virtual Address; VA) として記録される．VA は，アドレス空間配置のランダム化 (Address Space Layout Randomization; ASLR) が有効になっている場合，検体がロードされた際のセクションの基底アドレス (Base Address; BA) が，OS の起動ごとにランダム化されているため，検体のヘッダに示された BA と一致しない．

したがって，動的解析のアドレスを静的解析で用いることができないため，相対仮想アドレス (Relative Virtual Address; RVA) に変換する．VA から RVA の変換式は式 1 のとおりである．ここで，BA はプロセス空間における BA である．

$$\mathrm{RVA} = \mathrm{VA} - \mathrm{BA} \tag{1}$$

ゆえに，RVA の計算に，解析環境にマッピングされている検体モジュールの BA が必要となるために記録する．

4.2 静的解析による回避コード抽出手法

静的解析では，無害関数へ分岐する基本ブロックを，回避コードとして抽出する．解析環境で実行した基本ブロックすべてを無害関数とみなすため，誤検知が多く含まれることが予想される．そのため，回避コードを抽出した後に，誤検知と思われる基本ブロックを取り除く．

本節では，それぞれのフェーズについて説明する．

無害関数への到達可能性による回避コード抽出

回避コードは，解析環境と判断した場合に無害な振る舞いを行う必要があるため，回避コードから無害関数へ到達するフローが存在することが考えられる．解析環境で実行したコード全体が無害関数とみなしているため，解析環境で実行されていない基本ブロックを対象に，解析環境で実行したコードへ到達するか否かを以って回避コードを検知する．

解析環境で実行したコードへ到達するかを判断するアルゴリズムを，アルゴリズム 3 に示す．入力する基本ブロックは，解析環境で実行されなかった基本ブロックである．

アルゴリズムの流れとしては，入力された基本ブロックが，動的解析で記録された基本ブロックのアドレスへジャンプするか否かを調べるものである．しかし，間接ジャンプはジャンプ先が明確に示されないため，アルゴリズムの適用が難しいという課題が存在する．

そこで，ジャンプ先が不明な間接ジャンプ命令に対しては，そのジャンプ命令を無視することにする．間接ジャンプは，x86 アーキテクチャの場合，CALL 命令か JMP 命令のどちらかである．

CALL 命令の場合は，本来の用途で用いられていれば，ジャンプ先の命令が実行された後に，もとの場所へ制御が戻る．したがって，CALL 命令の次に含まれる命令が属する基本ブロックに対して，アルゴリズム 3 を実行する．

JMP 命令の場合は，CALL 命令と異なり，もとの場所へ制御が戻るとは限らない．そのため，解析環境で実行したコードへジャンプしなかったものとする．

Windows API の呼び出し有無によるカットオフ処理

前節で述べたように，検体が実行した基本ブロックのアドレスを無害関数とみなしているため，誤検知が多く含まれることが予想される．環境属性は，ハードウェアやシステムの情報であることが多い [3, 6]．このような情報を取得するためには，システムコールが呼び出される．Windows の場合，システムコールは Windows API によるラッパーから呼び出される．そこで，回避コードが所属する関数のはじめから，回避コードの基本ブロックまでに Windows API を呼び出していないものを取り除くことで，誤検知の削減に取り組む．この手法を，カットオフ処理と呼ぶことにする．直接システムコールを呼び出す方法もあるが，Windows のバージョンごとにシステムコールの番号が異なることや，そもそも Windows ではそのような使い方が意図されていないことを踏まえれば，そのような検体は非常に疑わしく，検知されやすいという観点でそれほど使われていないものと考える．

アルゴリズム 3: 回避コード取得アルゴリズム

Input: 基本ブロック X
CFG

Output: 回避コードの疑いがあるかどうか

```
1  N ⟵ 基本ブロック X の最後の命令
2  if N が間接ジャンプ命令 and 到達先が不明 then
3      if N が CALL 命令 then
4          S ⟵ N の次の命令
5          X ⟵ S が所属する基本ブロック
6      if N が JMP 命令 then
7          return False
8  for i ∈ X の後続 do
9      if i は解析環境で実行された基本ブロックへ到達 then
10         return True
11     i を入力にアルゴリズムの再実行
```

ここで，抽出された回避コード内には，Windows API が含まれていない．なぜならば，回避コードに含まれる分岐命令の一つは，無害関数へ到達し，基本ブロックの定義より，同一基本ブロックに分岐命令を 2 つ入れることができないためである[1]．

したがって，関数内で Windows API が呼び出されているものとすれば，回避コードの先祖ノードが示す基本ブロックに含まれているはずである．

ここで，先祖ノードが持つ分岐命令に，他の関数へジャンプするものが含まれている可能性がある．他の関数まで考慮に入れると，ネスト深さによっては莫大な時間を要するため，アルゴリズムを適用するネスト深さに上限 N のパラメータを導入することにする．N 以上のネスト深さを持つ関数に対しては，Windows API は含まれていないものとして進める．

例えば，図 3 のフローチャートにおいて，関数 A の灰色に相当するノードが回避コードとして抽出された場合を考える．フローチャート中における二重丸は，

[1] CALL 命令を基本ブロックの内部に入れる場合もあるが，本論文は CALL 命令を分岐命令の一つと考えている．

Windows API の呼び出しを示している．そして，破線の矢印は，ステップオーバーした場合の向きを示している．

$N = 1$ の場合，関数 B の内部は調べないため，関数 A の破線のようにエッジが接続されているものとみなされ，回避コードの先祖ノードは start ノードのみとなる．したがって，Windows API は含まれていないものとみなされる．次に，$N = 2$ の場合，関数 B まで調査する．関数 C の内部は調べないため，関数 B の破線のようにエッジが接続されているものとみなされる．この場合は，Windows API は含まれるものとみなされる．$N = 3$ の場合は，関数 C まで参照する．関数 B のときと同様に，Windows API は含まれるものとみなされる．

以上をもとに，与えられた始点から終点までに，Windows API の呼び出しが含まれているかどうかを判定するアルゴリズムを，アルゴリズム 4 に示す．このアルゴリズムは，グラフ G と基本ブロック bb を与える必要がある．G は，bb が所属する関数の基本ブロックで構成される CFG，bb は回避コードである．

なお，今回の擬似検体の場合は，回避関数と悪意関数で同じ関数を呼び出しているため，この改良を加えたとしても，悪意関数の誤検知を除外することはできない．

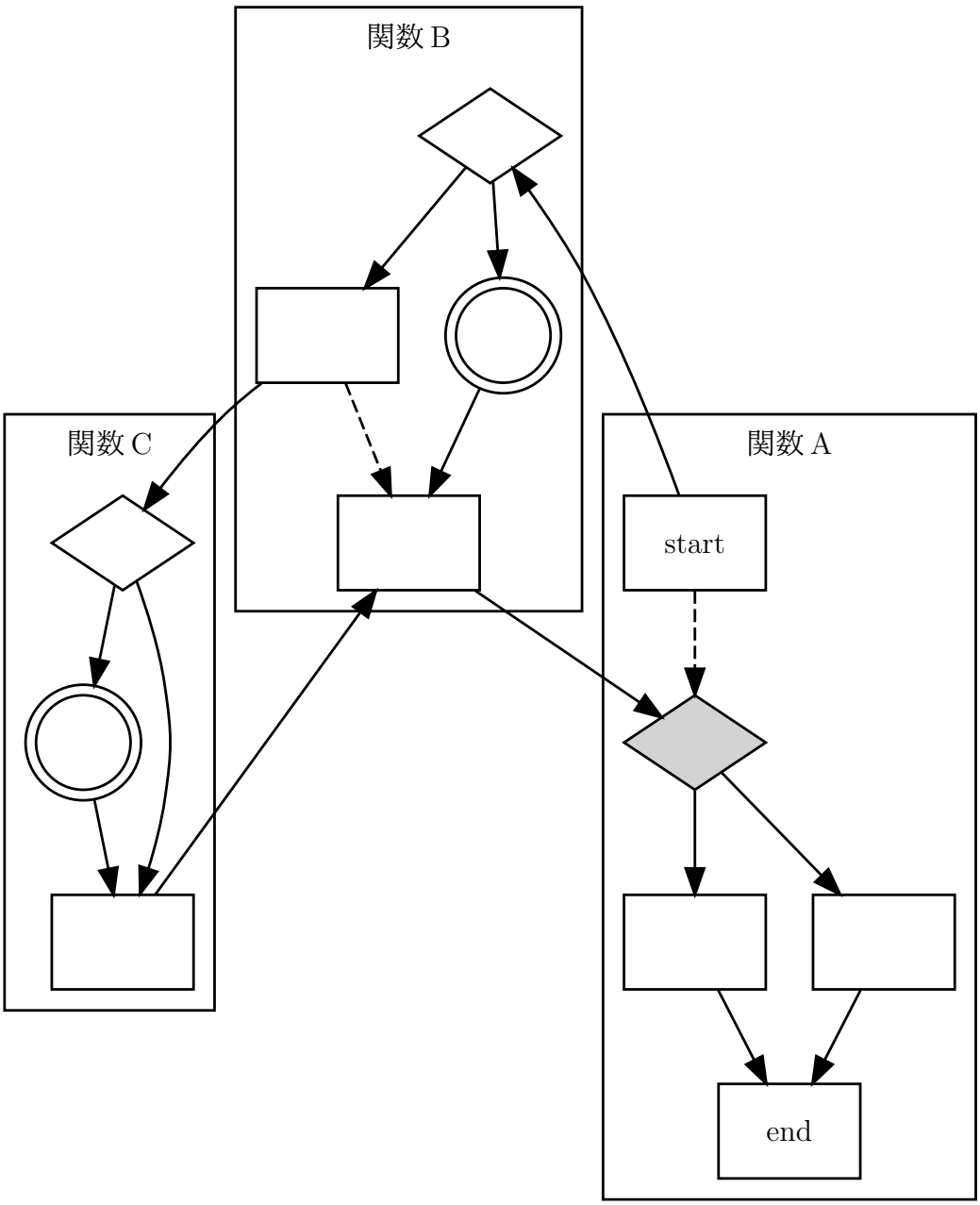

図 3　$N = 3$ の関数例．二重丸は Windows API を表す．start から灰色のノードまで到達する場合，$N = 1$ であれば関数 B と関数 C の中身を見ないため，見かけ上は関数 A の破線のようになり，Windows API は含まれていない関数とみなされる．$N \geq 2$ の場合，Windows API は含まれているものとみなされる．

アルゴリズム 4: Windows API の呼び出し判断アルゴリズム

Input: CFG G
基本ブロック bb
ネスト深さ N

Output: bb に Windows API が含まれるかどうか

```
1  P ⟵ G における bb の先祖ノード
2  for i ∈ P do
3      if i が Windows API を呼び出す分岐命令を含む then
4          return True
5      else if i は分岐命令を含み現在のネスト深さが N 以下 then
6          Q ⟵ i の後続ノード
7          for j ∈ Q do
8              _bb ⟵ j
9              return 再帰呼出し (G, _bb, N + 1)
10     return False
```

4.3 実装フレームワーク

本節では，プロトタイプ作成にあたり用いたフレームワークについて説明する．

4.3.1 DynamoRIO

DynamoRIO [19] とは，動的バイナリ計装（Dynamic Binary Instrumentation; DBI）フレームワークの一つであり，検体実行中に動的なコード挿入が可能な API を提供している．この API を利用したプログラムは，DynamoRIO クライアントと呼ばれ [20]，DynamoRIO からパラメータとして呼び出して利用する．

プロトタイプでは，4.1 節で述べた情報を取得するプログラムを，DynamoRIO クライアントとして実装した．以下に，DynamoRIO を用いた情報取得の実装について説明する．

検体がコードセグメント中で実行したアドレス

DynamoRIO 上で検体を起動すると，検体の基本ブロックがコードキャッシュへコピーされる．そのコピー時に，基本ブロックの内容を書き換えることができる．そこで，各基本ブロックに，アルゴリズム 1 のコードを追加した．

条件分岐命令の実行可否とそのアドレス

コードキャッシュへコピーされる際に，その基本ブロックを対象にアルゴリズム 2 のコードを追加した．

解析環境にマッピングされている検体モジュールの基底アドレス

検体モジュールがマッピングされているアドレス範囲は，検体起動時に取得するようにした．

4.3.2 Angr

Angr [5] とは，バイナリ解析フレームワークの一つであり，既存研究の様々なバイナリ解析技術が実装されている．この中には，間接ジャンプの解決をできる限り行って CFG を作成する機能も含まれている．したがって，CFG 作成のフレームワークとして Angr を用いることにした．

4.3.3 Radare2

Radare2 [21] とは，リバースエンジニアリングを行うフレームワークの一つであり，r2pipe を用いることで，他のプログラミング言語から呼び出すことができる．

今回の擬似検体は，評価のためにデバッグシンボル付きでビルドしたため，そのシンボルを検出するために用いた．また，Windows API が呼び出されているかどうかの判断にも用いた．同様のことは Angr でも行うことができるが，実装の都合上 Radare2 を利用した．

5. 評価

本章では，擬似検体を用いて，その有効性の評価を行う．提案手法により導出された基本ブロックのアドレスが真の回避コードであるか否かを調べるため，擬似検体はすべてデバッグシンボル付きでビルドした．

5.1 評価に用いる擬似検体

想定モデルに沿った疑似検体として，コード 1 に記したソースコードを用いた．コード中における悪意関数と無害関数の両方は，`printf` 関数でメッセージを標準出力するようにした．また，4 行目の条件式は，論文 [14] を参考に，解析環境上ではユーザ活動が欠如する条件を表現した．条件式のしきい値は，本来機械学習をもとに決定されるが，本項では構造に注目しているため，適当な数値を代入している．

ソースコードのコンパイルにあたり，同一のソースコードであったとしても，コンパイラの最適化によって，出力されるバイナリコードの構造に変化を与える可能性がある．また，静的リンクを用いることで，ビルド時に必要なライブラリのコードがコピーされるため，関数が増加する．最適化による構造の変化や，リンク方法による関数の増加が結果にどのような影響を与えるかを調べるため，以下に示す条件で，同一コードの疑似検体を作成した．なお，明記していないオプションはすべてデフォルトである．Microsoft Build Engine でビルドされた検体は，Microsoft Visual C++ 2019 において，ソリューションの構成として Release にした上でビルドしたものである．

Microsoft Build Engine 16.3.2.50909 でビルドされた検体

- 動的にリンクした検体
  - 通常の最適化（`/O2` オプション）でビルドした検体
  - 最適化無し（`/Od` オプション）でビルドした検体
- 静的にリンクした検体

表 1 混同行列

		抽出した結果	
		回避コード	回避コード以外
実際の結果	回避コード	TP	FN
	回避コード以外	FP	(TN)

- 通常の最適化（/O2 オプション）でビルドした検体
- 最適化無し（/Od オプション）でビルドした検体

MinGW 版の G++ 9.2.0 でビルドされた検体

- 動的にリンクした検体
- 強い最適化でビルドした検体[2]

5.2 実験環境

実験する解析環境は，VirtualBox 上で起動した 64bit 版の Windows 10 である．この環境は，ゴミ箱の中身が空であるため，コード 1 では，5 行目の `numBin < 10` が真となり，6 行目の無害関数が実行されて終了する．つまり，残りの条件式は評価されない．

5.3 評価指標

提案手法の評価指標として，表 1 に示す混同行列を用いる．それぞれの単語の意味は次のとおりである．

[2]標準規格を無視した最適化を行う `-Ofast -march=native` オプションでビルドする．Microsoft Build Engine にはこの機能に相当するオプションはない．

コード 1　本研究で想定する回避型マルウェアの例

```
1  int numBin = ゴミ箱のファイル数 ();
2  int numDoc = マイドキュメントのファイル数 ();
3  int numProc = プロセス数 ();
4  int numDesktop = デスクトップのファイル数 ();
5  if (numBin < 10) {
6    evasive(); // 無害関数
7    exit(0);
8  } else {
9    if (numDoc < 10) {
10     if (numDesktop < 10) {
11       evasive(); // 無害関数
12       exit(0);
13     } else {
14       if (numProc < 10) {
15         evasive(); // 無害関数
16         exit(0);
17       }
18     }
19   } else {
20     if (numProc < 10) {
21       evasive(); // 無害関数
22       exit(0);
23     }
24   }
25 }
26 malicious(); // 悪意関数
```

True Positive (TP) 正しく回避コードを抽出できた基本ブロックの数

False Positive (FP) 誤って回避コードを抽出した基本ブロックの数

True Negative (TN) 抽出すべきでない回避コードを抽出しなかった基本ブロックの数．本手法は回避コードの抽出が目的であるため，True Negative は重要ではない．したがって，評価では用いない．

False Negative (FN) 回避コードを抽出できなかった基本ブロックの数

True Positive Rate (TPR) 検出されるべき回避コードのうち，正しく検出された割合．式 2 で算出される．

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2}$$

Positive Predictive Value (PPV) 回避コードと抽出されたアドレスのうち，本物の回避コードが含まれていた割合．式 3 で算出される．

$$\mathrm{PPV} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{3}$$

5.4 実験結果

動的･静的リンク (`/O2`) の場合における CFG を図 4 に示す．図 4 では，回避コードに分類されている 5 つのノードが真の回避コードである．numProc < 10 のノードが 2 つ存在しているのは，numDoc < 10 の False が指す先と numDesktop < 10 の False が指す先が異なっていたためである．動的・静的リンク (`/Od`) と，動的リンク (G++) の場合における CFG を図 5 に示す．図 5 では，回避コードに分類されている 5 つのノードが真の回避コードである．無害関数を呼び出すノードが多数存在しているのは，それぞれの回避コードにおいて，無害関数を呼び出すノードが独立しているためである．強い最適化（G++）の場合における CFG を図 6 に示す．図 6 では，回避コードに分類されている 4 つのノードが真の回避コードである．図 4，図 5，図 6 において，灰色のノードと破線のエッジは，解析環境上で検体が実行した流れを示している．コンパイラによって生成されるコードは，

条件式が正規化されている場合があり，必ずしもコード 1 に記した条件式と一致しない．しかし，正しい動作を行うコードが生成されるため，それぞれの CFG に記されている条件式は，コード 1 に準拠した．

以上をもとに，提案手法を用いて真の回避コードが正しく分類できたかどうかを評価する．カットオフ処理を行わない状態での結果を表 2 に示す．TPR の相加平均は 58.3%であり，半分以上の回避コードが抽出できている．一方で，PPV の相加平均は 3.78%であった．カットオフ処理には，パラメータとしてネスト深さ N が必要である．表 3 に $N = 1$ の場合における結果を示す．コンパイラによる最適化が行われた検体では，PPV が向上していることが分かる．一方で，最適化が行われていない検体[3]では，PPV と TPR が 0%である．したがって，抽出された回避コードには，真の回避コードが一つも含まれていないことがいえる．しかし，TPR については，$N = 2$ における結果を示した表 4 と，カットオフ処理を行わない状態である表 2 が等しい．また，PPV の相加平均も 10.2 %に向上していることがわかる．しかし，コンパイラによる最適化が行われた検体に対しては，$N = 1$ の場合と比べると，PPV が低下している．表 5 に $N = 3$ における結果を示す．すべての検体において，TPR に変化は現れていない一方で，PPV は，$N = 2$ の場合と比較して低下している．カットオフ処理は，提案手法における回避コード抽出の結果から，TP の可能性が高い基本ブロックをフィルタリングする手法である．そのため，カットオフ処理を行わない状態と比較して，TP を取りこぼしたことによる TPR の低下はあり得るが，増加することはない．また，ネスト深さ N は，値が大きくなるほど，TP の可能性が高い基本ブロックが増加する．$N = 2$ の時点でカットオフ処理前の TPR を満たしたため，$N = 3$ 以降が優れた結果になることはない．したがって，ネスト深さ $N = 2$ でカットオフ処理を行うことで，PPV の相加平均が 10.2%に向上することが確認できた．表 6 に，カットオフ処理を行わない状態における条件分岐のパターン数を示す．検体のコードセグメントに含まれる条件分岐から，回避コードの疑いがある条件分岐を絞り込むことによって，最大 86.18%削減できた．表 7 に，$N = 1$ における条件分岐のパターン数を示す．$N = 1$ のとき，動的リンク (`/Od`) では，98.52 %の

[3]動的リンク（G++）は最適化が行われていない検体に含む．G++は，最適化オプションを明示的に指定しない場合，最適化を行わないためである．

削減率を達成しているが，TPR が 0 %であるため有用ではない．一方で，TPR がカットオフ前の状態の検体も存在し，それに絞り込むと，最大 97.02 %の削減率を達成している．表 8 に，$N = 2$ における条件分岐のパターン数を示す．最大 95.46%削減できることが確認でき，$N = 1$ の場合ほどではないものの，高い削減率を達成できている．特に，$N = 2$ の場合は，すべての検体で TPR を維持しているため，この検体においては，$N = 1$ が一番有益な結果であるといえる．表 9 に，$N = 3$ における条件分岐のパターン数を示す．削減率は，$N = 2$ と同等か低くなっていることがわかる．

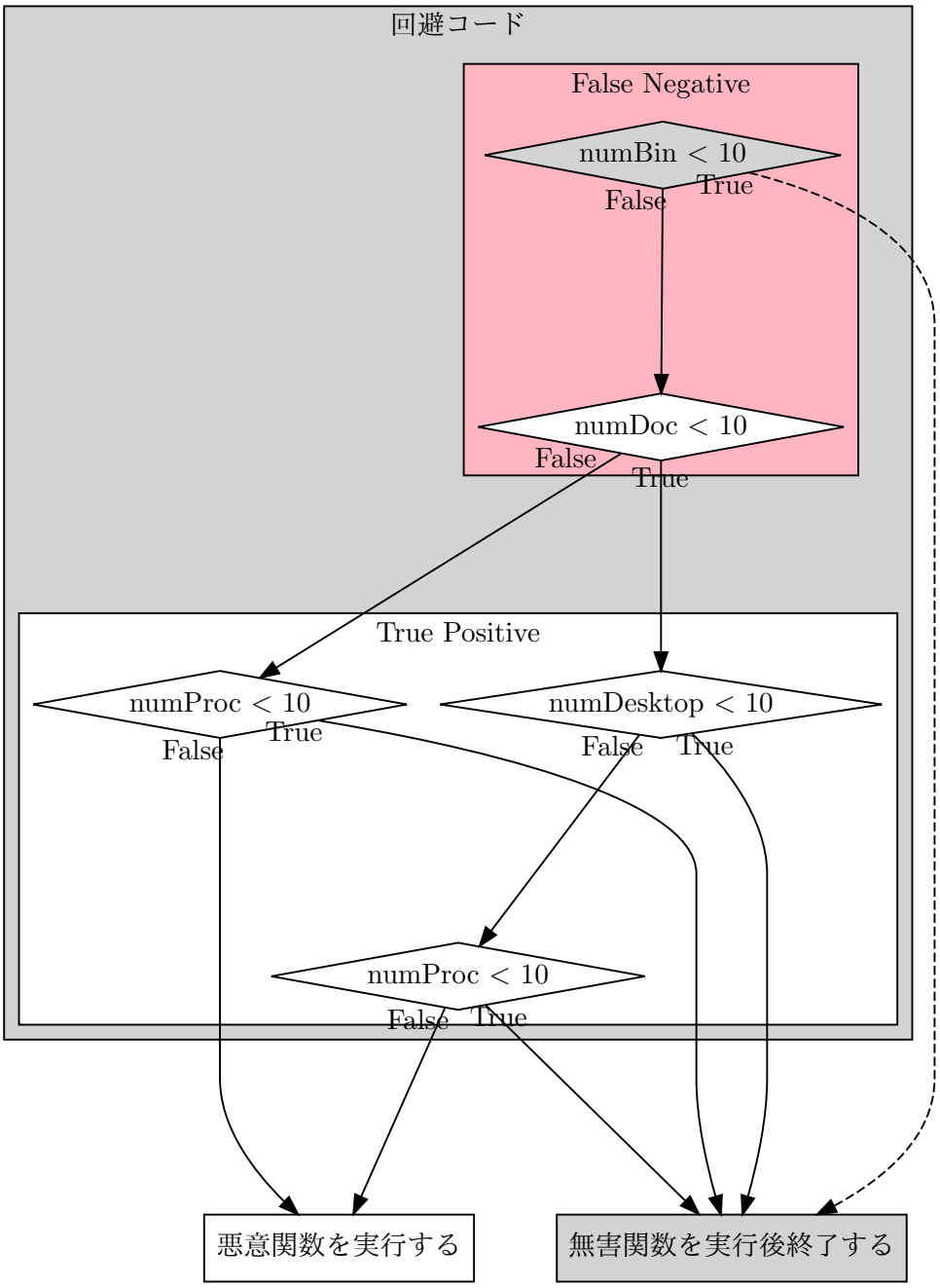

図 4 動的・静的リンク (/O2) の場合における CFG

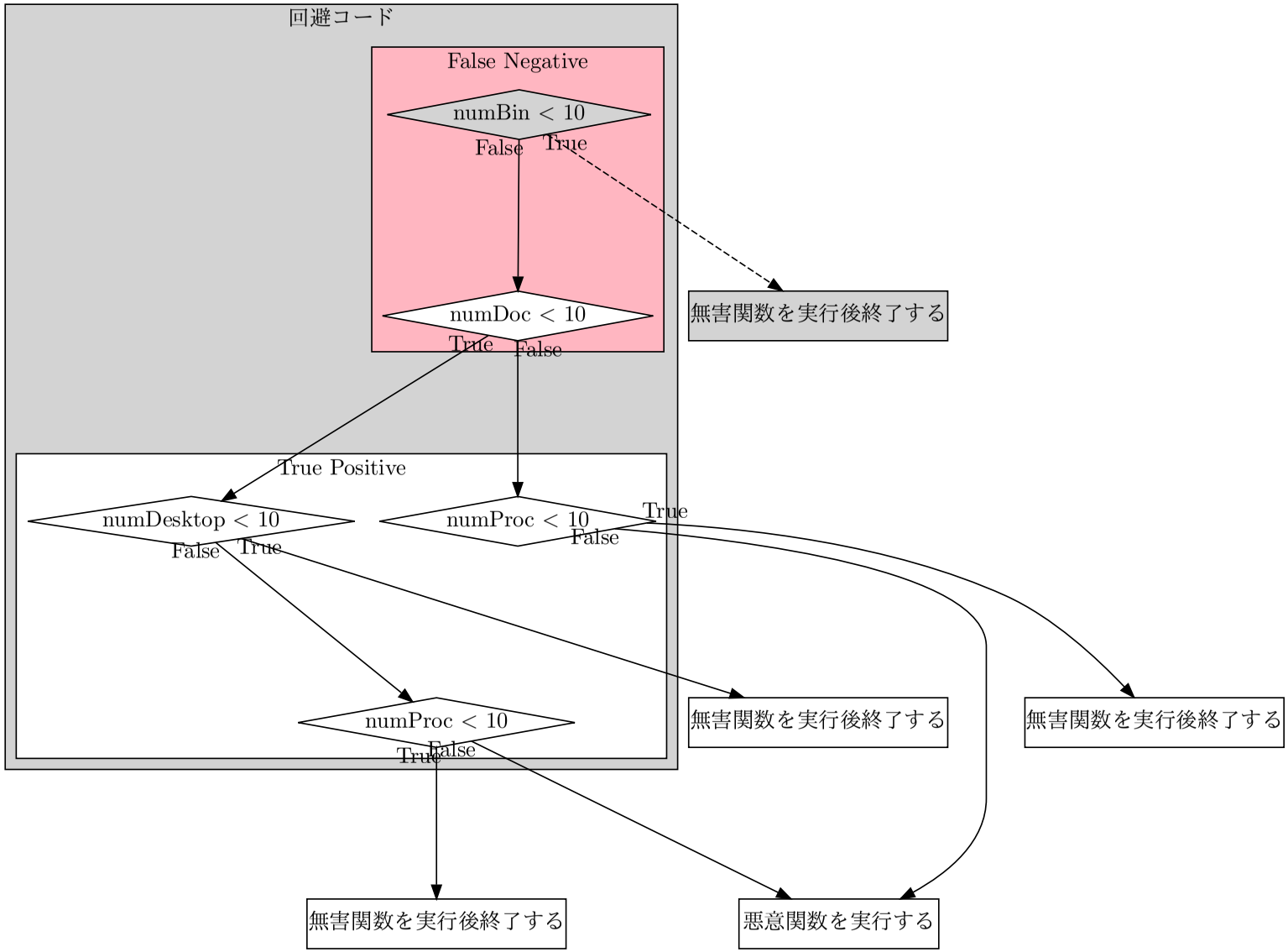

図 5 動的・静的リンク (/Od) と，動的リンク (G++) の場合における CFG

表 2 カットオフ処理を行わない状態における回避コードの抽出結果

検体の種類	TP	FN	FP	TPR	PPV
動的リンク (`/O2`)	3	2	29	60.0%	9.4%
動的リンク (`/Od`)	3	2	32	60.0%	8.6%
静的リンク (`/O2`)	3	2	649	60.0%	0.5%
静的リンク (`/Od`)	3	2	656	60.0%	0.5%
動的リンク (G++)	3	2	125	60.0%	2.3%
強い最適化 (G++)	2	2	132	50.0%	1.5%

表 3 回避コードの抽出結果 (ネスト深さ $N = 1$)

検体の種類	TP	FN	FP	TPR	PPV
動的リンク (`/O2`)	3	2	5	60.0%	37.5%
動的リンク (`/Od`)	0	5	2	0.0%	0.0%
静的リンク (`/O2`)	3	2	120	60.0%	2.4%
静的リンク (`/Od`)	0	5	116	0.0%	0.0%
動的リンク (G++)	0	5	17	0.0%	0.0%
強い最適化 (G++)	2	2	26	50.0%	7.1%

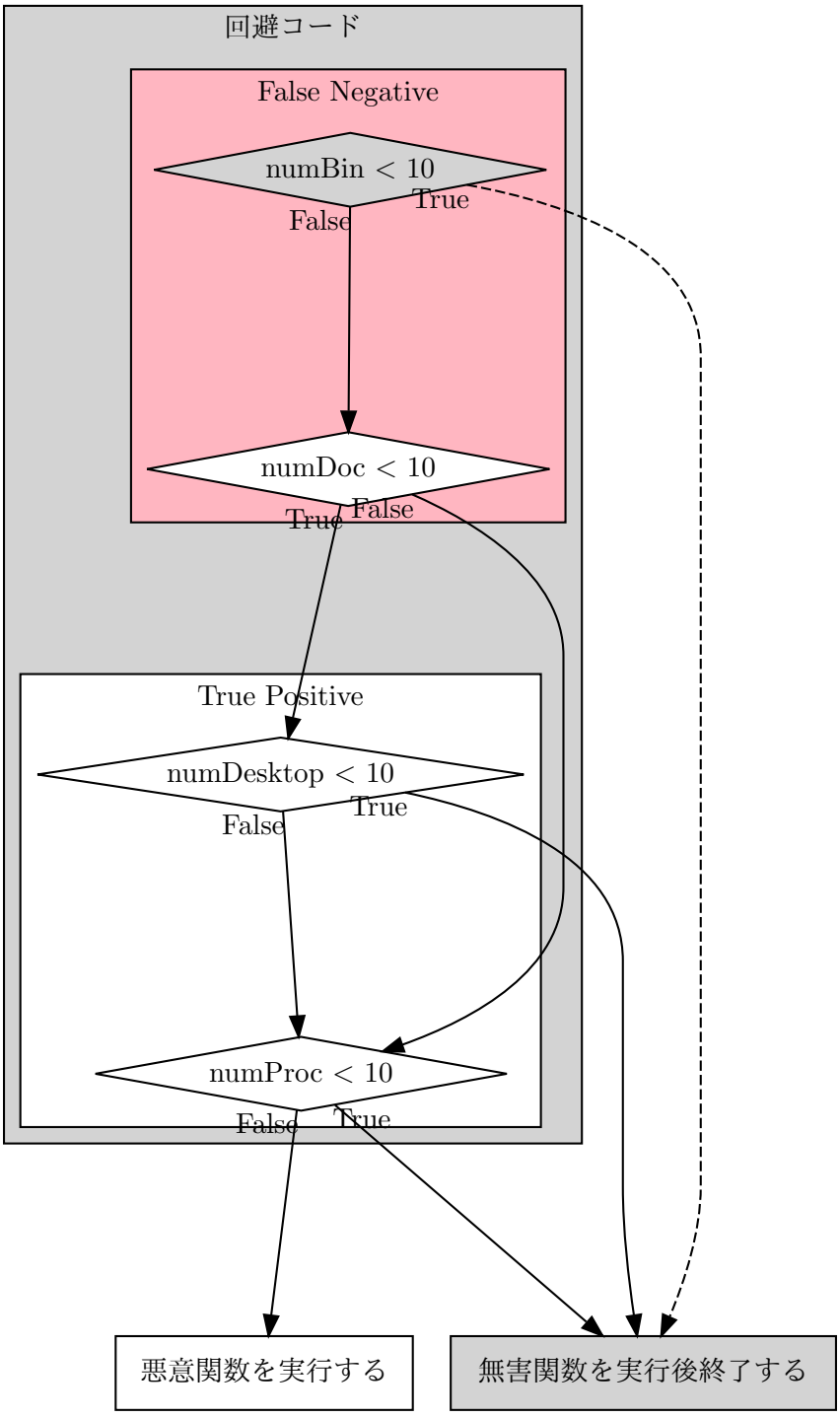

図 6 強い最適化（G++）の場合における CFG

表 4 回避コードの抽出結果 (ネスト深さ $N = 2$)

検体の種類	TP	FN	FP	TPR	PPV
動的リンク (`/O2`)	3	2	9	60.0%	25.0%
動的リンク (`/Od`)	3	2	10	60.0%	23.1%
静的リンク (`/O2`)	3	2	344	60.0%	0.9%
静的リンク (`/Od`)	3	2	348	60.0%	0.9%
動的リンク (G++)	3	2	39	60.0%	7.1%
強い最適化 (G++)	2	2	48	50.0%	4.0%

表 5 回避コードの抽出結果 (ネスト深さ $N = 3$)

検体の種類	TP	FN	FP	TPR	PPV
動的リンク (`/O2`)	3	2	9	60.0%	25.0%
動的リンク (`/Od`)	3	2	10	60.0%	23.1%
静的リンク (`/O2`)	3	2	371	60.0%	0.8%
静的リンク (`/Od`)	3	2	418	60.0%	0.7%
動的リンク (G++)	3	2	48	60.0%	5.9%
強い最適化 (G++)	2	2	69	50.0%	2.8%

表 6 カットオフ処理を行わない状態における条件分岐のパターン数

検体の種類	全条件分岐数	抽出数	削減率
動的リンク (/O2)	145	32	77.93%
動的リンク (/Od)	135	35	74.07%
静的リンク (/O2)	2935	652	77.79%
静的リンク (/Od)	2925	659	77.47%
動的リンク (G++)	926	128	86.18%
強い最適化 (G++)	939	134	85.73%

表 7 条件分岐のパターン数 (ネスト深さ $N = 1$)

検体の種類	全条件分岐数	抽出数	削減率
動的リンク (/O2)	145	8	94.48%
動的リンク (/Od)	135	2	98.52%
静的リンク (/O2)	2935	123	95.81%
静的リンク (/Od)	2925	116	96.03%
動的リンク (G++)	926	17	98.16%
強い最適化 (G++)	939	28	97.02%

表 8 条件分岐のパターン数 (ネスト深さ $N = 2$)

検体の種類	全条件分岐数	抽出数	削減率
動的リンク (`/O2`)	145	12	91.72%
動的リンク (`/Od`)	135	13	90.37%
静的リンク (`/O2`)	2935	347	88.18%
静的リンク (`/Od`)	2925	351	88.00%
動的リンク (G++)	926	42	95.46%
強い最適化 (G++)	939	50	94.68%

表 9 条件分岐のパターン数 (ネスト深さ $N = 3$)

検体の種類	全条件分岐数	抽出数	削減率
動的リンク (`/O2`)	145	12	91.72%
動的リンク (`/Od`)	135	13	90.37%
静的リンク (`/O2`)	2935	374	87.26%
静的リンク (`/Od`)	2925	421	85.61%
動的リンク (G++)	926	51	94.49%
強い最適化 (G++)	939	71	92.44%

6. 考察

本章では，実験結果をもとに提案手法の制約点や課題について議論する．

前節で述べた実験結果より，カットオフ処理を行わない状態における TPR の相加平均は 58.3 %であり，すべての回避コードを抽出することができなかった．図 4 より，FN に分類されたノードに注目すると，以下の特徴が見られる．

(1) 解析環境上で実行した回避コード

(2) 解析環境でまだ実行されていない回避コードへ分岐する回避コード

本提案手法は，解析環境で実行された基本ブロックすべてを無害関数とみなしているため，解析環境上で実行した回避コードも無害関数として扱われてしまう．そして，回避コードの検出手法が，無害関数に依存する手法であるため，解析環境上で実行されたことにより，無害関数に誤って分類された回避コードは検出できないという欠点がある．無害関数に依存する例として，アルゴリズム 3 は，入力された基本ブロックが，無害関数へ到達するか否かを以って回避コードを判断することが挙げられる．したがって，解析環境上でまだ実行されていない回避コードへ分岐する回避コードは，無害関数の情報だけでは特定できない．しかし，図 4 ， 5 ， 6 より，解析環境上で実行される回避コードや，解析環境でまだ実行されていない回避コードへ分岐する回避コードは，回避コード全体のはじめの部分であり，TP の先祖ノードに存在している．したがって，手動解析時に，先祖ノードも含めて関数全体を調査することで，FN の回避コードも解析できることが期待される．ここで，図 5 の場合，無害関数を実行する基本ブロックがそれぞれ独立しており，解析環境で実行された基本ブロックへ到達しないように見える．しかし，無害関数を実行する基本ブロックは，無害関数本体とは別であり，無害関数本体を呼び出すだけの基本ブロックである．したがって，無害関数本体は解析環境上で実行されているため，アルゴリズム 3 によって検知可能である．

カットオフ処理におけるそれぞれの N に対する TPR の変化を図 7 に，PPV の変化を図 8 に示す．図 7 より，TPR は，表 2 に示したカットオフ処理を行わない状態に収束する．また，PPV は，最適化を行った検体では，$N = 1$ のときを

最大として，低下していく様子が見られる．一方で，最適化されていない検体では，$N = 1$ のとき TPR が 0%となるため，PPV もそれに応じて 0 %になっている．これは，カットオフ処理によるフィルタリングが強すぎるからである．カットオフ処理は，無害関数への到達可能性による回避コード検知手法によって抽出された回避コードから，TP の疑いが強い基本ブロックを取り出す処理である．この処理は，関数の内部を 1 層探索し，Windows API の呼び出しが含まれている回避コードのみを抽出する $N = 1$ が一番強い．最適化が行われた検体の場合では，コード 1 における，ゴミ箱のファイル数，マイドキュメントのファイル数等を取得する関数の一部またはすべてが第 1 層の関数に展開されていた．したがって，Windows API の呼び出しが第 1 層の関数に出現するため，$N = 1$ の場合も TPR は維持できていた．しかし，最適化が行われていない検体の場合，できる限りコード 1 に近い構造でバイナリが出力されるため，ゴミ箱のファイル数，マイドキュメントのファイル数等を取得する関数の内部に Windows API が含まれている．したがって，第 2 層以降の関数まで探索する必要があるため，$N = 1$ では，TPR が 0%になった．$N = 2$ は，関数の内部を 2 層探索するため，最適化が行われていない検体でも TPR が回復する．一方で，探索する関数が増えるため，Windows API を発見する可能性も高くなり，$N = 1$ で回避コードを発見できていた検体に関しては，$N = 1$ と比較して PPV が小さくなる．

今回，私達が提案した手法は，論文 [15] の手法を補助できる可能性がある．2.3 節で述べたように，論文 [15] の手法は，システムコールの戻り値が用いられている条件分岐すべてに動的解析を行う．一方で，本提案手法は，検体のコードセグメントに含まれるすべての条件分岐から，解析環境の判断に用いられている可能性がある条件分岐を抽出する．表 6，表 8 では，探索空間を最大 86.15 %，カットオフ処理込みでは，有益な結果の中では $N = 2$ のとき，最大 95.46 %削減できることを示した．このことから，本提案手法により抽出された条件分岐命令へ到達する命令を対象に，論文 [15] の手法を適用することで，効率的に悪意関数の動的解析ができることが期待される．

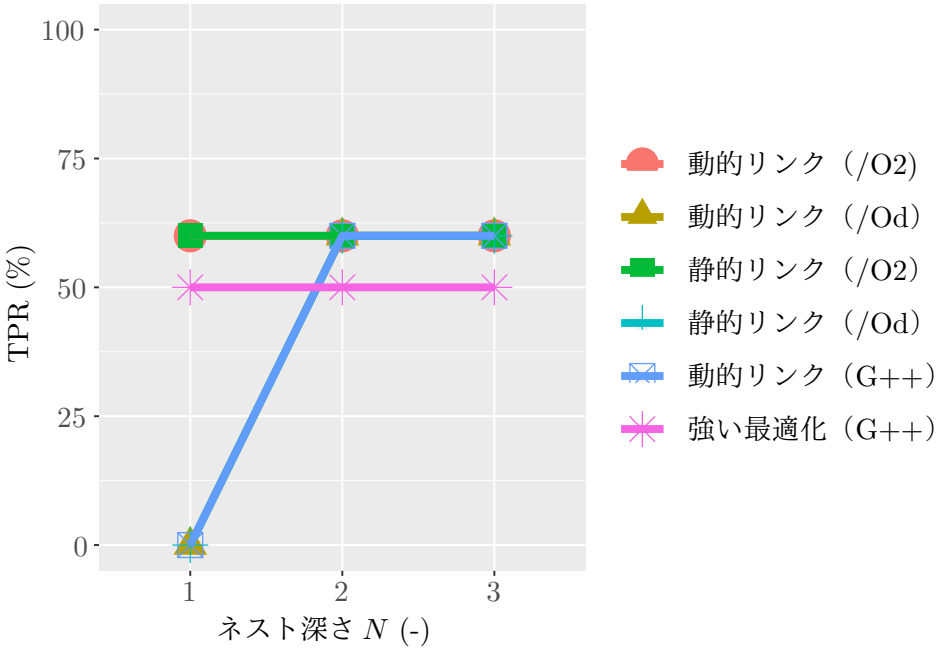

図 7　$N = 1$ から $N = 3$ までにおける TPR の変化

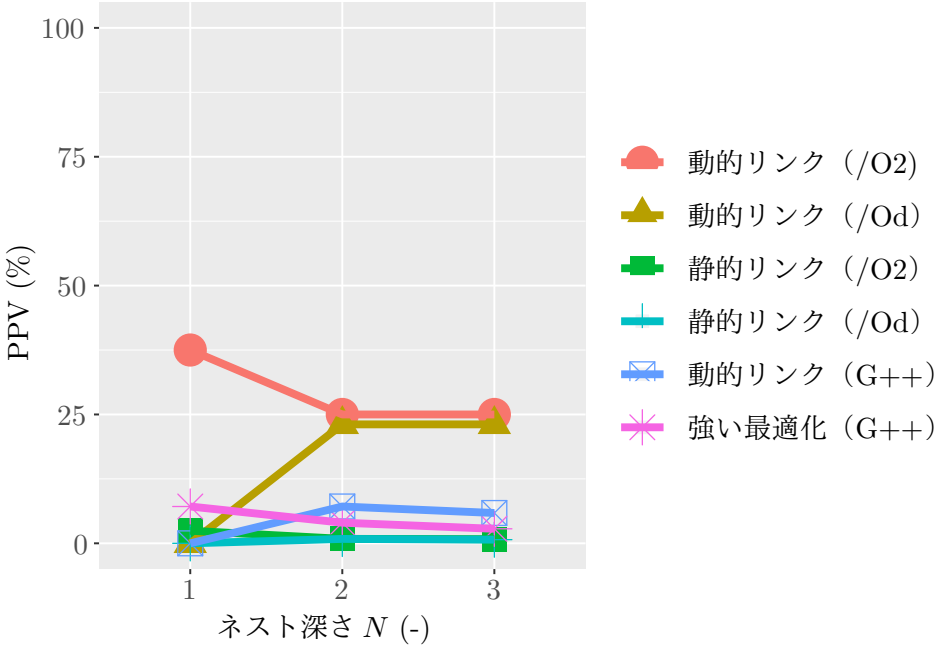

図 8　$N = 1$ から $N = 3$ までにおける PPV の変化

6.1 提案手法の限界

本提案手法では，解析環境で実行されなかった基本ブロックから回避コードが抽出される．そのため，解析環境上ですべての回避コードが実行された場合，回避コードが抽出できないものと見られる．したがって，解析環境で実行する回避コードは少ない方が望ましい．有名な解析環境を用いることで，回避コードの最初付近でプログラムを終了できることが期待される．また，解析環境上から無害関数の情報を取得して回避コードを抽出するアプローチを用いているため，無害関数が複数存在しており，回避コードごとに異なる無害関数が用いられた場合も，回避コードを正しく抽出できないものと見られる．

本研究における回避型マルウェアの想定モデルは，if-else 文によって実行経路が無害関数と有害関数に分岐することを前提としていた．しかし，if-else 文を用いずに実行経路を分岐することも可能である．本研究における回避型マルウェアの想定モデルは，回避関数，無害関数ともに CFG によって到達できる構造である．一方で，それ以外の CFG の構造として，以下の 2 つが考えられる．

- 回避関数が CFG によって到達できない場合
- 無害関数が CFG によって到達できない場合

それぞれの構造における具体的な例については，Appendix の A と B に記す．以上の構造を持つ検体の場合，回避コードが抽出できないものと見られる．

7. おわりに

本稿では，はじめに回避型マルウェアの脅威と既存の対策手法を述べた．既存の対策手法は，多くが仮想環境を検知する回避型マルウェアに対する手法である一方，解析環境の情報を収集する回避型マルウェアに対しては効果がないことも述べた．提案手法では，未知の環境属性を用いた回避型マルウェアに対応するために，回避型マルウェアの構造に着目し，動的解析・静的解析を組み合わせた回避コードの抽出手法を提案した．動的解析では，検体がコードセグメント中で実行したアドレスと条件分岐命令の実行可否とそのアドレス，解析環境にマッピングされている検体モジュールの基底アドレスを収集し，静的解析では，無害関数への到達可能性による回避コード検知手法と Windows API の呼び出し有無によるカットオフ処理手法を用いることで，回避コードを抽出する手法を述べた．本提案手法は，解析環境で悪意のある挙動を再現する必要がないという点で優位性がある．擬似検体を用いて回避コードを抽出したところ，TPR は 58.3%であり，PPV は 3.78%であった．しかし，カットオフ処理手法を用いることで，PPV を 10.2%まで向上できることを示した．また，検体のコードセグメントに含まれる条件分岐から，回避コードの疑いがある条件分岐を絞り込むことによって，最大 86.18%，カットオフ処理によって，有益な結果の中では，$N = 2$ のとき，最大 95.46%削減できることも示した．

一方で，無害関数の収集時に回避関数が混合し，取り除くことができないため，TPR や PPV 向上の妨げになっている．したがって，現時点では実用レベルまで達していない．そのため，無害関数の収集方法を見直すことで，回避コードの検知率を向上させることが今後の課題である．

謝辞

本研究の取り組みにあたり，ご指導を頂きました本学の門林雄基教授，林優一教授，中島康彦教授，妙中雄三准教授に深く感謝申し上げます．本研究を進めるにあたり，SecHack を通して，国立研究開発法人 情報通信研究機構の笠間貴弘氏，神薗雅紀氏，衛藤将史氏，横山輝明氏にもお世話になりましたことを，心より感謝申し上げます．また，サイバーレジリエンス構成学研究室の樫原茂助教，宮本大輔客員准教授，Doudou Fall 助教にも多大なるご協力を頂きましたことを，ここに厚く御礼申し上げます．さらに，サイバーレジリエンス構成学研究室の秘書，学生の皆様にも大変お世話になりました．この場を借りて感謝申し上げます．

最後に，これまで私を支援してくださった両親と家族に深い感謝の意を表します．

参考文献

[1] 総務省. 国民のための情報セキュリティサイト. https://www.soumu.go.jp/main_sosiki/joho_tsusin/security/basic/risk/02-2.html.

[2] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, Vol. 52, No. 5, September 2019.

[3] A. Jadhav, D. Vidyarthi, and Hemavathy M. Evolution of evasive malwares: A survey. In *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, pp. 641–646, March 2016.

[4] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware Analysis and Classification: A Survey. *Journal of Information Security*, Vol. 05, No. 02, pp. 56–64, 2014.

[5] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[6] Alexei Bulazel and Bülent Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ROOTS, pp. 2:1–2:21, New York, NY, USA, 2017. ACM.

[7] VMware. https://www.vmware.com/.

[8] Oracle VM VirtualBox. https://www.virtualbox.org/.

[9] a0rtega/pafish: Pafish is a demonstration tool that employs several techniques to detect sandboxes and analysis environments in the same way as malware families do. `https://github.com/a0rtega/pafish`.

[10] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pp. 403–412, New York, NY, USA, 2011. ACM.

[11] Marcus Botacin, Paulo Lício De Geus, and André Grégio. Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Trans. Priv. Secur.*, Vol. 21, No. 1, pp. 4:1–4:30, January 2018.

[12] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 287–301, San Diego, CA, August 2014. USENIX Association.

[13] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, and Christian Rossow. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pp. 165–187, Cham, 2016. Springer International Publishing.

[14] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 1009–1024, May 2017.

[15] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pp. 231–245, May 2007.

[16] Cuckoo Sandbox - Automated Malware Analysis. `https://cuckoosandbox.org/`.

[17] 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～. Technical Report 8, 株式会社ソリトンシステムズ, 国立研究開発法人情報通信研究機構, 株式会社 FFRI, NTT セキュアプラットフォーム研究所, 日本電信電話株式会社, 東京電機大学／株式会社日立製作所, July 2019.

[18] Yoshihiro Oyama. Trends of anti-analysis operations of malwares observed in api call logs. *Journal of Computer Virology and Hacking Techniques*, Vol. 14, No. 1, pp. 69–85, February 2018.

[19] DynamoRIO Dynamic Instrumentation Tool Platform. `https://www.dynamorio.org/`.

[20] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[21] Radare2. `https://rada.re/`.