一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

SBOMの整合性検証に向けたライブラリ間依存関係に基づく静的依存ライブラリ特定手法

長崎　智人† 　神田　哲也†† 　井上　克郎††† 　眞鍋　雄貴†††† 仇　　実†††††
肥後　芳樹†

† 大阪大学 〒565–0871 大阪府吹田市山田丘 1–5
†† ノートルダム清心女子大学 〒700–8516 岡山県岡山市北区伊福町 2–16–9
††† 立命館大学 〒567–8570 大阪府茨木市岩倉町 2–150
†††† 福知山公立大学 〒620–0886 京都府福知山市字堀 3370
††††† 株式会社東芝 〒212–8582 神奈川県川崎市幸区堀川町 72 番地 34
E-mail: †{t-nagask,higo}@ist.osaka-u.ac.jp, ††kanda@m.ndsu.ac.jp, †††inoue-k@fc.ritsumei.ac.jp,
††††manabe-yuki@fukuchiyama.ac.jp, †††††shi.qiu.b79@mail.toshiba

あらまし　ソフトウェア部品表（SBOM）と実行バイナリ間の整合性検証では，実行バイナリに静的リンクされたライブラリを正確に特定することが重要である．このようなライブラリ特定を行う技術はバイナリ SCA と呼ばれる．既存のバイナリ SCA 手法は各ライブラリを独立に検出するため，ライブラリ間の依存関係を考慮していない．本研究では，関数の意味的類似度に加えてライブラリ間の依存関係を活用し，類似関数ペアを段階的に絞り込むバイナリ SCA 手法を提案する．評価実験の結果，提案手法は既存手法を上回る精度で静的リンクライブラリを特定できることを確認した．さらに，依存関係に基づくフィルタリングの有効性について分析した．

キーワード　ソフトウェアサプライチェーン，SBOM，バイナリ SCA，ライブラリ間依存関係

1. はじめに

オープンソースソフトウェア（Open Source Software; OSS）はソフトウェア開発において広く利用されており [1]，企業のソフトウェア開発コスト削減に大きく寄与している [2]．一方で，OSS の利用はセキュリティリスクやライセンス違反リスクを伴う [3]．これらのリスクを回避するためには，ソフトウェアに利用されているコンポーネントを正確に把握することが重要であり，その手段としてソフトウェア部品表（Software Bill of Materials; SBOM）の活用が注目されている [3]．SBOM はソフトウェアに利用されているコンポーネントの情報を機械可読な形式で記述したドキュメントであり，コンポーネントの名称，バージョン，ライセンス，依存関係などの情報が含まれる．近年では，製造業者に SBOM の生成を求める法制度が整備され始めており [4] [5]，SBOM は実運用へ向けて普及が進みつつある．

一方で，SBOM 生成時のミスや改ざん，アップデートに伴う利用コンポーネントの変更などにより，SBOM の記載内容と実際のソフトウェアとの間に乖離が生じる可能性がある．そのため，SBOM の記載内容が実際のソフトウェアと整合していることを検証する技術が求められる．また，実行バイナリと SBOM のみが提供され，ソースコードや依存関係ファイルが提供されない状況では，実行バイナリから利用ライブラリを特定し，SBOM との整合性を確認することが求められる．特に，静的リンクされたライブラリの特定は困難かつ重要な課題である．

実行バイナリに静的リンクされたライブラリを特定する技術はバイナリ SCA と呼ばれ，複数の手法が存在する [6]〜[10]．しかし，既存手法は各ライブラリを独立に検出するため，ライブラリ間の依存関係を考慮していない．依存関係を利用することでライブラリ特定精度の向上が期待できるほか，依存関係の観点から妥当なライブラリ群が結果として得られることで，SBOM に記載された依存関係との比較が容易になる可能性がある．そこで本研究は，関数の意味的類似度に加えてライブラリ間の依存関係を活用したライブラリ特定手法を提案する．提案手法では，依存関係に基づいて類似関数ペアを段階的に絞り込むフィルタリングを行う．実験の結果，提案手法は既存手法を上回る精度でライブラリを特定できることを確認した．また，各フィルタリング戦略の効果を分析した．

2. 背　　景

2.1 SBOM と実行バイナリ間の整合性検証

2.1.1 SBOM 整合性検証の必要性

SBOM はソフトウェアに利用されているコンポーネントの情報を機械可読な形式で記述したドキュメントであり，コンポーネントの名称，バージョン，ライセンス，依存関係などの情報が含まれる．SBOM を活用することで，利用コンポーネントに存在する脆弱性やライセンス違反の有無を迅速に把握し，対応することができる．しかし，SBOM 記述内容が実際のソフトウェアと一致していない場合，誤った情報に基づいて脆弱性管

理やライセンス管理が行われるおそれがあるため，SBOM の記載内容が実際のソフトウェアと整合していることが重要である．SBOM の記述に対する信頼性を高める手法として，ブロックチェーンを用いて SBOM の改ざん耐性を高める手法 [11] や，SBOM の生成過程を追跡する SBOMit [12] などが提案されている．しかし，これらは実際のビルド成果物との整合性を直接保証するものではなく，SBOM を生成する提供者自身が悪意を有する場合に改ざんを完全に防止することは困難である．

この問題は，ソースコードや依存関係管理ファイルが利用できない状況において特に重要となる．コンポーネントに関するライセンスや依存関係などの情報を検証するためには，まずソフトウェアに含まれるコンポーネントを正確に特定する必要がある．ビルド成果物と SBOM のみが提供される状況では，ビルド成果物自体を解析して依存コンポーネントを特定する Software Composition Analysis（SCA）が必要となる．しかし，ビルド成果物から利用可能な情報は限定的であるため，コンポーネント特定精度には依然として課題が残されている．

2.1.2 バイナリ SCA の既存手法と課題

SBOM 整合性検証では様々なソフトウェア成果物が対象となるが，特に実行バイナリに静的リンクされたライブラリの特定は困難であり，SBOM 整合性検証における重要な課題である．コピー&ペーストや静的リンクなどによってバイナリ内に取り込まれたコンポーネントを特定する技術はバイナリ SCA と呼ばれる．バイナリ SCA の代表的なアプローチとして，埋め込み文字列を用いる手法と意味的類似度を用いる手法が存在する．

埋め込み文字列を用いる手法は，実行バイナリ中に含まれる文字列情報を手掛かりにライブラリを特定する．例えば，“OpenSSL 1.1.1f 31 Mar 2020” のようなバージョン文字列が検出できればライブラリ名やバージョンを推定できる．このアプローチは実行バイナリから SBOM を生成する手法にも応用されており，正規表現に基づくケースベース手法を採用した cve-bin-tool [13] や，Transformer モデルを用いた SBOM 生成手法 [14] が代表例として挙げられる．しかし，バージョン文字列を含むライブラリは限定的であるため [15]，検出漏れが生じる恐れがある．静的リンクされたライブラリを高精度に特定するためには，文字列情報への依存を避け，バイナリそのものが持つ意味的特徴を利用するアプローチが重要である．

2.2 意味的類似度に基づくバイナリ SCA

2.2.1 既 存 手 法

意味的類似度を用いる手法は，ライブラリと解析対象のバイナリの関数を比較し，意味的に類似する関数の存在からライブラリの利用有無を推定する．この際，類似バイナリコード検出（Binary Code Similarity Detection; BCSD）技術が用いられる．BCSD は，コンパイラや最適化オプションの違いによる命令列の変化を越えて関数間の意味的類似度を推定する技術である．近年では，アセンブリ命令列や制御フローグラフなどの情報を機械学習モデルでベクトル化し，コサイン類似度などで比較する手法が広く研究されている．代表的な BCSD 手法として，jTrans [16] や UniASM [17] が提案されている．

意味的類似度を用いたバイナリ SCA の近年の研究例として，

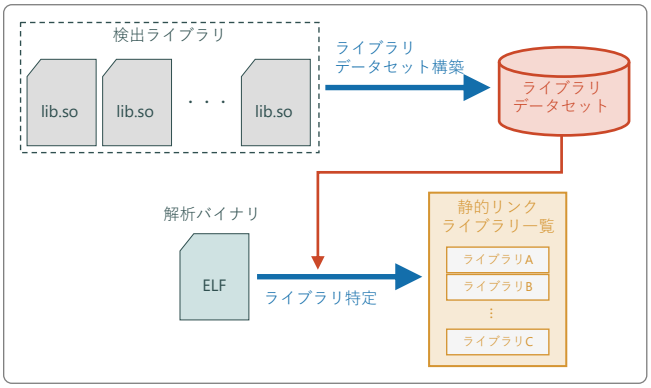

図 1　提案手法の全容

関数類似度にリンク時局所性やコールグラフを組み合わせる BinaryAI [9], プログラムをモジュールに分割しモジュール単位の意味的類似度マッチングを行う ModX [7] などが提案されている．また，これらを含む複数の既存手法を上回る精度を達成している BinCoFer [10] は，類似度比較の対象とする関数をライブラリの重要な機能を担う「コア関数」に限定することで，検出精度の向上とデータの冗長性の削減を達成している．BinCoFer では，後述の方法で抽出したコア関数に TF-IDF に基づく重み付けを行い，多くのライブラリに類似関数が存在する関数の重みを小さく，特定のライブラリを特徴付ける関数の重みを大きくする．その後，コア関数ごとに最も類似する関数を解析対象のバイナリから探し，類似度の重み付き平均を取ることで，ライブラリと解析対象バイナリとの類似度スコアを算出する．

2.2.2 コア関数の定義

本研究では，BinCoFer が採用するコア関数の考え方を導入する [10]．同論文では，ライブラリの重要な機能は複雑なロジックを持つエクスポート関数に含まれるという仮定に基づき，エクスポートのうち保守性指標（Maintainability Index; MI）が低い関数を一定割合抽出した関数群をコア関数として定義している．MI は Cyclomatic 複雑度（CC），コード行数（LOC），Halstead 複雑度（HV）を用いた以下の式で求められる．

$$MI = 171 - 5.2 * ln(HV) - 0.23 * CC - 16.2 * ln(LOC)$$

3. 提 案 手 法

3.1 提案手法のアイデア

既存のバイナリ SCA 手法は，一般に各ライブラリの利用有無を独立に判定する．しかし，ライブラリ間には依存関係が存在し，あるライブラリが実行バイナリに含まれるかどうかは他のライブラリの存在と無関係ではないと考えられる．ライブラリ間の依存関係情報を考慮することで，ライブラリ特定精度の向上が期待される．また，依存関係の観点から妥当なライブラリ群が結果として得られることで，SBOM に記載された依存関係との比較が容易になる可能性がある．そこで本研究は，バイナリ関数の意味的類似度に加えて，ライブラリ間の依存関係を活用するバイナリ SCA 手法を提案する．

3.2 提案手法の全容

本論文では，バイナリ関数の意味的類似度とライブラリ間

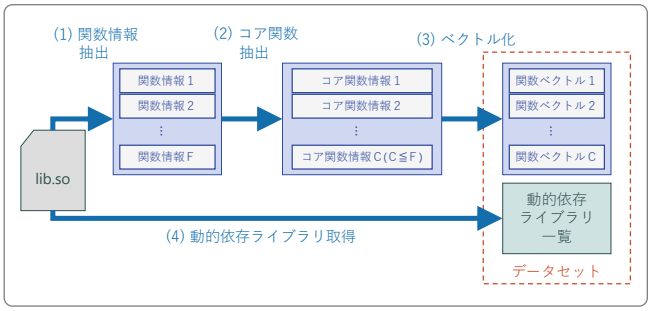

図 2　ライブラリデータセットの構築

の依存関係を利用し，解析対象のバイナリ（以降，解析バイナリ）に静的リンクされたライブラリを特定する手法を提案する．図 1 にその全容を示す．本手法は，ライブラリデータセット構築とライブラリ検出の 2 つの工程に分かれている．ライブラリデータセット構築では，検出対象のライブラリ（以降，検出ライブラリ）を解析しライブラリデータセットを作成する．ライブラリ検出では，検出ライブラリと解析バイナリとの間で関数の意味的類似度を比較し，最終的な静的リンクの有無を判断する．

本手法では，BCSD 手法として UniASM [17] を用いる．ただし，ライブラリデータセットに含まれる全ての関数ベクトル対から算出した類似度の最小値および最大値を用いて min–max 正規化を行い，関数類似度を [0, 1] の範囲に変換する．

3.3　ライブラリデータセット構築

ライブラリデータセット構築手順を図 2 に示す．本手法では，検出ライブラリの共有オブジェクト（.so ファイル）を情報源として利用する．データセットは，コア関数のベクトル集合と検出ライブラリの動的依存ライブラリ一覧で構成される．

はじめに，共有オブジェクトを逆アセンブルすることで関数情報抽出（図 2 の (1)）を行い，関数のアセンブリ命令列と MI を取得する．アセンブリ命令列の取得には UniASM が内蔵する逆アセンブルツールを利用し，MI は Ghidra [18] から取得した関数のメトリクス情報を用いて計算する．続いて，エクスポート関数のなかで MI が下位 20% の関数をコア関数として抽出し（図 2 の (2)），UniASM を用いてコア関数をベクトル化する（図 2 の (3)）．下位 20% というしきい値は BinCoFer に倣い設定した．また，バイナリのメタデータを解析することでライブラリが動的に依存するライブラリ一覧を取得する（図 2 の (4)）．

3.4　ライブラリの検出

ライブラリ検出手順を図 3 に示す．はじめに，Ghidra と UniASM を用いて解析バイナリの関数ベクトルの取得（図 3 の (1)）およびコールグラフの構築（図 3 の (2)）を行う．なお，ノイズ除去のため 10 行以下の関数は解析対象から除外する．次に，各ライブラリのコア関数ごとに，意味的類似度が最も高い解析バイナリ中の関数を探索する（図 3 の (3)）．以降，このような対応関係を類似関数ペアと呼ぶ．また，類似関数ペア p のライブラリ関数を $\mathrm{func}^{\mathrm{lib}}(p)$，解析バイナリ関数を $\mathrm{func}^{\mathrm{bin}}(p)$ と表記し，関数 $\mathrm{func}^{\mathrm{lib}}(p)$ が属するライブラリを $\mathrm{lib}(p)$ と表記する．

その後，次節にて詳述するライブラリ間の依存関係を利用したフィルタリングにより，類似関数ペアを段階的に削減する

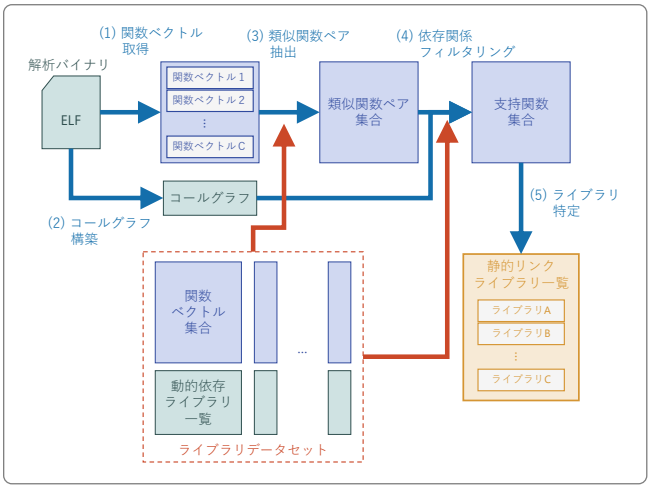

図 3　ライブラリの検出

（図 3 の (4)）．削減後に残った任意の類似関数ペア p は，$\mathrm{lib}(p)$ がバイナリに静的リンクされていることを支持する証拠とみなし，$\mathrm{func}^{\mathrm{lib}}(p)$ を支持関数と呼ぶ．最終的に，ライブラリのコア関数のうち支持関数が占める割合がしきい値 θ_{r} を超えたライブラリを，静的リンクされていると判定する（図 3 の (5)）．

3.5　類似関数ペアのフィルタリング

類似関数ペアのフィルタリングでは，関数単位の類似度に加えてライブラリ単位の類似度も利用することでフィルタリング精度の向上を図る．そこで，既存手法である BinCoFer のうち，ライブラリ・バイナリ間の類似度スコアを算出する部分を再実装し，ライブラリ類似度算出器として用いる．BinCoFer では BCSD 手法としてファインチューニング済みの jTrans [16] を利用しているが，本研究では UniASM を用いる．また，逆アセンブルツールとして利用されている IDA Pro [19] は Ghidra に置き換える．各しきい値の設定は BinCoFer の論文に従う．

類似関数ペアのフィルタリングは，ノイズ除去を目的とする前処理フィルタリングから開始する．この処理では，関数類似度がしきい値 θ_{fl} を下回る類似関数ペア，およびライブラリ類似度がしきい値 θ_{b} 未満のライブラリに属する類似関数ペアを除外する．前処理フィルタリング後，残存する類似関数ペアに対応するライブラリの集合を静的依存ライブラリ候補とする．

続いて，ライブラリ間の依存関係を利用して類似関数ペアを絞り込む．本処理では，類似関数ペア間の依存関係の矛盾を検出し，各類似関数ペアに付与した優先度に基づいて優先度の低い類似関数ペアを除外することで矛盾を解消する．具体的には以下の 3 種類のフィルタを適用する．ただし，各フィルタは独立に適用され，適用順序に依存しない．

（1）　重複フィルタ（DUP）：

ひとつの解析バイナリ関数に対し複数のライブラリ関数が類似関数ペアを成している場合，最も優先度が高いペア以外のペアを除外する．

（2）　依存関係整合フィルタ（CALL）：

2 つの類似関数ペア p_1 と p_2（ただし $\mathrm{lib}(p_1) \neq \mathrm{lib}(p_2)$）について，$\mathrm{func}^{\mathrm{bin}}(p_1)$ から $\mathrm{func}^{\mathrm{bin}}(p_2)$ への関数間の依存関

係があるにも関わらず，lib(p_1) から lib(p_2) へのライブラリ依存関係がない場合，p_1 と p_2 のうち優先度が低いペアを除外する．

（3） 依存ライブラリ充足フィルタ（LIB）：

類似関数ペア p について，lib(p) が依存しているライブラリのうち，静的依存ライブラリ候補および動的依存ライブラリどちらにも含まれないライブラリが存在する場合，類似関数ペア p を除外する．

ただし，類似関数ペアの類似度とライブラリ類似度をそれぞれ s_f，s_l と置いたとき，優先度の計算方式 P は ADD（$s_f + s_l$），MULT（$s_f * s_l$），FUNC（s_f），BCF（s_l）の 4 つを用意する．

最後に，関数類似度がしきい値 $\theta_{\mathrm{f2}} (\geq \theta_{\mathrm{f1}})$ を下回る類似関数ペアを除外する後処理フィルタリングを行う．これにより他の関数やライブラリとの依存関係が少なく，上記のフィルタでは除外できなかったペアを除去する．

4. 実験設計

提案手法を評価するため，バイナリ SCA タスクを対象として 2 種類の実験を行う．まず，すべてのフィルタリングを適用した提案手法を対象とする評価実験を行い，現実的なしきい値設定のもとで性能を評価するとともに，既存手法との比較を通じて提案手法の有効性を検証する．また，依存関係フィルタ DUP，CALL，LIB が性能に与える影響を分析するため，各フィルタの有無による性能の変化を比較するアブレーション実験を行う．

4.1 データセットおよび比較対象

既存手法との比較のため，BinCoFer の GitHub リポジトリで公開されているデータセットを利用する．同データセットには，ArchLinux から収集された 102 個の検出ライブラリ，110 個の学習用解析バイナリ，および 38 個の評価用解析バイナリが含まれている．なお，BinCoFer の発表論文には評価用バイナリ数が 40 個であると記載されており，両者の間に不一致が確認された．以降，同論文内で利用された評価用バイナリデータセットを BCF-Paper，リポジトリで公開されている評価用バイナリデータセットを BCF-Repo と呼ぶ．また，同論文で比較対象とされている BinaryAI の当時の最新リリース (2024.2.26) は，独自のライブラリデータセットを持つオンラインプラットフォームであるため，BinCoFer リポジトリに含まれる検出ライブラリデータセットを BCF-Lib，BinaryAI の検出ライブラリデータセットを BinaryAI-Lib と呼び区別する．

各実験とデータセットの対応を表 1 に示す．先行研究では，BCF-Paper と BCF-Lib を用いて BinCoFer [10]，B2SFinder [6]，ModX [7]，LibAM [8] の評価実験が行われている．また，BCF-Paper および BinaryAI-Lib を用いて BinaryAI [9] の評価実験が行われている．比較対象としてこれらの実験結果を引用する．提案手法の評価実験およびアブレーション実験には，いずれも BCF-Repo と BCF-Lib を用いる．また，BinCoFer 全体を再実装した手法である再実装版 BinCoFer についても同データセット上で評価を行う．再実装版 BinCoFer は，先述のライブラリ類似度算出器に類似度しきい値によるライブラリ利用判定処理を組み合わせたものであり，ライブラリ類似度算出器が適切に実装されていることを確認するために実施する．最後に，BinCoFer のリポジトリにおける 110 個の学習用バイナリからランダムに選択した 10 バイナリを BCF-Mini と呼び，BCF-Mini および BCF-Lib を用いてパラメータ設定のための予備実験を行う．

4.2 評価指標

一般的に使われる指標である再現率，適合率，F1 値を計算する．それぞれの計算式および付随する概念を以下に示す．

- 真陽性数（TP）：実際に静的リンクされており，正しく静的リンクされていると判断したライブラリの数．
- 偽陽性数（FP）：実際には静的リンクされていないが，静的リンクされていると判断したライブラリの数．
- 偽陰性数（FN）：実際には静的リンクされているが，静的リンクされていないと判断したライブラリの数．
- 再現率（Recall）：Recall = TP/(TP + FN)
- 適合率（Precision）：Precision = TP/(TP + FP)
- F1 値 : F1 = 2 × Precision × Recall/(Precision + Recall)

ただし，再現率と適合率は一般にトレードオフの関係にあり，一方を軽視することでもう一方を向上させることが可能である．そのため，両指標を総合的に評価できる F1 値を主要な評価指標とし，手法間および条件間の比較は主に F1 値に基づいて行う．

4.3 パラメータの設定

提案手法の評価実験では，4 つのしきい値 $\theta_{\mathrm{f1}}, \theta_{\mathrm{b}}, \theta_{\mathrm{f2}}, \theta_{\mathrm{r}}$ および優先度計算方式 P をパラメータとして事前に設定する．予備実験にて，$\theta_{\mathrm{f1}} \in \{0.75, 0.80, 0.85, 0.90\}$，$\theta_{\mathrm{b}} \in \{0.75, 0.80, 0.85, 0.90\}$，$\theta_{\mathrm{f2}} \in \{0.80, 0.85, 0.90, 0.95\}$，$P \in \{\mathrm{ADD}, \mathrm{MULT}, \mathrm{FUNC}, \mathrm{BCF}\}$ を組み合わせた全 208 組（ただし $\theta_{\mathrm{f1}} \leq \theta_{\mathrm{f2}}$ ）を実行し，F1 値が最大となる θ_{r} を 0.01 刻みで探索した．その結果，上位 15 組の F1 値の差が 0.01 以内の範囲に収まり，一方で 15 位と 16 位の間におよそ 0.03 の差が生まれたため，上位 15 組を高性能群として抽出した．高性能群には P = BCF は含まれておらず，一方で $(\theta_{\mathrm{f1}}, \theta_{\mathrm{b}}, \theta_{\mathrm{f2}}) = (0.85, 0.75, 0.90), (0.85, 0.80, 0.9)$ のしきい値の組み合わせについては，P = ADD, MULT, FUNC の全てが高性能群に含まれていた．加えて，その 6 件のうち 5 件は $\theta_{\mathrm{r}} \in [0.26, 0.38]$ の範囲で最大の F1 値をとったため，その中央値である 0.32 を θ_{r} の代表値とした．以上より，評価実験では $(\theta_{\mathrm{f1}}, \theta_{\mathrm{f2}}, \theta_{\mathrm{r}}) = (0.85, 0.90, 0.32)$ を固定したうえで，$\theta_{\mathrm{b}} \in \{0.75, 0.80\}$，$P \in \{\mathrm{ADD}, \mathrm{MULT}, \mathrm{FUNC}, \mathrm{BCF}\}$ を満たす 8 つの組み合わせを用いる．P = BCF は性能において有力な選択肢とは考えられないが，性能比較の網羅性のために実行する．

表 1　実験とデータセットの組み合わせ

実験	評価用バイナリ	検出ライブラリ
提案手法	BCF-Repo	BCF-Lib
再実装版 BinCoFer	BCF-Repo	BCF-Lib
予備実験	BCF-Mini	BCF-Lib
既存手法（BinaryAI を除く）†	BCF-Paper	BCF-Lib
既存手法（BinaryAI）†	BCF-Paper	BinaryAI-Lib

† は先行研究からの引用 [10] を示す．

アブレーション実験では，しきい値 θ_{f1}，θ_b，θ_{f2} は評価実験と同様の値を用い，優先度の算出方式は P = MULT のみに限定する．しきい値 θ_r の最適値は適用するフィルタの組合せによって大きく異なると考えられるため，各条件ごとに最良の F1 値を与える θ_r を事後的に採用する．

5. 実験結果

5.1 評価実験

表 2 に既存手法の精度を示す [10]．提案手法の一部に用いた再実装版 BinCoFer は BinCoFer よりも低い精度となり，これは BCSD 手法および逆アセンブルツールの違いに起因すると考えられる．一方で，各指標において 1 つ以上の既存手法を上回る性能を示したため，ライブラリ単位の類似度算出手法として提案手法内で利用するうえで致命的な問題はないと判断した．

すべてのフィルタを適用した場合の提案手法の実行結果を表 3 に示す．最良の F1 値を太字で示している．θ_b = 0.80, P = ADD, MULT で最良の F1 値 0.847 を示した．いずれの θ_b においても，P = ADD, MULT において最良の F1 値をとり，P = BCF で最悪となった．提案手法は，P = BCF を除く 6 種類のパラメータ設定において，全ての指標で既存手法を上回った．F1 値においては，既存手法で最良の結果を示す BinCoFer と比較し，最低でも 0.05 以上，最大で 0.09 以上の向上が見られた．

また，予備実験と同様の全パラメータの組み合わせ 208 組を実行し，最適なしきい値 θ_r を事後的に与えた場合，最良の F1 値は 0.867 であり，BinCoFer を 0.11 以上上回った．これは，評価データセットに対して最適なパラメータを事前に選択できた場合の性能であり，本手法の上限性能に相当する．

表 2　既存手法の精度

	再現率	適合率	F1 値
再実装版 BinCoFer	0.515	0.745	0.609
BinCoFer†	0.649	0.893	0.752
B2SFinder†	0.609	0.679	0.642
ModX†	0.648	0.693	0.669
LibAM†	0.384	0.936	0.545
BinaryAI†	0.684	0.684	0.684

† は先行研究からの引用 [10] を示す．

表 3　提案手法の実行結果

θ_{f1}	θ_b	θ_{f2}	θ_r	計算方式	再現率	適合率	F1 値
0.85	0.75	0.90	0.32	ADD	0.735	0.980	0.840
				MULT	0.735	0.980	0.840
				FUNC	0.691	0.979	0.810
				BCF	0.544	0.974	0.698
0.85	0.80	0.90	0.32	ADD	0.735	1.000	**0.847**
				MULT	0.735	1.000	**0.847**
				FUNC	0.691	1.000	0.817
				BCF	0.544	1.000	0.705

評価実験結果の総括：提案手法は，適切なパラメータ設定を行うことで既存手法を全ての評価指標で上回る性能を示した．

5.2 アブレーション実験

表 4 に結果を示す．表中の ✓ および ✗ はそれぞれフィルタの使用および不使用を表し，θ_r 列は最良の F1 値を与えた θ_r の中央値を示す．また，各パラメータ設定における最良の F1 値を太字で示す．いずれのパラメータ設定においても，DUP または CALL を使用した場合には全フィルタ不使用時と比較して F1 値が 0.05 以上向上した．一方，LIB のみの使用による性能向上は見られなかった．また，DUP または CALL を使用した状態で他のフィルタを追加した場合の F1 値の変化は 0.02 以内に留まり，性能向上は限定的だった．特に，CALL のみを使用した場合に最大の F1 値を示しており，追加のフィルタは性能を改善しないだけでなく，場合によっては低下させる結果となった．

アブレーション実験結果の総括：DUP および CALL フィルタは性能向上に寄与した一方，LIB フィルタによる性能向上は見られなかった．また，複数のフィルタを同時に使用した場合の効果は限定的であった．

6. 考　　察

6.1 優先度計算方式の検討

実験では，優先度計算方式として ADD および MULT を用いた場合に高い F1 値を示し，FUNC および BCF はより低い性能となった．FUNC では偶然類似する関数を持つライブラリを高く評価してしまう可能性がある一方，BCF では関数単位の特徴が一切考慮されないため，関数単位で候補を絞り込む処理の効果が十分に活用されない可能性がある．ADD および MULT では，両者の欠点を補うことで高い性能が得られたと考えられる．本研究では単純な和と積のみの検証にとどまっているため，重

表 4　アブレーション実験の結果

パラメータ	DUP	CALL	LIB	θ_r	再現率	適合率	F1 値
	✓	✓	✓	0.13	0.765	0.945	0.846
	✗	✓	✓	0.41	0.765	0.981	**0.860**
θ_{f1} = 0.85	✓	✗	✓	0.28	0.735	0.980	0.840
θ_b = 0.75	✓	✓	✗	0.32	0.735	0.980	0.840
θ_{f2} = 0.90	✗	✗	✓	0.52	0.574	0.951	0.716
P = MULT	✗	✓	✗	0.41	0.765	0.981	**0.860**
	✓	✗	✗	0.30	0.735	0.980	0.840
	✗	✗	✗	0.52	0.574	0.951	0.716
	✓	✓	✓	0.13	0.765	1.000	**0.867**
	✗	✓	✓	0.26	0.765	1.000	**0.867**
θ_{f1} = 0.85	✓	✗	✓	0.28	0.750	1.000	0.857
θ_b = 0.80	✓	✓	✗	0.13	0.765	1.000	**0.867**
θ_{f2} = 0.90	✗	✗	✓	0.47	0.735	0.877	0.800
P = MULT	✗	✓	✗	0.26	0.765	1.000	**0.867**
	✓	✗	✗	0.28	0.750	1.000	0.857
	✗	✗	✗	0.47	0.735	0.877	0.800

み付き和や調和平均など，他の計算方式も検証する必要がある．

6.2 依存関係フィルタリングの有効性と限界

LIB フィルタの効果が確認できなかった要因として，静的依存ライブラリ候補の生成方法が考えられる．本手法では，前処理フィルタを通過した類似関数ペアが 1 件でも存在すれば当該ライブラリを候補として保持するため，多くのライブラリが候補集合に残り，LIB フィルタによる除外が十分に機能しなかった可能性がある．LIB フィルタの有効性を高めるためには，候補をより厳密に選定する方法を検討する必要がある．ただし，実験結果は DUP または CALL フィルタ単独の使用によって誤検出の大部分を除去できている可能性を示唆している．このことから，LIB フィルタを改善したとしても手法全体の性能向上への寄与は限定的であり，類似関数ペア抽出に用いる BCSD 手法や前処理・後処理フィルタなど，他の処理が性能のボトルネックとなっている可能性がある．これらの仮説を検証するため，今後は各フィルタリング処理の詳細な振る舞いを分析する必要がある．

7. 妥当性への脅威

本研究で使用した BinCoFer リポジトリのデータセットは，各バイナリに静的リンクされたライブラリ数は 1〜2 個程度のものが多く，利用ライブラリは libc など一部に偏っていた．そのため，より多くのライブラリが静的リンクされた複雑なバイナリでは，ライブラリ識別性能が変化する可能性がある．提案手法は既存研究と同様の実験設計において既存研究を上回る精度を達成しているため，既存研究が想定する対象に対する有効性の結論には影響しないものの，より複雑な依存関係を持つ実ソフトウェアに対する有効性については追加の評価が必要である．

また，本研究ではライブラリ単位の依存関係を前提としているが，実際はライブラリ全体ではなく必要な関数のみが取り込まれる場合がある．そのため，ライブラリ間の依存関係がバイナリ上では成立せず，LIB フィルタにおいて本来不要なライブラリを要求してしまう可能性がある．本研究では LIB フィルタによる再現率の低下は確認されなかったため，少なくとも本研究で用いたデータセットに対しては大きな影響はなかったと考えられる．ただし，現実に使用されるライブラリやコンパイル設定においては未検証であり，今後は実際のリンク結果を考慮したフィルタリング手法への拡張が必要である．

8. おわりに

本研究では，ライブラリ間の依存関係に着目したバイナリ SCA 特定手法を提案した．実験の結果，依存関係に基づくフィルタリングはライブラリ特定に有効であり，特に重複フィルタおよび依存関係整合フィルタが精度向上に大きく寄与することが分かった．また，提案手法は既存手法を上回る F1 値を達成し，ライブラリ間の依存関係を活用することの有効性を示した．

一方で，本研究で用いた評価データセットは依存関係が少ない比較的単純なバイナリが中心であり，現実で利用される複雑なソフトウェアに対する有効性については十分に検証できていない．特に，提案手法によって得られるライブラリ群が依存関係の観点からどの程度妥当な構成となっているか，またそのような検出結果が SBOM 整合性検証においてどのように活用できるかについては，今後詳細に評価する必要がある．

今後は，実際のソフトウェアから生成された SBOM を対象とした検証や，より多様なライブラリ構成および複雑な依存関係を持つソフトウェアを対象とした検証を行い，提案手法の適用範囲や実運用環境における有効性を明らかにする．

謝辞 本研究は JSPS 科研費 JP23K28065，JP24K14895，JP24H00692，JP25K03102，JP26K02889，JP26H02500 の助成を受けたものです．

文　献

[1] M. Gerosa and A. Lawson, “The State of Global Open Source 2025,” Technical report, The Linux Foundation, 2025.
[2] M. Hoffmann, F. Nagle, and Y. Zhou, “The Value of Open Source Software,” Harvard Business School Working Paper 24-038, 2024.
[3] Black Duck Software, “2026 Open Source Security and Risk Analysis Report,” Technical report, Black Duck Software, 2026.
[4] European Parliament and Council of the European Union, “Regulation (EU) 2024/2847 on Horizontal Cybersecurity Requirements for Products with Digital Elements (Cyber Resilience Act),” 2024.
[5] Executive Office of the President of the United States, “Executive Order 14028: Improving the Nation’s Cybersecurity,” 2021.
[6] M. Feng, Z. Yuan, F. Li, G. Ban, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu, A. Piao, J. Xue, and W. Huo, “B2SFinder: Detecting Open-Source Software Reuse in COTS Software,” Proc. ASE, pp.1038–1049, 2019.
[7] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, “ModX: Binary Level Partially Imported Third-Party Library Detection via Program Modularization and Semantic Matching,” Proc. ICSE, pp.1393–1405, 2022.
[8] S. Li, Y. Wang, C. Dong, S. Yang, H. Li, H. Sun, Z. Lang, Z. Chen, W. Wang, H. Zhu, and L. Sun, “LibAM: An Area Matching Framework for Detecting Third-Party Libraries in Binaries,” ACM Trans. Softw. Eng. Methodol., vol.33, no.2, pp.52:1–52:35, 2024.
[9] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, “BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching,” Proc. ICSE, pp.224:1–224:13, 2024.
[10] Y. Zou, Y. Zhang, G. Zhao, Y. Wu, S. Shen, and C. Fu, “BinCoFer: Three-stage Purification for Effective C/C++ Binary Third-Party Library Detection,” J. Syst. Softw., vol.229, p.112480, 2025.
[11] B. Xia, D. Zhang, Y. Liu, Q. Lu, Z. Xing, and L. Zhu, “Trust in Software Supply Chains: Blockchain-Enabled SBOM and the AIBOM Future,” Proc. EnCyCriS/SVM Workshop, pp.12–19, 2024.
[12] OpenSSF SBOMit Project, “SBOMit Specification”. https://sbomit.dev, Accessed: June 23, 2026.
[13] CVE Binary Tool Developers, “cve-bin-tool”. https://github.com/intel/cve-bin-tool, Accessed: June 23, 2026.
[14] D. Pereira, C. Molloy, S. Acharya, and S.H.H. Ding, “Automating SBOM Generation with Zero-Shot Semantic Similarity,” Proc. ICSPIC, pp.65–77, 2025.
[15] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, “LibvDiff: Library Version Difference Guided OSS Version Identification in Binaries,” Proc. ICSE, pp.1–12, 2024.
[16] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, “jTrans: Jump-Aware Transformer for Binary Code Similarity Detection,” Proc. ISSTA, pp.1–13, 2022.
[17] Y. Gu, H. Shu, F. Kang, and F. Hu, “UniASM: Binary Code Similarity Detection Without Fine-Tuning,” Neurocomputing, vol.630, p.129646, 2025.
[18] National Security Agency, “Ghidra Software Reverse Engineering Framework”. https://github.com/NationalSecurityAgency/ghidra, Accessed: June 23, 2026.
[19] Hex-Rays SA, “IDA Pro Disassembler and Debugger”. https://hex-rays.com/ida-pro/, Accessed: June 23, 2026.