

# オブジェクトの構造的特徴の解析に基づく メモリ位置特定手法

碓井 利宣<sup>1,a)</sup> 大月 勇人<sup>1</sup> 川古谷 裕平<sup>1</sup> 塩治 榮太朗<sup>1</sup> 岩村 誠<sup>1</sup>

**概要：**オブジェクトのメモリ上での位置の特定は、エクスプロイト開発やメモリフォレンジックスなど、多様な応用先で求められている。しかし、アドレス空間の配置がランダム化される現在では、シンボル情報なしでの実現は容易でない。メモリ上の位置特定の技術には、特徴的な値でのスキャンや、メモリ漏洩などで得た既知のポインタを基点にしたオフセット計算がある。しかし、特徴的な値や基点のポインタやオフセットが得られない場合、特定ができない問題が存在する。

この問題を解決するため、本論文では、従来の特徴的な値や基点のポインタの代わりに、オブジェクトの特徴的な構造に新たに着目し、それをヒントにしたメモリスキャンで位置を特定できることを明らかにする。その上で、実行時のメモリアクセスからオブジェクトの構造的特徴を解明するバイナリ解析手法と、それを基にオブジェクトの構造的特徴でスキャンするメモリ解析手法を提案する。複数のバイナリへの実験を通じて、提案手法が多様なオブジェクトのメモリ位置を現実的な時間内に特定できることを示した。

**キーワード：**メモリ解析、バイナリ解析、オブジェクト位置特定、構造的特徴、動的解析

## Identifying Memory Locations of Objects by Analyzing Structural Characteristics in Binaries

TOSHINORI USUI<sup>1,a)</sup> YUTO OTSUKI<sup>1</sup> YUHEI KAWAKOYA<sup>1</sup> EITARO SHIOJI<sup>1</sup> MAKOTO IWAMURA<sup>1</sup>

**Abstract:** Identifying the memory locations of objects is essential in various security applications including exploit development and memory forensics. However, address space layout randomization makes it difficult without symbol information. Existing approaches, which rely on scanning for distinctive values or calculating offsets from leaked pointers, have a problem of locating objects when such values, pointers, or offsets are unavailable.

To address this, we leverage structural characteristics of objects, such as member types, sizes, offsets, and pointer relations. By capturing these features, memory can be scanned to identify object locations. We propose a binary analysis technique that extracts structural characteristics from runtime memory accesses, and a memory analysis technique that locates objects based on them. Experiments on multiple binaries show that our approach can locate diverse objects within practical time.

**Keywords:** memory analysis, binary analysis, object location identification, structural characteristics, dynamic analysis

### 1. はじめに

アドレス空間配置のランダム化 (Address Space Layout

Randomization, ASLR) が普及している今日では、様々なセキュリティ技術の成立のために、変数のメモリ上の位置を特定する技術が重要である。例えばエクスプロイト開発では、ポインタなど攻撃に有用な変数のメモリ位置を特定して上書きすることで、攻撃の実現可能性を実証する。また、メモリフォレンジックスでは、解析対象が持つ変数の

<sup>1</sup> NTT 社会情報研究所  
NTT Social Informatics Laboratories  
a) tos.usui@ntt.com

メモリ位置を特定して読み取ることで、解析を進める。特に、重要な変数はオブジェクトで管理されることも多いため、オブジェクトやそのメンバを標的としたメモリ位置特定が求められる。

そのための従来技術として、主に2つのアプローチがある。1つ目は、スキャンベースのアプローチ [1] で、標的のオブジェクトに特徴的な値を埋め込んだり、元々存在する特徴的な値を用いて、スキャンによって位置を発見する。2つ目は、オフセット計算ベースのアプローチ [2, 3] で、既知のポインタを基点にして、その指し先からのオフセットを用いた相対位置の計算で発見する。しかし、一定の条件を持ったオブジェクトのメモリ位置特定が困難であるという問題を発見した。すなわち、①スキャンに適した特徴的な値を持たず、埋め込むこともできない、②基点となるポインタが得られない、またはポインタからのオフセットが不变でない、③バイナリが大きく複雑で、値やポインタ、オフセットなどの手動解析での抽出が労力の観点から現実的でない、という3つの条件である。この問題を解決できれば、たとえば従来技術では実現できない脆弱性の実証や、メモリフォレンジックスでのスキャンの安定化が実現できる。

そこで本論文では、特徴的な値の代わりに、オブジェクトの特徴的な構造を用いてメモリ上を探索する新たなアプローチを提案する。構造的特徴には、メンバの型やサイズやオフセット、パディング、オブジェクト間のポインタ参照の関係を用いる。これにより、前述の①および②の困難な状況でも、メモリ位置特定を可能にする。加えて、③に対処するために、標的のバイナリから構造的特徴を自動抽出するバイナリ解析手法を提案する。そして、抽出した構造的特徴に基づき、オブジェクトのメモリ位置を特定するメモリ解析手法を提案する。

提案手法に基づくプロトタイプを実装し、30個の実際のバイナリを収集して、バイナリ解析およびメモリ解析の精度と所用時間を評価した。その結果、バイナリ解析によって、過半数のオブジェクトのレイアウトを80%以上の精度で復元し、メンバの型も平均して90%以上の精度で推論できることを示した。また、メモリ解析によって、いずれのバイナリに対しても標的オブジェクトのメモリ位置を特定できることも示した。これらの解析はいずれも数十秒程度の現実的な時間内で実現可能なことも確認できた。

本論文の貢献は以下の通りである。

- 特徴的なオブジェクト構造をメモリ位置特定に用いる、新たなアプローチを提案した。
- バイナリとメモリの解析を組み合わせて、メモリ位置特定を自動的に実現する技術を開発した。
- 実験を通じて、提案手法が多様なバイナリに対して実際に有効であることを示した。

表1 スキャンできない値

Table 1 Unscannable Values.

値の種類	値の特異性	値の不揮発性
真理値	×	—
小さな整数値	×	—
2バイト以下の変数の値	×	—
ポインタ	●	×
流動的な変数の値	—	×

## 2. メモリ上のオブジェクトの位置の特定

### 2.1 既存のメモリ位置特定技術と問題点

#### 2.1.1 スキャンベースのアプローチ

スキャンベースのアプローチは、メモリ上を特徴的な値で検索して位置を特定する。そのため、これが成立するためには、以下の2点が要件となる。

- (1) **値が特徴的** 外部から特徴的な値を埋め込める操作可能な変数がある、または特徴的な値を元々保持している。
- (2) **値が不揮発** スキャンの時点で標的変数が当該の値を保持している。

これらの要件を満たさず、スキャンできない値を表1に示す。表中の✓は要件を満たすこと、✗は満たさないこと、●は限定的に満たすこと、—は言及なしを示す。まず、特徴的でない値として、真理値や、0, 1, -1といった小さな整数値、サイズが2バイト以下の任意の値のように、候補の値がメモリ上に頻出するものが挙げられる。次に、揮発性の高い値として、ポインタがある。ポインタはASLRによって実行ごとに値が変化するため、スキャン時に狙った値が存在しない可能性が高い。また、実行時の流動性が高い変数の値についても、同様に値が揮発しやすい。以上より、標的オブジェクトがこうしたスキャンできない値で構成されている場合、スキャンベースのアプローチは適用できないという問題がある。

#### 2.1.2 オフセット計算ベースのアプローチ

オフセット計算ベースのアプローチは、情報漏洩などで得られた基点となるポインタと既知のオフセットを用いて標的的位置を導出する。そのため、これが成立するためには、以下の2点が要件となる。

- (1) **ポインタが漏洩可能** オフセット計算の基点に適したポインタが得られる。
- (2) **オフセットが不变** オフセットが得られ、変化しない。

この観点から、変数のスコープごとの位置特定の可能性を表2にまとめる。なお、表記は表1と同様である。静的領域は、イメージベースのアドレスが一般にスキャンで取得可能であり、そこから静的領域(.dataや.bss)へのオフセットが不变であるため、要件を満たす。スタックは、スタック上の定点を指すポインタ(退避されたスタックベースなど)が漏洩可能な場合もあるが、常に得られるとは限

表 2 変数の位置特定の可能性  
Table 2 Locability of Variables.

スコープ	ポインタ漏洩可能性	オフセット不变性	位置特定
静的	✓	✓	✓
スタック	○	✓	○
ヒープ	✗	✗	✗

表 3 アプローチごとのオブジェクトのメモリ位置特定可能性  
Table 3 Locability of objects in memory for each approach.

アプローチ	特徴的な値不要?	静的	スタック	ヒープ
スキャン	✗	✓	✓	✓
オフセット計算	✓	✓	○	✗
我々のゴール	✓	✓	✓	✓

らない。一方、スタックフレーム内のオフセットは不变である。ヒープは、標的オブジェクトの存在するヒープブロックのポインタを漏洩できる場合は多くなく、基点の確保自体が難しい。さらに、実行ごとに各ヒープブロックのレイアウトが変化するため、それらの間のオフセットには不变性がない。

以上から、標的オブジェクトがスタックやヒープ上の場合、オフセット計算ベース手法の適用は難しい問題がある。

### 2.1.3 オブジェクトのメモリ位置の特定可能性とゴール

ここまで議論から、表 3 に、各手法でのオブジェクトのメモリ位置の特定可能性をまとめる。表より、特徴的な値なしに、スタックやヒープ上のオブジェクトの位置を特定可能な手法に乏しいことが分かる。そこで、我々のゴールとして、特徴的な値を要さずに、スタックやヒープ上のオブジェクトの位置特定を可能にすることを目指す。

## 2.2 解析モデル

本論文のモデルは、ローカル環境でバイナリを解析して情報を抽出する準備段階と、その情報に基づいて標的環境でメモリを解析する実行段階に分かれる。準備段階では、標的プロセスの生成元と同一のバイナリが得られ、解析環境で任意の解析技術を適用できることを前提とする。したがって、デバッグ機能などを利用した高度な解析を想定する。一方、実行段階では、任意のアドレスに対するメモリ読み込みのみを前提とする。これは、たとえばエクスプロイト開発でのメモリ漏洩の脆弱性の攻略やメモリフォレンジックスでのメモリダンプの走査などを想定している。

## 2.3 アプローチ

1つ目のキーアイデアとして、オブジェクトの構造的特徴を導入する。これは、メンバのオフセットや型、サイズ、パディングを指す。本来はスキャンに適さないポインタや真理値を、構造的特徴を踏まえることで、探索に活用する。例えば、オフセット 0x8 と 0x28 にポインタを、0x30 に真

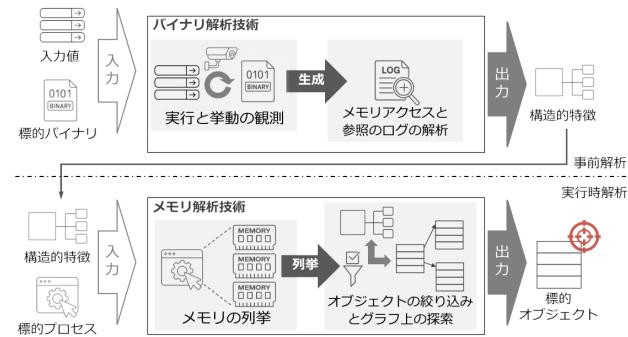


図 1 提案手法の全体像

Fig. 1 Overview of the proposed technique.

理値を、0x38 に 2 バイトのメンバを持つオブジェクトを探索する。これにより、個々のメンバの値は特徴に乏しくとも、オブジェクト全体での特徴を基に発見できる。標的オブジェクトの構造的特徴は、バイナリ解析によってあらかじめ抽出し、その特徴に基づいてメモリ上を探索して、標的と一致するメモリ領域を発見する。

2 つ目のキーアイデアとして、オブジェクトグラフを導入する。これは、複数のオブジェクト間のポインタ参照の関係をグラフ構造で表現したもので、ノードがオブジェクト、エッジがポインタ参照を表す。これにより、単一のオブジェクトでは位置特定に十分な構造的特徴がない場合でも、参照先のオブジェクトも再帰的に構造を検証することで、発見を可能にする。

## 3 提案手法

### 3.1 提案手法の全体像

提案手法は図 1 の通り、バイナリ解析とメモリ解析の 2 つのフェーズからなる。前者では標的オブジェクトの構造的特徴をあらかじめ抽出し、後者ではそれに基づいて標的オブジェクトの位置を特定する。

バイナリ解析フェーズは、以下の 2 ステップからなる。

- (1) オブジェクトの構造の解析: メモリアクセスを監視し、オブジェクトの再構築とメンバの推論をする。
- (2) オブジェクト間の参照関係の解析: オブジェクト間のポインタ参照を追跡し、グラフを構築する。

メモリ解析フェーズは、以下の 2 ステップからなる。

- (1) オブジェクトの検索: メモリを列挙し、オブジェクトの構造的特徴や参照関係により候補を絞り込む。
- (2) オブジェクトグラフ上の探索: 発見されたグラフ上をポインタ参照とオフセット計算で探索し、標的のオブジェクトやメンバのメモリ位置を特定する。

以降では、それぞれのステップについて詳細を述べる。

### 3.2 バイナリ解析による構造的特徴の抽出

#### 3.2.1 メモリアクセストレースの取得

提案手法でのメモリアクセストレースは、標的バイナリ

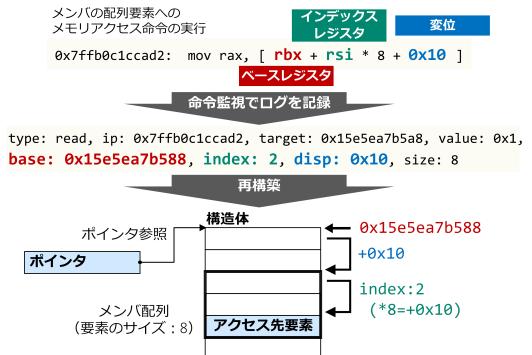


図 2 メモリアクセストレースの記録とオブジェクトの再構築

Fig. 2 Memory access trace logging and object reconstruction.

の実行時に発生したメモリの読み書きを命令レベルで記録したものである。図 2 の上部に、この記録の例を示す。図のように、実行中のメモリ操作の種類（読み書き）と命令ポインタ、対象のアドレス、読み書きした値、サイズの具体値を記録する。さらに、ベースレジスタやインデックスレジスタ、変位の値も用いられていれば記録する。これらは、命令フックでログ出力用のコードを挿入し、メモリ操作命令の実行ごとに呼び出して記録していく。

### 3.2.2 オブジェクトの再構築

記録したメモリアクセストレースを基に、図 2 の下部の例のように、オブジェクトを再構築する。構造体のベースアドレスはベースレジスタから、メンバのオフセットは変位から、メンバのサイズはメモリアクセス時のサイズからそれぞれ取得する。また、メンバが配列の場合はさらに、要素の添字をインデックスレジスタから取得する。このように、メンバへのメモリアクセスを全て走査することで、構造を再構築できる。

### 3.2.3 メンバの推論

前節で再構築されたオブジェクトの各メンバに対して、以下の手法でポインタ、真理値、パディングの推論を行う。  
ポインタの推論 アドレス幅と同じサイズのメンバへのメモリアクセスを確認し、実行中にポインタとして参照されているか、されていないとも指す先が有効なメモリ領域である場合に、ポインタと判定する。

真理値の推論 真理値と同じサイズのメンバへのメモリアクセスを確認し、メモリアクセス時に真理値として用いられているか、そうでなくとも実行中に真理値型が取り得る値(0x00/0xffなど)のみをとる場合に、真理値と判定する。

パディングの推論 オブジェクト内で、①メンバに挟まれている、②実行中にアクセスされていない、③アライメント境界で終端している、の3つの条件を全て満たすメモリ領域を、パディングと判定する。

### 3.2.4 オブジェクト間の参照の解析

オブジェクトグラフの構築に要する、ポインタのメンバの参照元と参照先を解析する。そのために、以下のティント解析とポインタテインティングの技術を応用する。

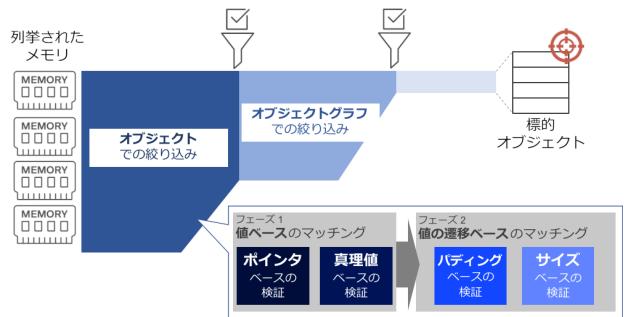


図 3 メモリ解析の全体像

Fig. 3 Overview of our memory analysis.

**ティント解析** データの流れを追跡する技術。ティントタグという属性情報を追跡対象のデータに付与し、転送や演算にあわせて伝播させていく。伝播後のタグの付与状況から、データの流れや由来を判断できる。

**ポインタテインティング** ティント解析でポインタ参照を追跡する技術。追跡したいポインタにタグを付与し、タグ付きのデータがポインタ参照された際に、参照先アドレスのデータにタグを伝播させる。

これらを用いて、ポインタ参照の流れを追跡する。メモリ読み込み時に各ポインタ候補にタグを付与し、実行の過程でこのポインタ候補を参照すると、タグが伝播していく。そのため、このタグを追跡することで、どのポインタがどのアドレスを参照したかが分かる。これにより、ポインタのメンバの参照元・参照先の情報を復元する。

### 3.2.5 オブジェクトグラフの構築

3.2.2 項で再構築した各オブジェクトを、グラフのノードとする。それらのメンバのポインタを参照し、前項で得た参照元・参照先と突き合わせて、エッジを追加する。これを参照先のオブジェクトにも同様に適用して、再帰的にエッジを追加していく、オブジェクトグラフを構築する。

## 3.3 メモリ解析による標的オブジェクトの探索

### 3.3.1 メモリの列挙

メモリ解析の全体像を図 3 に示す。初めに、探索の対象として、静的領域、スタック、ヒープを含むメモリを列挙する。これは、2.2 節で前提に置いた任意アドレス読み込みをしながらメモリ空間を走査することで実現できる。

以降では、この列挙したメモリを絞り込んでいく。ポインタを列挙した上で、その周辺の構造が標的のオブジェクトに適合するかと、オブジェクトの周辺の参照が標的のオブジェクトグラフに適合するかで絞り込む。具体的には、メンバのポインタや真理値、パディング、サイズ、参照が3.2 節で抽出した構造的特徴と一致するかを検証する。

### 3.3.2 ポインタの抽出

列挙されたメモリからポインタを抽出する。まず、割り当てメモリ領域は連続しやすく、ポインタの値が特定の範

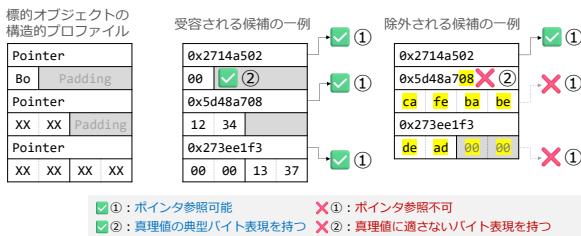


図 4 値による絞り込みの例

Fig. 4 An example of value-based filtering.

囲に集中する性質を利用して、バイト表現のポインタを判定する。列挙されたメモリをアドレスレジスタ幅ごとに切り出してクラスタリングし、上位バイトが頻出するものをポインタの候補とする。また、それらのアライメントが前述の幅で整っているかも確認する。

次に、候補の参照先のメモリ領域が有効かを検証する。エクスプロイト開発などのランタイムでは、任意アドレス読み込みなどを用い、メモリフォレンジックスでは、VAD や PTE などのカーネル構造体の情報で確認する。

この判定を通過したものをポインタとしてリスト化し、以降はそれらを基点として標的オブジェクトを絞り込む。

### 3.3.3 オブジェクトでの絞り込み

まず、ポインタリスト中の 1 つを候補に取り、標的オブジェクトのポインタメンバの 1 つに当てはめて、構造的特徴が一致するかで候補を絞り込む。候補ポインタを基点に各メンバへのオフセットを算出して判定する。判定は値ベースと値遷移ベースの 2 段階で、まず前者の検証で絞り込んだ後、残った候補のみ値の遷移を監視して後者の検証をする。以下に、それぞれの検証方法を示す。

- **値①：ポインタの検証** ポインタと想定されるメンバが、ポインタとして適切な値を保持しているか検証する。前節のポインタリストに含まれているかをみる。
- **値②：真理値の検証** 真理値と想定されるメンバが、真理値として適切な値を保持しているか検証する。真理値型の取る値 (0x00 や 0xff など) かをみる。
- **値遷移①：パディングの検証** パディングと想定される領域の値が実行中に不变か検証する。
- **値遷移②：サイズの検証** 各メンバへの書き込みが想定されるサイズを超えないか検証する。

図 4 に、値による絞り込みの例を示す。ポインタや真理値として適切な値を保持している場合のみ受容し、不適切な値（図中黄色）を保持している場合は除外する。また、図 5 に、値遷移による絞り込みの例を示す。定常的に監視し、パディング領域（図中灰色）やメンバサイズを超える領域（図中橙色）の上書きが観測された候補を除外する。

### 3.4 オブジェクトグラフでの絞り込み

前節での候補のオブジェクトに対し、オブジェクトグラ



図 5 値遷移による絞り込みの例

Fig. 5 Value-transition-based filtering.

フの構造でさらに絞り込む。グラフの各オブジェクトに、前節と同様の検証を再帰的に実施する。まず、候補オブジェクトのメンバのポインタを参照し、参照先のオブジェクトを前節と同様に検証して絞り込む。参照先のオブジェクトがメンバにポインタを持つ限り、再帰的に参照と検証を続け、標的オブジェクトを絞り始めた段階で終了する。

### 3.5 オブジェクトグラフ上の探索

発見した標的のオブジェクトやオブジェクトグラフから、標的メンバを探索する。グラフ上の前向き探索は、通常のオブジェクトへの参照と同様に、ポインタ参照とオフセットの加算で順次進む。一方、後ろ向き探索は、通常のオブジェクトへの参照と逆向きに辿る必要が生じる。そこで、通常とは逆に、前ノードのオブジェクトの先頭を指すポインタをメモリ上から検索し、見つけたアドレスからオフセットを減算して前ノードのオブジェクトの先頭を得る。この処理を反復することで、グラフ上を後ろ向きに進む。以上によって、発見したグラフ上を前後に探索し、標的オブジェクトおよびメンバのメモリ位置を特定する。

## 4. 評価

### 4.1 実験環境

提案手法の評価のため、プロトタイプを実装した。実行トレースの取得に動的バイナリ計装フレームワークの Intel Pin [4] を、ティント解析に libdft64 [5] を用いた。libdft64 にはポインタティントの機能がないため、独自に伝播ルールを追加した。また、グラフの構築と描画には NetworkX と Graphviz を用いた。メモリ解析のロジックはいずれも独自実装である。このプロトタイプに対して、以下の研究上の疑問に答えるために実験を実施した。

**RQ1** バイナリ解析手法は必要な構造的特徴を正しく抽出できるか？

**RQ2** バイナリ解析は現実的な時間内で完了するか？

**RQ3** オブジェクトの位置特定の精度はどの程度か？

**RQ4** メモリ解析は現実的な時間内で完了するか？

表 4 実験環境

Table 4 Experimental environment.

CPU	Intel Core i7-6600U CPU @ 2.60GHz
メモリ	32GB
OS	Ubuntu 24.04 LTS, Windows 11 64-bit

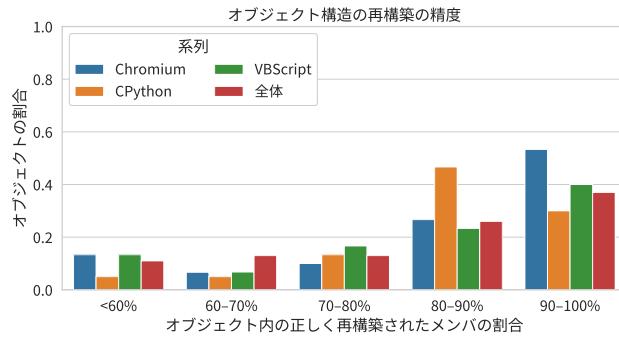


図 6 オブジェクト構造の再構築の精度

Fig. 6 Accuracy of object reconstruction via binary analysis.

実験のために、表 4 に示す環境を VM で構築した。また、標的バイナリとして、GitHub 上の C/C++ のプロジェクトをスター数でソートした中から、ビルド可能であり、テスト入力があり、OSなどを除外したプロセスとして起動できる 30 のプロジェクトを収集し、量的評価に用いた。加えて、ケーススタディとして、オフェンシブセキュリティやメモリフォレンジックスの両面で興味の対象となる、Chromium (V8 を含む), CPython, VBScript の 3 つを用いた。これらの実行状態を保持するオブジェクトは、ヒープ上に存在し、特徴的な値に乏しく流動的な変数も多いため、2.1.3 節の位置特定が困難なオブジェクトにあたる。評価のためにデバッグ情報ありでビルドし、その型情報やオフセットに基づいて正解データを作成した。なお、あくまで評価のみの用途であり、提案手法による解析にデバッグ情報は用いていない。

#### 4.2 バイナリ解析の精度

RQ1 に答えるため、バイナリ解析の精度を評価した。まず、オブジェクトの構造がどの程度の精度で再構築できるかの評価を、図 6 に示す。オフセットとサイズが共に正解データと一致したメンバを正しく再構築されたと判断し、全メンバ数に対する正しく再構築されたメンバ数の割合をオブジェクトの再構築率として評価した。

横軸は再構築率で層別した階層を表し、縦軸はその階層の全体のオブジェクトに占める割合を示す。図より、過半数のオブジェクトは 80% 以上のメンバを正しく再構築できていることが分かる。また、メンバの再構築率が 60% 未満のオブジェクトは、全体の 20% 程度に留まっている。これらから、提案手法による構造体の再構築は、一定の精度を持っていることが分かる。

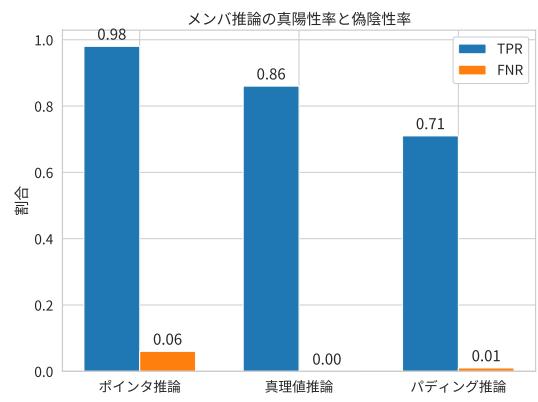


図 7 メンバの推論の精度

Fig. 7 Accuracy of member inference via binary analysis.

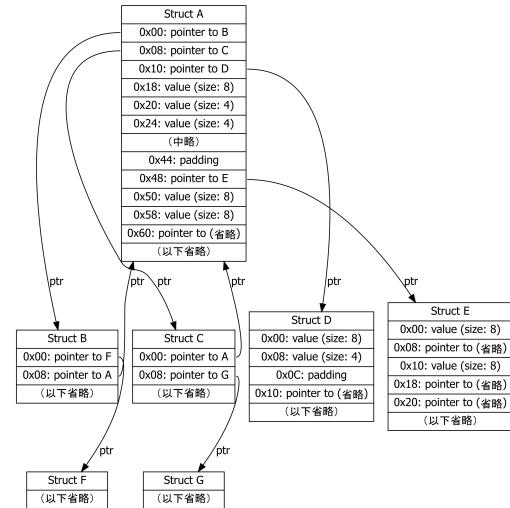


図 8 再構築されたオブジェクトグラフの一例

Fig. 8 An example of reconstructed object graphs.

また、再構築されたメンバに対する推論の精度を、図 7 に示す。ポインタ推論の検知率は 98% と高い一方、見逃し率も 6% あり、主にヌルポインタなどを判定できないことによる。また、真理値推論の検知率は 86% と、こちらも一定の精度を示していた。一方、パディング推論では、推論結果の約 30% がパディングでなかった。この誤りは主に、実行の監視中にメンバへのメモリアクセスが発生しなかったことによる。真理値とパディングの見逃し率はいずれも低く、保守的なロジックによると考えられる。

図 8 に、再構築されたオブジェクトグラフの一例を示す。これは、CPython の実行状態を格納するオブジェクトに関するグラフである。紙面の制約により、一部を省略している。オブジェクト内のメンバのオフセットやサイズ、ポインタや真理値の型、パディング、オブジェクト間の参照関係などは定義と一致しており、構造的特徴を再構築してグラフで表現できていることが分かる。

#### 4.3 バイナリ解析の実行速度

RQ2 に答えるため、前節の実験の際に各ステップの平

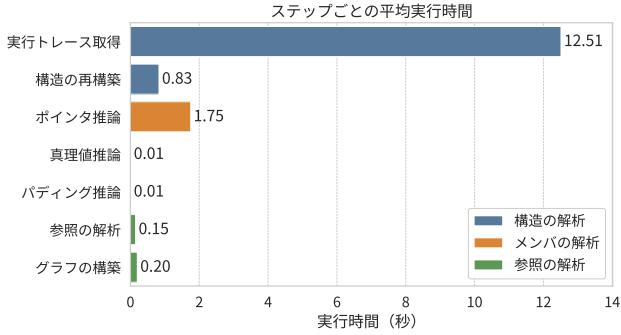


図 9 バイナリ解析手法の実行時間

Fig. 9 Execution duration of our binary analysis technique.

均の実行時間を計測した。計測結果を図 9 に示す。まず、メモリアクセストレースの取得に最も長い時間を要している。これは、取得に際して Intel Pin の VM 上で実行していることと、メモリアクセスの命令の実行ごとに記録のコールバックが発生することによる。また、構造の再構築にはトレースの走査が必要となるため、レコード数に対して  $O(n)$  であり、一定の時間を要している。加えて、値の収集とクラスタリングを要することから、ポインタの推論にも時間を要している。一方、他の真理値推論、パディング推論、参照の解析、グラフの構築については、限られた範囲への素朴なログ分析であり、ごく短い時間で済んでいる。解析全体として、数十秒程度に収まっており、現実的な時間内で構造的特徴の抽出を実現できている。

#### 4.4 メモリ解析の精度

RQ3 に答えるため、メモリ解析の精度を評価した。具体的には、提案手法の各ステップでどの程度オブジェクトを絞り込めるか、最終的に標的オブジェクトの位置を特定できるかを評価した。図 10 に、メモリ解析による絞り込みを示す。縦軸は標的オブジェクトの候補の数であり、横軸のステップが進むにつれて絞り込まれていく。まず、候補数はバイナリによって様々であるものの、ポインタによる検証で大きく絞り込めることが分かる。また、標的オブジェクトに真理値が含まれていなかった CPython を除き、真理値による検証も有効であることが分かる。さらに、パディングやサイズによる検証でも、他の判定ほどではないものの、一定の絞り込みが可能であることも分かる。最終的に、オブジェクトグラフの検証を通じて、いずれのバイナリも標的オブジェクトを絞り込み切れている。

#### 4.5 メモリ解析の実行速度

RQ4 に答えるため、前節の実験の際に各ステップの平均の実行時間を計測した。計測結果を図 11 に示す。

まず、メモリの列挙に最も大きな時間を要していることが分かる。次に時間を要しているのはポインタの判定であり、ポインタの数の多さに加えて、ポインタ参照されてい

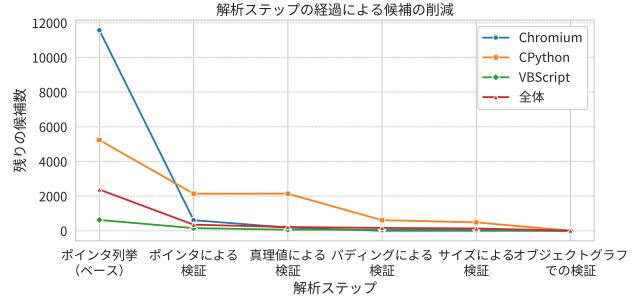


図 10 メモリ解析手法の精度

Fig. 10 Accuracy of our memory analysis technique.

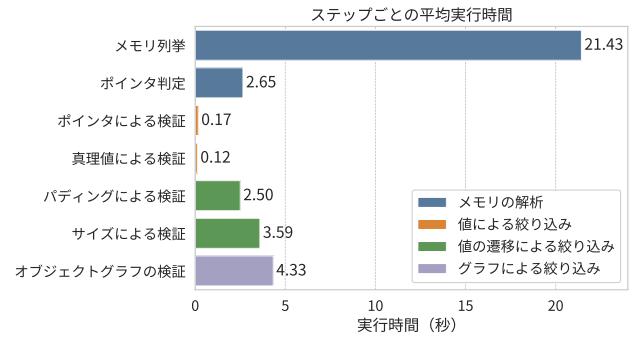


図 11 メモリ解析手法の実行時間

Fig. 11 Execution duration of our memory analysis technique.

るかと参照先が有効かの確認を要することが原因だと考えられる。また、パディングによる検証やサイズによる検証には、候補の定常的な監視を要するため、一定の時間がかかる。一方、値による検証の各ステップについてはあまり時間を要さず、いずれもわずかな時間で完了している。オブジェクトグラフの検証については、各オブジェクトに上記の処理の再帰的な適用が必要であるものの、絞り込まれた候補のみが対象であるため、数秒程度で済んでいる。全体としての実行時間は数十秒程度に収まっており、現実的な時間内での位置特定が可能であることが分かる。

## 5. 議論

### 5.1 制約

まず、バイナリ解析技術の制約を議論する。本手法はオブジェクトの構造的特徴の観測可能性に依存するため、3.2 節の手法では単一経路で観測された実行状態のみでの解析となり、オブジェクトの構造や参照の解明に網羅性の問題が生じ得る。例として、メモリ解析時にポインタが未代入のメンバは、ポインタの検証が失敗する。また、メンバが共用体や汎用ポインタを介してバイナリ解析時とメモリ解析時で異なるオブジェクトを指す場合、グラフでの絞り込みが失敗する。対策として、ファジングなどの多様な入力値を用いた、実行状態の拡張が考えられる。

次に、メモリ解析技術の制約を議論する。少數の小さなオブジェクトで構成されるグラフは、絞り込みに要する構

造的特徴に乏しく、過検知を生じやすい。さらに、ポインタ暗号化などのメモリ保護技術により、生ポインタにアクセスできない場合では、メモリ解析が阻害され得る。

## 5.2 メモリ解析の成功に要するオブジェクト構造の条件

オブジェクトの位置特定に寄与する構造的特徴を、実験の過程で得られた発見的な観点から論じる。メンバ数が4以下、ポインタや真理値の数が2以下、全メンバ数が5以下の小規模オブジェクトは、単体での識別が原理的に難しい。一方で、オフセット0x30以上などの高位にポインタや真理値がある場合、単体でも強い特徴となる。また、オブジェクトグラフのノード数4以上など十分に大きければ、オブジェクト単体の特徴は乏しくとも、グラフ構造を手掛かりに位置特定が成功しやすい。

## 5.3 オブジェクト構造を暴露するリスクと対策

今後、エクスプロイト開発などで構造的特徴の活用が進むと、悪用の可能性も生じ、構造の曝露はリスクとなる。そこで、構造を隠蔽する現実的な対策技術を2つ提案する。

第1に、メモリ保護技術による対策として、ポインタのエンコードや暗号化がある。これによって、ポインタの特徴が消失して検証が困難になるとともに、メモリ解析時の参照の追跡も困難にし、グラフでの判定を失敗させる。

第2に、オブジェクトの構造の変更による対策として、メンバのレイアウトのランダム化がある。これにより、バイナリ解析時とメモリ解析時で構造が一致しなくなり、オブジェクトの発見が困難になる。変更を加えるタイミングに応じて、ソースコードでのメンバの定義順の変更、コンパイル時のコード生成や最適化によるレイアウトの変更、計装によるオブジェクト割り当て時のレイアウトの変更がある。例として、POLaR [6] は LLVM による静的計装で割り当て単位のランダム化を実現している。

## 6. 倫理的配慮

提案手法は幅広いセキュリティ技術に応用でき、その中には攻撃に悪用される可能性のある技術も含まれる。そのため、その技術的詳細と潜在的な脅威について、産業系国際会議 REcon 2025 にて登壇 [7] して先行して警鐘を鳴らしており、セキュリティコミュニティに周知済みである。また、対策手法も5.3節で議論しており、実際に悪用された際にも速やかに対策を講じられる。以上より、セキュリティ技術の発展のメリットが悪用のデメリットを上回ると考え、本論文を公開する。

## 7. 関連研究

DSCRETE [8] は、バイナリを解析して、データ解釈やスキャンのためのロジックを抽出し、メモリフォレンジックスに再利用している。DeepMem [9] は、グラフニュー

タルネットワークを用いてカーネルオブジェクトをメモリダンプから学習し、メモリスキャンを実現している。DroidScraper [10] は、Android のランタイムによって提供されるクラスや参照の情報に基づき、メモリダンプからオブジェクトを再構築している。

しかし、これらの研究では、オブジェクトの構造的特徴を用いるという概念や、そのためのバイナリおよびメモリの解析技術は提案されていない。

## 8. おわりに

本論文では、オブジェクトのメモリ位置特定に従来技術が有効でないケースに着目し、特徴的な値の代わりに特徴的なオブジェクト構造を用いる、新たな概念のアプローチを提案した。その実現のために、標的バイナリからオブジェクトの構造的特徴を抽出するバイナリ解析技術と、それに基づいてメモリ上の標的オブジェクトを探索するメモリ解析技術をそれぞれ提案した。実世界のバイナリに対する実験を通じて、提案手法が標的のオブジェクトのメモリ位置を現実的な時間内で特定できることを示した。他のバイナリ解析技術との併用による解析精度のさらなる向上が今後の課題である。

## 参考文献

- [1] Németh et al.: When Every Byte Counts — Writing Minimal Length Shellcodes, *SISY '15*, pp. 269–274.
- [2] Roney et al.: Identifying Valuable Pointers in Heap Data, *SPW '21*, pp. 373–382.
- [3] Liang et al.: K-Leak: Towards Automating the Generation of Multi-Step Info leak Exploits against the Linux Kernel, *NDSS '24*.
- [4] Luk et al.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *PLDI '05*, pp. 190–200.
- [5] Chen et al.: Angora: Efficient Fuzzing by Principled Search, *SP '18*, pp. 711–725.
- [6] Kim et al.: POLaR: Per-Allocation Object Layout Randomization, *DSN '19*, pp. 505–516.
- [7] Usui et al.: Egg Hunting without Eggs: Identifying Memory Locations of Objects Using Structural Characteristics, <https://cfp.recon.cx/recon-2025/talk/SEKYGC/>.
- [8] Brendan et al.: DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse, *USENIX Security '14*, pp. 255–269.
- [9] Song et al.: DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis, *CCS '18*, pp. 606–618.
- [10] Ali et al.: DroidScraper: A Tool for Android In-Memory Object Recovery and Reconstruction, *RAID '19*, pp. 547–559.