L-010

関数名に着目した静的ライブラリと実行ファイルの依存関係解析手法

Dependency analysis method from an executable file to static libraries focusing on function names

谷 知哉 †
Tomoya Tani
山内 利宏 ††
Toshihiro Yamauchi

1. はじめに

IoT 機器の普及に伴い，これらを標的とした攻撃が増加している．効果的なセキュリティ対策には，網羅性の高い SBOM（Software Bill of Materials）の活用によるソフトウェア構成要素の正確な把握が求められる．特に，静的ライブラリは多くの IoT 機器に含まれており，静的ライブラリがリンクされた実行ファイルとの依存関係を把握することは，脆弱性管理に不可欠である．しかし，静的リンクされた実行ファイルはライブラリとの依存関係を示す情報を保持しないため，特定が困難であるという課題がある．

本稿では，この課題への対処として，関数名に着目した依存関係解析手法を提案する．また，提案手法により，依存関係の特定に有用な情報が取得可能であることを示す．

2. IoT 機器ファームウェアの SBOM 作成と課題

本研究では，IoT 機器のファームウェアにおいてソースコードやパッケージマネージャが管理に用いるメタデータが削除されることが多いため，ソフトウェア構成の特定にバイナリのみを対象とした解析を想定する．静的ライブラリに着目し，静的ライブラリが含まれる場合の調査と，既存の SBOM 生成ツールでの静的ライブラリの依存関係の解析機能を調査した．

まず，先行研究 [1] と同じ手法で 1,893 件の IoT 機器のファームウェアを調査した結果，508 件のファームウェアに合計 1,376 件の静的ライブラリが含まれることが確認された．

次に，市販または OSS として広く利用されている SBOM 生成ツール計 5 件を調査したところ，バイナリからの SBOM 作成機能は備えているものの，静的ライブラリとこれをリンクしている実行ファイルとの依存関係をファイル単位で解析する機能は確認できなかった．このため，SBOM において実行ファイルと静的ライブラリの依存関係が正しく反映されず，網羅性や正確性が損なわれる．

3. 静的ライブラリの依存関係解析手法

3.1 静的ライブラリの依存関係解析における課題

（課題 1）静的リンクで生成された実行ファイルが使用するライブラリを特定することは困難である．これは，静的リンクがビルド時に対象のライブラリを実行ファイルに組み込む方式であり，リンク元のライブラリ名などの情報が実行ファイル内に明示的に残らないためである．

† 岡山大学大学院環境生命自然科学研究科, Okayama University
†† 岡山大学学術研究院環境生命自然科学学域, Okayama University

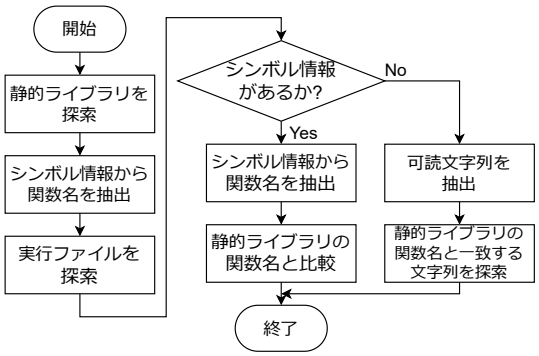

図 1　提案手法の流れ

（課題 2）解析の負荷が大きいという課題がある．IoT 機器には複数の実行ファイルと静的ライブラリが含まれており，それぞれに対して解析が必要となる．また，バイナリ間の関数のマッチングを行う手法も存在するものの，逆アセンブルや特徴抽出などの高度な解析処理が求められるため，処理時間が長く，環境構築の負荷も大きい．

3.2 関数名に着目した依存関係解析手法

本研究では 3.1 節で示した（課題 1）への対処として，実行ファイル内に存在する関数名の情報から，静的ライブラリとの依存関係を推定する手法を提案する．提案手法では，バイナリの表層情報を用いた解析を行い，関数名の取得には一般的なコマンドを用いる．このため，解析のコストが低く，専用の環境を必要とせず，（課題 2）にも対処できる．

図 1 に提案手法の流れを示す．提案手法では，関数名の取得と比較に以下の 2 つの手法を用いる．

シンボル情報の抽出: バイナリにシンボル情報が保持されている場合，これから関数名を抽出する．具体的には実行ファイル，および静的ライブラリに対して `nm` コマンドを適用する．出力されるシンボル情報のうち，テキストセクションに定義された関数に対応するシンボルのみを抽出し，両者を比較する．

一方，動的ライブラリから参照する関数はバイナリ内に定義が存在しないため，`nm` コマンドの出力では種別が未定義として表示される．本手法では，種別が未定義の関数を抽出対象から除外することで，動的ライブラリと実行ファイルの依存関係を誤って検出することを防ぐ．

可読文字列の抽出: 実行ファイルはサイズの削減，およびセキュリティ対策のためにシンボル情報が削除されていることがあり，この場合はシンボル情報の抽出が行えない．そこで，本手法では，実行ファイルに含まれる可読文字列から関数名を抽出する．具体的には，実行ファイルに `strings` コマンドを適用して可読文字列を抽出し，静的ライブラリから抽出した関数名と一致する文字

表 1　提案手法適用結果（シンボル情報有）

実行ファイル名	静的ライブラリ名	関数総数	一致した関数の数	一致率 (%)
ldapwhoami	liblber.a	148	146	98.6
	libldap.a	517	444	85.9
	libldap_r.a	595	444	74.6
ldapsearch	liblber.a	148	146	98.6
	libldap.a	517	456	88.2
	libldap_r.a	595	456	76.6

表 2　提案手法適用結果（シンボル情報無）

実行ファイル名	静的ライブラリ名	関数総数	一致した関数の数	一致率 (%)
ldapwhoami	liblber.a	148	98	66.2
	libldap.a	517	219	42.4
	libldap_r.a	595	220	37.0
ldapsearch	liblber.a	148	98	66.2
	libldap.a	517	232	44.9
	libldap_r.a	595	233	39.2

列を探索する．関数名の一致判定には以下の正規表現を適用した．

```
\b<function>\b
```

\b は単語境界を意味しており，対象の関数名が単語単位で出現する場合のみ一致しているとみなす．

一方，実行ファイル中の可読文字列には，動的ライブラリから参照する関数名も含まれている場合がある．これらの関数名は静的ライブラリの依存関係とは無関係であるため，誤検知を防ぐために抽出対象から除外する必要がある．本手法では，まず strings により抽出した可読文字列と静的ライブラリの関数名とを照合し，候補を抽出する．その後，objdump -R により得られた動的リンクの関数名と一致する候補を除外することで，静的リンクによる依存関係のみを判定対象とする．

4. 提案手法の評価

4.1 評価の目的

提案手法を実際の IoT 機器のファームウェアに含まれる静的ライブラリと実行ファイルを対象に適用し，得られる解析結果が静的ライブラリの依存関係の推定に有用か否かを明らかにする．

4.2 評価対象

ファームウェアのアンパックとファイルシステムの抽出には文献 [1] の Firmware Analyzer を用いた．評価対象の IoT 機器のファームウェアは，MIPS32 向けにビルドされており，ファイルシステム内には静的ライブラリが 3 件，実行ファイルが 79 件含まれていた．このうち，シンボル情報を保持していた実行ファイルは 33 件，保持していなかった実行ファイルは 46 件であった．

なお，評価は，ホスト OS として Windows 11 Pro を使用し，VirtualBox 7.0 上に構築した Ubuntu 18.04.6 LTS のゲスト OS 環境上で実施した．CPU は Intel Core i9-14900K（3.20GHz），メモリは 16GB を割り当てた．

4.3 評価結果

提案手法を評価対象に適用し，静的ライブラリの関数名と実行ファイルの関数名の一致する割合（一致率）を解析した．評価結果より，以下のことが分かる．

(1) 高い一致率を示した静的ライブラリと実行ファイルの組を**表 1** に示す．他の実行ファイルについて，一致する関数名が検出されなかった実行ファイルが 70 件，1〜2 個のみ一致した実行ファイルが 7 件であった．この結果から，静的ライブラリと依存関係がある実行ファイル（シンボル情報有）の場合，関数名の一致率の高さで，依存関係の有無を判断できると推察できる．

(2) シンボル情報が無い場合の評価を行うため，表 1 の実行ファイル（シンボル情報有）に strip コマンドを適用し，シンボル情報を削除した後に可読文字列の抽出による依存関係の解析を行った結果を**表 2** に示す．表 2 より，シンボル情報無の場合，シンボル情報有の場合より，関数の一致率が低下することが確認できた．

(3) readelf -d コマンドを表 1 の実行ファイルに適用し，リンクされている動的ライブラリを調査した結果，OpenLDAP の動的ライブラリに関する情報は得られなかった．このため，表 1 に示す静的ライブラリは実行ファイルに静的リンクされていると推察できる．

(4) 表 1 と表 2 において関数名が一致しなかった関数について，静的にリンクされているか否かを調査した．調査は，同一バージョンの OpenLDAP 2.4.35 をインストールし，一致しなかった関数を静的ライブラリのソースコードから削除し，ビルドを行い，エラーの有無で判断した．表 1 において一致しなかった関数を削除した場合，正常にビルドが完了した．一方，表 2 の場合，静的ライブラリのビルド時にエラーが発生した．以上のことから，可読文字列の抽出による依存関係の解析は，実行ファイルに組み込まれている関数であっても一致しないと判断される可能性があることが分かった．

(5) 可読文字列の抽出による関数名の一致について調査した．表 2 のシンボル情報無の実行ファイルは，関数名がソースコードにエラーメッセージやデバッグ用等の文字列として埋め込まれている場合がある．これらの埋め込まれた文字列を抽出することで，関数名との一致が確認でき，一致率が向上することを確認した．

(6) 提案手法を評価対象のファイルシステムに適用した結果，処理時間は 17.21 秒を要した．提案手法は，実行ファイルと静的ライブラリの間で一対一の比較を行っており，組み合わせは計 237 通りとなる．この処理時間は，組み合わせ数に対して十分に短く，実環境での利用においても実用的な範囲であると推察できる．

5. おわりに

本稿では，実行ファイルと静的ライブラリの依存関係を関数名の一致に基づいて推定する手法を提案した．評価により，依存関係の特定に有用な情報が取得可能であること，シンボル情報の有無が関数名の取得精度に影響を与えることを明らかにした．

謝辞　本研究の一部は，JST【経済安全保障重要技術育成プログラム】【JPMJKP24K2】の支援を受けたものです．

参考文献

[1] Akiyama, M., Shiraishi, S., Fukumoto, A., Yoshimoto, R., Shioji, E., Yamauchi, T.: Seeing is Not Always Believing: Insights on IoT Manufacturing from Firmware Composition Analysis and Vendor Survey, Computers & Security, (2023). https://doi.org/10.1016/j.cose.2023.103389.