修士学位論文

題目

パッケージ間の依存関係を有する SPDX ファイル
自動生成ツールの開発

指導教員
肥後 芳樹 教授

報告者
田邉 傑士

令和 5 年 2 月 1 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

令和 4 年度 修士学位論文

パッケージ間の依存関係を有する SPDX ファイル自動生成ツールの開発

田邉 傑士

内容梗概

近年，ソフトウェアのサプライチェーン上のリスクを管理するため SBOM（Software Bill of Materials）を利用しようといいう動きが進んでいる．SBOM の形式の 1 つとして SPDX（Software Package Data Exchange）がある．SPDX は現在最も利用が進められている SBOM の形式である． しかし，既存の SPDX ファイル自動生成ツールはパッケージ間の依存関係に関する情報を取得できなかったり，ソフトウェアライセンスを解析しなかったりと，ソフトウェアライセンスや脆弱性を特定するために利用するには不十分な SPDX ファイルを生成する．そこで本研究では，Debian パッケージを対象にパッケージ間の依存関係を推移的に解析し，なおかつ，ソフトウェアライセンスも解析する SPDX ファイル自動生成ツール debiantospdx を開発した．現在，既存の SPDX ファイル生成ツールの中に Debian パッケージに対応しており，パッケージ間の依存関係も解析できるオープンソースのツールはない．開発したツールを用いることで Debian パッケージ間の依存関係を推移的に解析し依存関係を含む SPDX ファイルを生成することができた．また，開発したツールでパッケージ間の依存関係を調査すると，ツールがない場合に比べて依存関係特定のための工数を大幅に削減すること，および，他の調査方法で多く出現する不要な依存関係を削除できることを確認した．本論文ではツールの作成とその実証について示す．

主な用語

SBOM
SPDX
OSS
依存関係
ソフトウェアライセンス
脆弱性

目 次

1 はじめに

近年ソフトウェアの機能複雑化に伴い，ソフトウェアの再利用はごく一般的に行われており，ソフトウェア開発の際に OSS などのサードパーティ製のソフトウェアの利用される機会はますます増加している．Contrast Security 社によると、今日のソフトウェアの大部分（79 %）にサードパーティ製のライブラリが使用されている [12]．サードパーティ製のソフトウェアを利用するここと で開発にかかる費用と時間を削減することができる．しかしその一方で，サードパーティ製のソフトウェアとの依存関係がソフトウェアライセンス（以降単にライセンス）や脆弱性などの問題を引き起こす潜在的なリスクをソフトウェアに持ち込むことがある．

例えば，サードパーティ製のソフトウェアを利用する際，サードパーティ製のソフトウェアとの依存関係があるソフトウェアを利用する際にはライセンスに注意する必要がある．ライセンスとはソフトウェアの利用者がそのソフトウェアを利用する上で守るべき事項のことで，もしライセンスに違反すると，法的紛争に発展する場合がある [2]．特に利用している他社のソフトウェア製品がソフトウェアライセンスに違反してサードパーティ製のソフトウェアに依存している場合には注意が必要である．例えば 2013 年にメディアプレイヤーメーカが訴えられた事例では，メディアプレイヤーメーカが利用していたソフトウェアがライセンスに違反してサードパーティ製のソフトウェアに依存していたことが原因でメディアプレイヤーメーカにバッキンガ命じられた．そして，その判決の中ではソフトウェアサプライヤがライセンスコンプライアンスにしたがっているものだと判断するのは過失であり、ソフトウェアを利用している組織の責任でサードパーティの権利を侵害していないことを確認する必要があるということが示された [4].

また，脆弱性に関した問題がある．あるソフトウェアに対して発見された脆弱性は依存関係を通して多くのソフトウェアに影響を与える場合がある．例えば 2021 年 12 月に公表された Apache Log4j の脆弱性は，Apache Log4j が多くのソフトウェアの基盤となるソフトウェアだったため，その依存関係を通して多くのソフトウェアに影響を与えた [3]．依存関係は何層にも渡って存在するため脆弱性は推移的に伝播する．そのため，Apache Log4j のようなソフトウェアではその影響範囲は甚大であり，対処には 10 年以上かかるという試算もある-[9].

このような開発されたソフトウェアにサードパーティ製のソフトウェアから持ち込まれた問題はそのソフトウェアを利用する管理者には認識しづらい．その中で SBOM を利用してソフトウェアの情報の透明性をあげライセンスや脆弱性の問題に対処しようという動きが多くの組織の中で広まりつつある [15].

SBOM（Software Bill of Materials）とはソフトウェアの機械可読なメタデータであり，ソ

フトウェアのライセンスやコピーライト，依存関係など様々な情報を 1 つのファイルにまとめることでソフトウェアを管理しやすくしようとするものである [8]．そして SPDX（Software Package Data Exchange）は最も主要な SBOM の形式の 1 つである [10]．しかし，SPDX の導入に向けては，現状では課題が残る．その課題の 1 つは SPDX に関するツールの整備がまだ不十分なことである [15]．特に SPDX 形式で書かれたファイル（以降 SPDX ファイル）を自動生成する無償ツールの機能や精度については実務者からも不安の声が上がっている [13]．

そこで本研究ではより完全な SPDX ファイルの自動生成ツールを開発する．提案するツールは今まで推移的に依存関係を解析する既存ツールがなかった Linux の Debian パッケージを対象とする．本研究では Debian パッケージに対して推移的な依存関係を解析して，SPDX ファイルを生成することによる効果を，手動で推移的な依存関係を調査する場合や簡易ツールで推移的な依存関係を調査する場合と比較した．その結果，提案ツールが依存関係特定のための工数を大幅に削減すること，および，他の調査方法で多く出現する不要な依存関係を削除できることを確認した．これらの結果はライセンスや脆弱性などの問題への対処に寄与する．

以降，2 章では本研究の背景として SBOM や SPDX について述べる．3 章では既存ツールとの比較から見える本ツールの特徴と Debian パッケージを対象とした理由，機能の実装方法の詳細について述べる．4 章では提案ツールを用いた実証と評価について述べる. 5 章では本研究のまとめと今後の課題について述べる．提案ツールは PyPI 上で公開されており，pip でインストールし Debian 系の OS の上で実行することができる．また，そのソースコードは `https://github.com/tk-tanab/debiantospdx` で公開している．

2 背景

2.1 SBOM

SBOM とは「Software Bill of Materials」の略称であり，ソフトウェアの機械可読なメタデータのことを指す．ここでいうメタデータとは，ソフトウェアのコンポーネントとその依存関係，およびライセンスデータなどであり，SBOM にはそれらが一意に識別できる形式でまとめられる．SBOM を記述する形式には後述する SPDX をはじめ CycloneDX や SWID などいくつかの形式が存在する．

SBOM が注目されるようになったきっかけから現在に至るまでの流れを説明する．2020 年に SolarWinds 社のソフトウェアである Orion platform の脆弱性を利用して，同ソフトウェアを利用していたアメリカの政府機関を含む多くの組織が攻撃を受けていたことが発覚した-[1][14]．この件を発端としてソフトウェアのサプライチェーンを管理しようとする動きはますます大きくなり，2021 年 5 月に「Executive Order on Improving the Nation's Cybersecurity」という大統領令がアメリカのバイデン大統領によって署名された [10]．この大統領令の「Sec. 4. Enhancing Software Supply Chain Security.」の項ではソフトウェアの配布者に SBOM の提供を要求している．そして，このことが要因となって現在では日本の経済産業省を含む様々な組織で SBOM の利用が検討されている．

大統領令の中で，SBOM は「SBOM は，ソフトウェアを開発または製造する人，ソフトウェアを選択または購入する人，ソフトウェアを運用する人に有用である．開発者は，オープンソースやサードパーティのソフトウェアコンポーネントを利用して製品を作ることが多い．SBOM を利用することで，これらのコンポーネントが最新であることを確認し，新しい脆弱性に迅速に対応することができる．購入者は，SBOM を使用して，製品のリスクを評価するために使用できる脆弱性解析やライセンス解析を行うことができる．ソフトウェアを運用する人は，SBOM を使用して，新たに発見された脆弱性の潜在的なリスクにさらされているかどうかを迅速かつ容易に判断することができる.」と説明されており，ソフトウェアを利用する様々な場面で多くのステークホルダーにとって有用なものになると期待されている．

2.2 SPDX

SPDX とは「Software Package Data Exchange」の略称であり，SBOM を記述するための形式の 1 つである [11]．LinuxFoundation が主体となって作成しており，ライセンスやセキュリティ，およびその他の関連情報を含むソフトウェアの情報を伝達するための標準化されたフォーマットである [8].

元々，SPDX はオープンソースソフトウェア (以降 OSS) のライセンスを遵守させることを

目的として作成された [7]．同じソフトウェアに対してライセンスの解析するという作業が重複しないようにするため，またライセンスの認識に齟齬が生じないようにするため，ライセンスを簡潔で標準化された形で表記することで時間の節約とライセンスの認識の正確性を向上させることができると考えられたからである．このライセンスを簡潔な形で表記したものはライセンスリストとして管理されている．現在では 2021 年 7 月に先の大統領令を受けて NTIA（アメリカ合衆国国家電気通信情報管理庁）が定めた SBOM の最小要素に対応し，依存関係なども詳しく記述できるように改良されている [11]．SBOM の最小要素の内容については 3.1 節で説明する．そして SPDX はその SBOM の最小要素が示されてから最初に ISO/IEC JTC1 標準として認められた SBOM の形式であり，現在最も有名な SBOM の形式である [5].

2.2.1 SPDX の例

SPDX ファイルは TagValue 形式や RDF 形式，JSON 形式など様々な形式を取ることが可能となっている．TagValue 形式で書かれた SPDX ファイルの例として cpio パッケージについての SPDX ファイルの冒頭部分を図 1 に示す．SPDX ファイルは多くの情報を記載できるため長くなることが多い．この例でも総行数は 121 行であったため，そのすべてを載せることはできず，後半のファイル情報のブロックを割愛している．TagValue 形式の SPDX ファイルは空行でブロックに分けられている．またその中で各項目に関する情報は「フィールド名：値」という形式で表される．各ブロックは SPDX ファイル自体の情報，SPDX ファイルの作成に関する情報，解析したパッケージの 1 つに関する情報，解析したパッケージに含まれるファイルの 1 つに関する情報など，1 つのブロックが 1 つの対象についての情報に対応している．SPDX，特に TagValue 形式は人でも読めるように作られているため，形式を理解していなかったとしても読むことは比較的容易い．ここからは図 1 の例において各ブロックに記載されている情報をいくつか説明する．

1 つ目のブロックは SPDX ファイルの内容に対する情報が記載されている．この例からは SPDX のバージョン 2.2 で記述されていることや SPDX ファイル自体のライセンスが CC0-1.0 であることなどが読み取れる．

2 つ目のブロックは SPDX ファイルの作成に関する情報が記載されている．この例からは scancode-toolkit 31.2.4 と debiantospdx というツールを用いて SPDX ファイルが生成されていること，作成したのは Tanabe Taketo という人物であること，作成した日は 2023 年 1 月 11 日であることが読み取れる．この Creator フィールドの例から分かるように 1 つのフィールドに関する情報が複数ある場合はフィールドと値の組みを複数記述する．

3 つ目のブロックはパッケージの情報が記載されている．この例では cpio というパッケー

ジ名であることやそのバージョンが 2.13+dfsg-7 であること，含んでいるライセンスが GPL-1.0-or-later と GPL-3.0-or-later であることなどが読み取れる．

そして 4 つ目以降のブロックではパッケージを構成する各ファイルの情報が記載されている．ファイルがどこにあるかやファイルに含まれるライセンスやコピーライト，ファイルのハッシュ値などの情報を読み取ることができる．

また，SPDX ファイルと作成情報はそれぞれ 1 ブロックしか記述できないが，パッケージとファイルのブロックは複数記述することができる．つまり，複数のパッケージの情報を 1 つの SPDX ファイルに記述することもできる

```
## Document Information

SPDXVersion: SPDX-2.2
DataLicense: CC0-1.0
DocumentNamespace: http://spdx.org/spdxdocs/
cpio_2.13+dfsg-7-216cd900-0f32-51df-8268-15dc23cb93a7
DocumentName: cpio_2.13+dfsg-7
LicenseListVersion: 3.18
SPDXID: SPDXRef-DOCUMENT
ExternalDocumentRef: DocumentRef-adduser.Cycle http://spdx.org/spdxdocs/
adduser_3.118ubuntu5-203b1c8a-ddae-52cd-9f39-0d8d684e6070 SHA1:
ff4cac613229d6873a2c15770df30e81c970fefe
Relationship: SPDXRef-DOCUMENT DESCRIBES SPDXRef-cpio


## Creation Information

Creator: Tool: scancode-toolkit 31.2.4
Creator: Tool: debiantospdx
Creator: Person: Tanabe Taketo
Created: 2023-01-11T12:27:13Z


## Package

PackageName: cpio
PackageVersion: 2.13+dfsg-7
SPDXID: SPDXRef-cpio
PackageHomePage: https://www.gnu.org/software/cpio/
PackageDownloadLocation: NOASSERTION
PackageVerificationCode: 8d5a874bf101292c9539926df3ad8b763acf0038
PackageLicenseDeclared: NOASSERTION
PackageLicenseConcluded: NOASSERTION
PackageLicenseInfoFromFiles: GPL-1.0-or-later
PackageLicenseInfoFromFiles: GPL-3.0-or-later
PackageCopyrightText: <text>Copyright (c) 1990, 1991, 1992, 2001, 2003, 2004, 2005, 2006,
2007 Free Software Foundation, Inc.
</text>
PackageComment: <text>GNU cpio -- a program to manage archives of files
 GNU cpio is a tool for creating and extracting archives, or copying
 files from one place to another.  It handles a number of cpio formats
 as well as reading and writing tar files.</text>
Relationship: SPDXRef-cpio DEPENDS_ON DocumentRef-adduser.Cycle:SPDXRef-libc6


## File

FileName: /usr/share/doc/cpio/copyright
SPDXID: SPDXRef-cpio-file-0
FileChecksum: SHA1: 0acbf876b5dd8a28a44f7a5be3785961656e5aaf
LicenseConcluded: NOASSERTION
LicenseInfoInFile: GPL-1.0-or-later
LicenseInfoInFile: GPL-3.0-or-later
FileCopyrightText: <text>Copyright (c) 1990, 1991, 1992, 2001, 2003, 2004, 2005, 2006,
2007 Free Software Foundation, Inc.
</text>
Relationship: SPDXRef-cpio CONTAINS SPDXRef-cpio-file-0


FileName: /bin/cpio
SPDXID: SPDXRef-cpio-file-1
FileChecksum: SHA1: 2f438f9e2da23bbf75abf0d79a38eea641e0bee5
LicenseConcluded: NOASSERTION
LicenseInfoInFile: NOASSERTION
```

図 1: cpio パッケージの SPDX ファイルの冒頭部分

2.2.2 SBOM(SPDX) 活用の利点

SBOM を活用する大きな利点の 1 つは脆弱性発表から特定までのリードタイムの短縮と工数の削減である．この節では日本の経済産業省が SBOM 活用について実証している内容を紹介する [15]．実証に使用している SBOM の形式は SPDX である．

ここではあるソフトウェアを対象に，業務で対象のソフトウェアを使用する企業をユーザ，対象のソフトウェアを作成してユーザに売る企業をベンダ，対象のソフトウェアを構成するソフトウェア部品をベンダに提供する企業をサプライヤとするサプライチェーンで実証を行なっている．

ユーザが対象のソフトウェアに使用されている OSS を管理して脆弱性の特定およびライセンスの特定を行うという作業に対し，以下の 4 つの管理シナリオにおける工数とツール導入のための費用を比較している．

1. 手動で独自形式の OSS 一覧を作成して管理
2. 手動で SBOM を作成して管理
3. 無償ツールで SBOM を作成して管理
4. 有償ツールで SBOM を作成して管理

1・2・3 の手法ではサプライチェーンの各工程でサプライヤとベンダが使用した OSS を収集し，SBOM または独自形式の OSS 一覧を作成してユーザまで引き継ぐ．4 の手法では有償ツールが対象のソフトウェアの依存関係を解析し，使用している OSS を解析できるため，サプライヤとベンダが SBOM を作成する必要はない．また，2・3・4 の手法では脆弱性とライセンスの特定に SPDX 用の既存ツールを利用している．

コストを計測した結果が図 2 に示す表である．導入に向けた初期コストはかかるものの SBOM を利用した方が運用時のコストが低減していることがわかる．この計測において SBOM 利用時の運用コストが低くなった主な要因は SPDX を利用した既存ツールを用いているからと考えられる．そのため同様のツールを独自形式に対して開発すれば運用コストが同程度になる可能性もある．しかし，既存ツールを利用できることこそが標準化された仕様を用いて OSS を管理するメリットである．SPDX を使うことで運用するために独自ツールを開発する必要はなくなる．また今後も多くの SPDX を利用した管理ツールが開発されるため，SPDX を活用した脆弱性やライセンスの管理はますます便利になると考えられる．

2.2.3 SPDX 関連ツール

SPDX 関連ツールには 2.2.2 節で紹介した実証の中で利用されているツールを始めいくつかの種類が存在する．ここでは既存の SPDX 関連ツールの種類とその内容をいくつか紹介す

シナリオ	初期工数		運用工数				ツール費用(初期・運用)
	環境整備工数	体制構築・学習工数	SBOM(部品情報)作成工数		脆弱性特定工数	ライセンス特定工数	
			作成工数	精査工数			
①従来の部品管理(独自形式)	1.3人時間 (フォーマットの定義)	0.5人時間 (フォーマット・運用の連携)	3.0分/部品	3.8分/部品 (OSS一覧のレビュー)	6分/部品・回 (頻度:1.5カ月)	10分/部品 *別途法務チェックに2-5営業日	(なし)
②SBOM(手動作成)	102.1人時間 (活用ツール導入・処理調査、テンプレート整備) *全社の合計値	48.2人時間 (SBOM作成方法の学習、作業手順書作成) *全社の合計値	2.9分/部品	(なし) *開発者が手動作成した部品情報が正確と仮定	0分/部品・回 (頻度:即時)	0分/部品 *別途法務チェックに2-5営業日 *ライセンスの検知漏れが発生	
③SBOM(無償ツール)	71.1人時間 (作成・活用ツール導入) *全社の合計値	57.2人時間 (作業手順書作成、学習) *全社の合計値	0.2分/部品	1.4分/部品 (検知結果確認)			
④SBOM(有償ツール)	3.0人時間 (商談、スキャン用アプリ導入)	6.5人時間 (学習、開発元問い合わせ)	0.3分/部品	1.4分/部品(検知結果確認) (81.5分/部品(OSSの依存関係の解析によって新たに検知した部品に関する精査及びベンダー確認))	0分/部品・回 (頻度:即時)	0分/部品 *別途法務チェックに2-5営業日	ユーザー数やデータ量等に応じたライセンス費用が発生

図 2: 実証におけるコスト・効果に関する計測結果

る [6].

検証ツール　SPDX ファイルが SPDX に必要な要素を満たしているかを確認するツール．また，SPDX のフィールドの表記ミスや SPDX 内の参照の不一致なども指摘する．SPDX Online Tool[1]から利用できる．

形式変換ツール　SPDX ファイルの形式を相互に変換するツール．RDF 形式や JSON 形式，TagValue 形式など様々な形式を持つ SPDX ファイルの形式を変換する．現在複数の形式変換ツールがリリースされているが，変換前と変換後で依存関係情報の欠落や不一致が見られるなど，この修士論文作成時点では完全な形式変換ツールは見つけられていない．

自動生成ツール　パッケージを解析して SPDX ファイルを生成するツール．OSS などファイル構成が分かっているパッケージに対して実行することでファイルを解析しライセンスの検出や依存関係を解析して SPDX ファイルを生成する．SPDX ファイルは手動で作成することもできるが，ツールを利用することで 2.2.2 節の実証にもあるように大きな時間の節約となる．またこの実証では部品の粒度がパッケージ単位であったが，ファイル単位の場合はツールの利用の有無による要する時間の差はさらに顕著になる．FOSSology[2]や spdx-SBOM-generator[3]などいくつかの自動生成ツールが存在するが解析できるパッケージの種類や解析できる内容には違いがある．

[1]https://tools.spdx.org/app/
[2]https://www.fossology.org/
[3]https://github.com/openSBOM-generator/spdx-SBOM-generator

脆弱性確認ツール　SPDX ファイルから含まれる脆弱性を検出するツール．SPDX ファイルに含まれるパッケージ名やバージョンの情報を既知の脆弱性のデータベースと照合し，脆弱性がないかどうかを確認する．OSV-scanner や spdx-to-osv などのツールがある．

2.2.4　SPDX 活用の課題

前述のようにツールを利用することによって SPDX で OSS を管理する方が独自形式で OSS を管理するよりもコストの面で有効であると述べられている [15]．しかし，SPDX を本格的に運用していくためには現状ではまだ課題も残る．実証成果として確認されたこととして以下のような課題が挙げられていた．

- SPDX 導入に向けた環境の整備や学習が必要になる
- 既存の SPDX ツールだけでは機能が不十分である
- ツール利用のドキュメントが不足している

まず，図 2 にも示されているように SPDX を導入する上では初期工数の大きさが障害となる．実証のシナリオでは最終的な SPDX の利用者はソフトウェアのユーザだが，依存関係を解析できない無償ツールを利用している場合や手動で SPDX を作成している場合はユーザがソフトウェアに使用されている OSS の情報を得るために，SPDX ファイルをサプライヤからベンダにベンダからユーザに引き継いでもらう必要がある．そのため，その場合はサプライヤとベンダの各々が SPDX について学習し，環境の整備や体制を整える必要が生じる．したがってサプライチェーン全体での初期工数の合計はかなり大きくなってしまう．また，脆弱性の特定やライセンス特定の頻度が大きくない場合は全体の工数に占める初期工数の割合が高くなり，SPDX の活用がコスト面で有効であるとは言えなくなる可能性もある．

また，利用する SPDX ツールの機能の問題がある．この実証で利用していたツールを含め，既存のツールが生成する SPDX ファイルは NTIA の定めた SBOM の最小要素を満たしていない部分がある．また対象ソフトウェアにも制限があるなど既存のツールだけでは SPDX の利用環境を整えるには不十分な点がある．ここについては 3.1 節で後述する．加えてソフトウェアの実務者や 2.2.2 説の実証に携わった人々からは既存の OSS の SPDX ツールに対して精度面に不安があるという声があがっている [13][15]．実際に先ほどの実証でも使われている Syft という OSS ツールを実際に使ってみたところライセンスの検出にも一部誤りが見られ，ファイルのハッシュ値がすべて 0 になっているなど，問題が見られるところもあった．

そして，ツールや SPDX の利用のしやすさも問題となる．実証で確認されたこととしてドキュメントが不十分であることが挙げられていた．これは初期工数における学習工数の大きさに関わる．

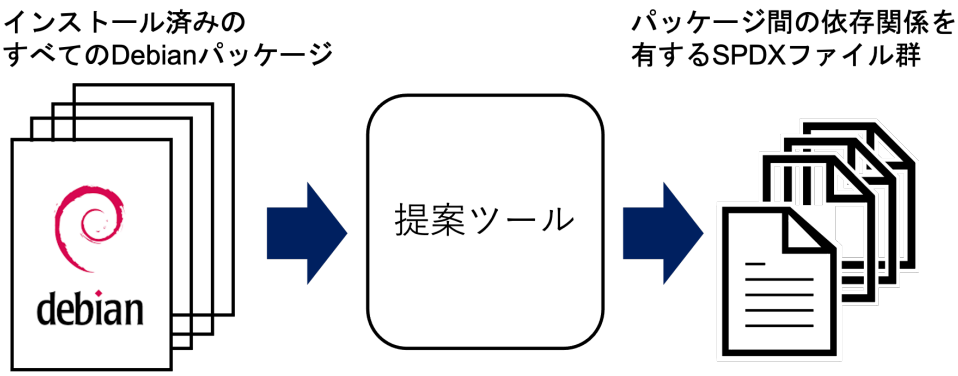

図 3: 提案ツールの概要

3　提案ツール

本研究では SPDX 活用の課題の中でも SPDX ツールの機能の問題を踏まえて，SPDX に必要とされる要素を可能な限り満たす SPDX ファイルを自動生成するツール debiantospdx を開発した．図 3 はツールの概要を示したものである．提案ツールはインストール済みのすべての Debian パッケージを，依存関係を含めて解析し，SPDX ファイルを生成するコマンドラインツールである．また，SPDX ファイルの管理ツールがない場合の代用として出力した SPDX ファイルからパッケージの情報を抽出する機能も持つ．

提案ツールのユーザは Debian パッケージを含むシステムを管理・運用する人を想定しており，使用している Debian パッケージに既知の脆弱性がないかライセンス上の問題がないか確認する場合，または新しく報告された脆弱性の影響をどの Debian パッケージが受ける可能性があるかを確認するに提案ツールで生成した SPDX ファイルが有効に働くと考えている．

ツールは Python で記述されており，PyPI に公開されている．pip install debiantospdx でインストールすることができる．

3.1　特徴と既存ツールとの比較

本節では提案ツールの特徴を 4 つ説明する．

まず，提案ツールの特徴の 1 つ目は対象のパッケージが Debian パッケージであることである．Debian パッケージを対象とした理由の 1 つは私が確認した限り，既存の SPDX ファイルの自動生成ツールの中に，インストールされている Debian パッケージを対象としてその依存関係情報を含む SPDX ファイルを生成するツールがなかったからである．そして 2 つ目の理由は Debian パッケージを厳密に管理するためには提案ツールが生成するような依存

関係情報を含む SPDX ファイルを用いるのが適当だと考えたためである．これについては 3.2.4 節で詳細を説明する．

提案ツールの特徴の 2 つ目は複数のパッケージを一度に解析するということである．既存ツールの多くは指定された 1 つのパッケージまたはそのパッケージと依存先のパッケージを解析して SPDX ファイルを生成する．しかし，提案ツールが運用される状況を考えたとき，それだけでは不十分だと考えた．ソフトウェアを運用する人にとっては 1 つのパッケージに問題があるかどうかも重要だが，管理するすべてのパッケージが問題なく利用し続けられるかどうかもまた重要である．そこでメインの機能の 1 つとしてインストール済みのすべての Debian パッケージを依存関係を含めて解析して SPDX ファイルを生成する機能を実装した．

提案ツールの特徴の 3 つ目はパッケージ単位で SPDX ファイルを生成するということである．依存関係を推移的に解析できる SPDX ファイルの自動生成ツールを含め，既存の SPDX ファイル自動生成ツールは解析したすべてのパッケージに関する情報を 1 つの SPDX ファイルにまとめて出力する．しかし，提案ツールのように，インストール済みのすべての Debian パッケージを解析して SPDX ファイルを生成する場合に同様の出力をしようとすると，すべてのパッケージの情報を含む非常に巨大な SPDX ファイルを生成することになる．また，SPDX ファイルはツールで分析するだけでなく人間が読むことも想定されているため，SPDX ファイルのサイズが大きくなりすぎると人間には情報が辿りずらくなってしまう．そこで，提案ツールではできる限り 1 つの SPDX ファイルに記述されるパッケージの数が少なくなるように情報を分割してファイルを生成するようにした．

そして，提案ツールの特徴の 4 つ目は SPDX ファイルに必要とされる情報を可能な限り解析して SPDX ファイルを生成しているということである．SBOM が最低限満たすべき条件として NTIA が定めた SBOM の最小要素というものがある．その中で SBOM に記載すべき情報として以下の項目が挙げられている [11].

- サプライヤ名
- パッケージ名・バージョン
- 固有識別子
- 依存関係
- SBOM の作者名
- SBOM の作成時刻

また，SPDX が作られた元々の目的は OSS のライセンスの遵守であることから SPDX においてはライセンス・コピーライトの情報も重要な項目である．これは経済産業省の資料に記載

項目	debiantospdx	FOSSology	SBOM-spdx-generator
サプライヤ名	×	×	×
パッケージ名・バージョン	○	×	○
固有識別子	○	○	×
依存関係	○	×	○
SBOM の作者名	○	○	×
SBOM の作成時刻	○	○	○
ライセンス・コピーライト	○	○	×

表 1: 生成する SPDX ファイルが含む情報と既存ツールとの比較

されていた，企業が SPDX の導入の目的とするところの 1 つでもあり，実際に OSS のライセンス・コピーライトを明らかにすることは重要視されている [15]．したがって，OSS の SPDX ファイルを生成するのであればその SPDX ファイルには前述した最小要素とライセンス・コピーライトの情報が記述されていることが望ましいと言える．そこで私はツールが生成する SPDX ファイルにはできる限りそれらの情報を記載するようにした．既存の SPDX ファイル自動生成ツールの FOSSology と SBOM-spdx-generator，そして提案ツールが生成する SPDX ファイルが含む情報の比較を表 1 に示す．この表では生成されたファイルに項目が存在していても，NOASSERTION になっているなど情報が表示されていないものに関しては×にしている．また SBOM-spdx-generator の SBOM の作者名の項目には SBOM-spdx-generator というツール名のみが表示され，実行時に作成者の名前を指定することはできない．この表に示すように，比較対象にした SPDX ファイル自動生成ツールに比べて提案ツールがより情報を網羅している．サプライヤ名に関してはツールの利用者の環境からでは不明であり，パッケージからもサプライヤ名の情報を取得することができないために記載できなかった．

3.2 Debian パッケージ

提案ツールの対象とした Debian パッケージとは Debian や Ubuntu などの Debian 系 Linux ディストリビューションで使われるソフトウェアパッケージである．Debian ポリシーマニュアルによってディストリビューションに含まれるための要件が示されており，dpkg や apt などの専用のパッケージ管理ツールによって管理される．本章では Debian パッケージを構成するファイルの中で提案ツールで利用するファイルと Debian パッケージ間の依存関係の取得方法を示し，提案ツールが Debian パッケージを対象とした理由を説明する．

3.2.1 control ファイル

control ファイルとは Debian ポリシーマニュアルによって示された Debian パッケージが含んでいなければならないファイルの 1 つであり，パッケージ管理ツールがパッケージを制御・管理するために必要な情報が記述される．

control ファイルの例を図 4 に示す．control ファイルでは情報を「フィールド名： 値のリスト」の形で表す．値のリストはコンマで区切られる．例えば Package フィールドはパッケージ名，Version フィールドはパッケージのバージョンを表す．この例ではパッケージ名が python3，バージョンが 3.8.2-0ubuntu2 ということが読み取れる．

ここからは Debian パッケージをインストールまたは実行する上で依存関係を解決する際に利用されるいくつかのフィールドについて説明する．説明上 control ファイルを持つパッケージの名前を A，フィールドの値のリストの 1 つとなるパッケージ名または仮想パッケージ名を B とする．

Debian パッケージの依存関係を表す Depends，Recommends，Suggests，Pre-Depends の 4 種類のフィールドについて説明する．まず，Depends はパッケージ A を実行するためにはパッケージ B がインストールされていなければいけないような依存関係を示すフィールドである．次に，Recommends はパッケージ A を利用する上で，存在しなければ実行できなくなるわけではないが，多くの人が必要とする拡張機能を有効にするためにパッケージ B が必要となるような依存関係を示すフィールドである．さらに，Suggests はパッケージ A を利用する際に必ずしもあった方が良いわけではないものの，あればより便利になる拡張機能を有効にするためにパッケージ B が必要となるような依存関係を示すフィールドである．そして，Pre-Depends はパッケージ B がなければパッケージ A をインストールすることさえできないような依存関係を示すフィールドである．

依存関係に影響を与える Replaces と Provides というフィールドについて説明する．まず，Replaces はパッケージ B の機能をパッケージ A で完全に置き換えることができることを示すフィールドである．すなわちこのフィールドが存在する場合，パッケージ B を依存先にとる依存関係をパッケージ A で代替することができる．次に Provides はパッケージ A が仮想パッケージ B を提供することを示すフィールドである．仮想パッケージとは複数のパッケージに共通するサービスに名前をつけたものであり，Debian パッケージは Depends，Recommends，Suggests，Pre-Depends を用いて依存先に仮想パッケージを指定することができる．このフィールドがある場合，仮想パッケージ B が依存先として要求された場合にパッケージ A でその依存関係を満たすことができる．

```
Package: python3
Source: python3-defaults
Version: 3.8.2-0ubuntu2
Architecture: amd64
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Installed-Size: 189
Pre-Depends: python3-minimal (= 3.8.2-0ubuntu2)
Depends: python3.8 (>= 3.8.2-1~), libpython3-stdlib (= 3.8.2-0ubuntu2)
Suggests: python3-doc (>= 3.8.2-0ubuntu2), python3-tk (>= 3.8.2-1~), python3-venv (>=
3.8.2-0ubuntu2)
Replaces: python3-minimal (<< 3.1.2-2)
Provides: python3-profiler
Section: python
Priority: optional
Multi-Arch: allowed
Homepage: https://www.python.org/
Description: interactive high-level object-oriented language (default python3 version)
 Python, the high-level, interactive object oriented language,
 includes an extensive class library with lots of goodies for
 network programming, system administration, sounds and graphics.
 .
 This package is a dependency package, which depends on Debian's default
 Python 3 version (currently v3.8).
Original-Maintainer: Matthias Klose <doko@debian.org>
```

図 4: control ファイルの例

3.2.2 copyright ファイル

copyright ファイルとは，control ファイルと同じく，Debian ポリシーマニュアルによって示された Debian パッケージが含んでいなければならないファイルの 1 つであり，パッケージ内に含まれるすべてのファイルのライセンスとコピーライトが記述される．

3.2.3 依存関係の取得方法と課題

本研究の重要な点の 1 つである依存関係の取得について Debian パッケージの依存関係を取得する既存の方法を 3 つ紹介し，それぞれの課題についても触れる．

1 つ目は apt depends コマンドを利用する方法である．apt は Ubuntu であれば標準でインストールされているツールであり，新たにパッケージをインストールする必要なく利用することができる.「apt depends パッケージ名」という形でコマンドを実行することにより，指定したパッケージが持つ依存関係を表示する．しかしながら表示されるのは 1 階層のみであり，推移的な依存関係は得られない．また，このコマンドの利用目的はそのパッケージをインストールした際にどんなパッケージがインストールされるかを示すことである．そのため，現在どのような依存関係が存在するかを反映しない．表示される依存関係はあくまで依存先の候補に過ぎず実際に依存先として表示されたパッケージがインストールされているかは分からない．加えて，パッケージのどのバージョンがインストールされているかも不明である．

2 つ目は apt-rdepends コマンドを利用する方法である．apt-rdepends コマンドは現在 Ubuntu

に標準でインストールされているパッケージではないため，apt-rdepends パッケージをインストールすることで初めて利用できるようになる.「apt-rdepends パッケージ名」という形でコマンドを実行することにより，指定したパッケージの推移的な依存関係を表示する．しかしながら，仮想パッケージにたどり着くと，それ以上依存関係を追跡しなくなる．さらに apt depends コマンドと違い仮想パッケージを提供しているパッケージの候補も表示しない．また，表示される依存関係はあくまで候補に過ぎないということは apt depends コマンドと同じであり，実際に依存先として表示されたパッケージがインストールされているかどうかと，パッケージのどのバージョンがインストールされているかは出力結果からは分からない．さらにこのコマンドが表示する依存関係は Depends と Pre-depends にあたるパッケージだけであり，Suggests や Recommends については表示しない.

3 つ目は control ファイルの情報を確認する方法である.「apt show パッケージ名」もしくは「dpkg -s パッケージ名」という形でコマンドを実行することにより，指定したパッケージの control ファイルに記載されている情報を取得することができる．apt show コマンドはパッケージをインストールする際に参照する情報を表示するので実際にそのパッケージがインストールされていなくても使用できるが，dpkg -s コマンドは実際にインストールされているパッケージの状態を表示するのでパッケージがインストール済みでないと使用できないというところに違いがある．これらのコマンドを利用して情報を表示した後で 3.2.1 節で説明した依存関係に関連するフィールドを読み取ることで依存関係を確認する．しかしながらこのようにして取得した情報も apt コマンドや apt-rdepends コマンドと同じく，実際に依存先として表示されたパッケージがインストールされているかどうかと，パッケージのどのバージョンがインストールされているかは分からない.

3.2.4 Debian パッケージを対象とした理由

Debian パッケージを対象にした理由は 2 つあり，1 つは既存の SPDX ファイル自動生成ツールの中に Debian パッケージを対象としたものがなかったからである．そして 2 つ目の理由は Debian パッケージの推移的な依存関係を取得するには手間を要するため，Debian パッケージを管理する上で提案ツールが生成する SPDX ファイルを利用するのが適当であると考えたからである．本節ではなぜパッケージの管理に依存関係の取得が必要だと考えているかを改めて説明し，なぜそれが Debian パッケージでは手間を要するかを説明する.

例えば Debian パッケージを含むシステムを管理している場合に，ある Debian パッケージに関する新たな脆弱性がインターネット上で報告されているのを見つけたときを考える．該当する Debian パッケージがインストール済みのパッケージの中に存在しなければそれ以上対処に追われることはない．しかし，もし該当する Debian パッケージが見つかった場合は

そのパッケージが影響する範囲も考える必要がある．一般にパッケージが持つ脆弱性はそのパッケージを依存先に持つパッケージにも影響を与える可能性があるからである．すなわち厳密に脆弱性の影響を考慮する場合には以下の 2 つの方法のいずれかを行う必要がある．

1. 脆弱性を持つパッケージの依存関係を逆に辿る
2. インストール済みのすべてのパッケージの依存関係を推移的に辿り，その中に脆弱性を持つパッケージがないかを確かめる

しかし，Debian パッケージにおいてはどちらの方法も簡単ではなく，手動で行うことは実現性が低い．3.2.3 節で説明したように既存の方法で得られる Debian パッケージの依存関係はインストールする際にどのようなパッケージが必要になる可能性があるかがわかる程度でどのパッケージのどのバージョンが入っているかまでは分からない．そのため実際に現在構築されている依存関係を正確に知るためには出てきた依存関係の候補のそれぞれに対して，実際に存在するか，それはバージョンの制約を満たしているか，また依存関係を満足するパッケージが見つからなかった場合は Replace するパッケージがないか，さらに仮想パッケージであった場合にはどのパッケージから提供されているかを確認する必要がある．そのため，推移的な依存関係を手動で辿るためには多くのコマンドの実行や手間を要する．これは依存関係を逆に辿る場合も同じである．そこで，現在構築されている依存関係のみを正確に捉えるもっと良い方法の 1 つとして．SBOM（SPDX）を利用したパッケージの管理があげられるのではないかと考えた．

3.3 SPDX ファイルの生成機能

提案ツールは Debian パッケージを依存関係を辿りながら推移的に解析し，解析したすべてのパッケージの SPDX ファイルを生成する．本節では生成される SPDX ファイルの仕様とその機能の処理手順と実装方法について説明する．

3.3.1 生成する SPDX ファイルの仕様

提案ツールが生成する SPDX ファイルのフィールドとその説明を表 2 と表 3 に示す．生成する SPDX ファイルは解析した内容を以下の 5 種類のブロックに分けて記載する．提案ツールでは各種類のブロックの集合の前にコメントでブロックの種類を記し，よりブロックの種類が明確にわかるようにしている．

- Document Information … SPDX ファイル自体に関する情報
- Creation Information … SPDX ファイルの作成に関する情報

- Package ･･･　パッケージに関する情報
- File ･･･　ファイルに関する情報
- Extracted License ･･･　ライセンスリストにない検出されたライセンスに関する情報

Document Information ブロック内の LicenseListVersion や，Extracted Licence ブロックの説明に含まれるライセンスリストとは 2.2 節にて説明したライセンスを簡潔に表記し，管理しているリストである．また，Document Information と Creation Information のブロックは 1 つまでだが，Package，File，Extracted License のブロックは複数記述する場合がある．Package ブロックが複数ある場合，各ファイルがどのパッケージのファイルであるかがわかるようにするため File ブロックの Relationship フィールドでパッケージとの関係を示している．そして SPDXID とは SPDX ファイル内でその SPDX を含むブロックが説明する SPDX ファイル，パッケージ，ファイルまたはライセンスを指すために使われる ID である．

パッケージ間の依存関係の表記　パッケージ間の依存関係の表記方法は同じ SPDX ファイル内のパッケージへの依存関係を表記する場合と，異なる SPDX ファイルのパッケージへの依存関係を表記する場合とで異なる．

同じ SPDX ファイル内のパッケージへの依存関係を表記する場合には依存元のパッケージの Relationship フィールドで依存元のパッケージの SPDXID と依存先のパッケージの SPDXID を用いて「[依存元のパッケージの SPDXID] DEPENDS_ON [依存先のパッケージの SPDXID]」と表す．

異なる SPDX ファイルのパッケージへの依存関係を表記する場合にはまず依存元のパッケージの情報が記載された SPDX ファイルの Document Information ブロックの ExternalDocumentRef で「[依存先のパッケージの情報が記載された SPDX ファイルの SPDXID] [依存先のパッケージの情報が記載された SPDX ファイルの名前空間] [依存先のパッケージの情報が記載された SPDX ファイルのハッシュ値]」を記述する．この際，依存先のパッケージの情報が記載された SPDX ファイルの ID は好きに決めることができ，この SPDXID はその外部ファイルの情報を利用する際に使用される．そして依存関係自体は，依存元のパッケージの Relationship フィールドで，ExternalDocumentRef で決めた SPDX ファイル内で参照するドキュメント ID と依存先のパッケージの SPDXID を用いて「[依存元のパッケージの SPDXID] DEPENDS_ON [依存先のパッケージの情報が記載された SPDX ファイルの SPDXID]：[依存先のパッケージの SPDXID]」と表す．

SPDX のフィールド名	内容
Document Information	
SPDX Version	SPDX 形式のバージョン
DataLicense	SPDX ファイルの内容に対するライセンス
DocumentNamespace	SPDX ファイルの名前空間
DocumentName	SPDX ファイルの名前
LicenseListVersion	準拠したライセンスリストのバージョン
SPDXID	SPDX ファイル自体を指す SPDXID
DocumentComment	SPDX ファイルの内容に対するコメント（免責事項）
ExternalDocumentRef	外部の SPDX ファイルへの参照
Relationship	SPDX ファイルとパッケージの関係
Creation Information	
Creator	作成者（人名・組織名・ツール名）
Created	作成時刻
Package	
PackageName	パッケージの名前
PackageVersion	パッケージのバージョン
SPDXID	パッケージを指す SPDXID
PackageHomePage	パッケージに関する Web ページのアドレス
PackageDownloadLocation	パッケージのダウンロード場所（git など）
PackageVerificationCode	パッケージの固有識別子
PackageLicenseDeclared	パッケージ内で宣言されたライセンス
PackageLicenseConcluded	パッケージに結論づけたライセンス
PackageLicenseInfoFromFiles	パッケージ内のファイルが含むライセンス
PackageCopyrightText	パッケージが含むコピーライト
PackageComment	パッケージのコメント
FileAnalyzed	ファイル解析の有無（デフォルトは True）
Relationship	パッケージ間の依存関係

表 2: 生成する SPDX ファイルのフィールド 1

SPDX のフィールド名	内容
File	
FileName	ファイルの名前・パス
SPDXID	ファイルを指す SPDXID
FileChecksum	ファイルのハッシュ値
LicenseConcluded	ファイルに結論づけたライセンス
LicenseInfoInFile	ファイルが含むライセンス
FileCopyrightText	ファイルが含むコピーライト
Relationship	ファイルが属するパッケージとの関係
Extracted License	
LicenseID	ライセンスを指す SPDXID
LicenseName	ライセンスにつけた名前
LicenseComment	ライセンスに対するコメント
ExtractedText	ライセンスのテキスト

表 3: 生成する SPDX ファイルのフィールド 2

3.3.2 処理手順・実装方法

処理の概要を 4 つのステップに分けて説明する．その後，循環依存の処理について詳細を説明する．

処理の概要を図 5 に示す．まずステップ 1 として，Debian パッケージのパッケージ名から dpkg -s コマンドを用いて control ファイルに記載されている情報を取得し SPDX ファイル生成に当たって必要な情報を抽出する．

次にステップ 2 としてパッケージによってインストールされたファイルのリストを取得した後，copyright ファイルを解析してライセンスとコピーライトの情報を抽出し，copytight ファイルを含むすべてのファイルからハッシュ値を計算する．この copyright ファイルの解析には ScanCode toolkit というツールを使っている．

そしてステップ 3 としてステップ 1 とステップ 2 から得られた情報をマージして，依存関係に関する情報以外の情報を SPDX の形式に変換する．

さらにステップ 4 として依存関係の処理を行う．この際に依存関係情報があった場合は依存先にあたるパッケージが実際に存在するか，存在する場合はバージョンの制約を満たしているか，依存関係を満たすパッケージがない場合は依存関係に指定されているパッケージを Replace するパッケージがないか，また仮想パッケージである場合には依存関係を提供するパッケージがないかを確認する．ここで依存関係を満たしているパッケージが見つかった場

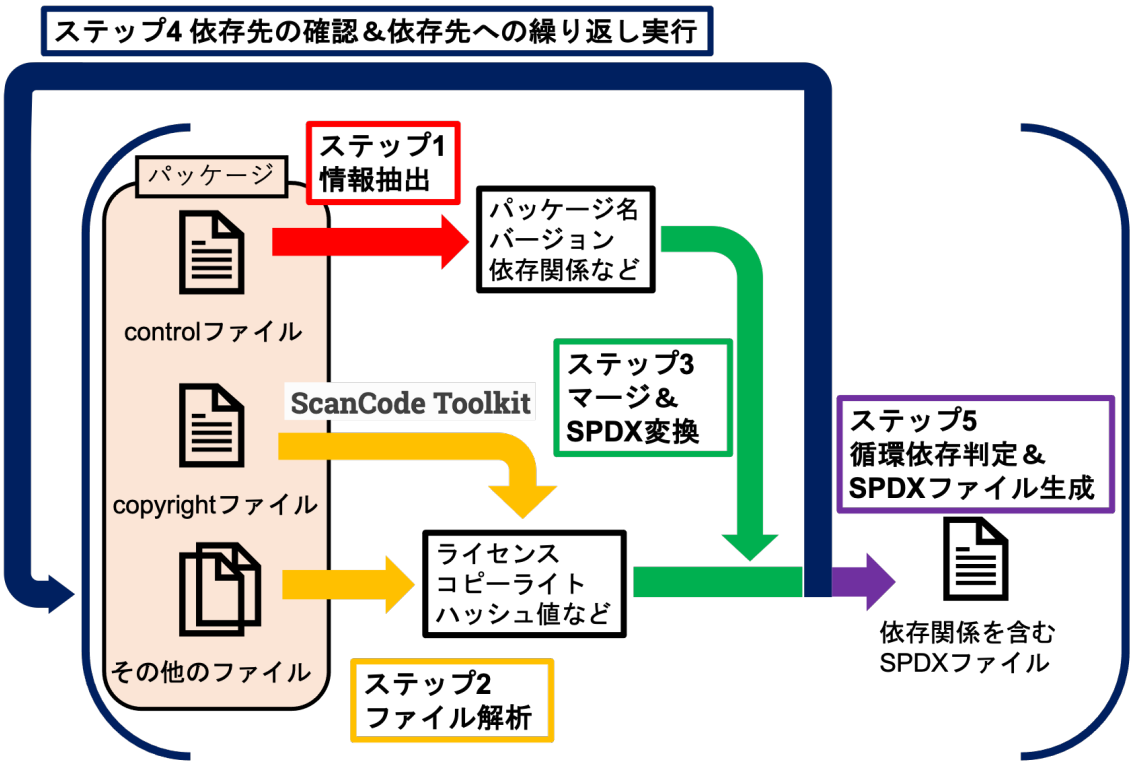

図 5: 処理の概要

合には現在解析中のパッケージに対する SPDX ファイルは生成せず，依存先のパッケージに対してステップ 1 からステップ 5 までを繰り返し実行する．

最後にステップ 5 として依存先のパッケージの解析がすべて終わると現在解析中のパッケージが循環依存の中にあるかどうかを検証し，循環依存の中にない場合は SPDX ファイルを生成し，循環依存の中にある場合は循環依存に含まれるすべてのパッケージの解析が終わってから循環依存に含まれるパッケージの情報をすべてまとめて 1 つの SPDX ファイルとして出力する．

依存先のパッケージが存在する場合に依存元の SPDX ファイルを生成するよりも依存先のパッケージの SPDX ファイルの解析・生成を優先させる理由は SPDX における依存関係の表し方にある．3.3.1 節にて説明したように異なる SPDX ファイルのパッケージへの依存関係を表記する場合にはその SPDX ファイルのハッシュ値を ExternalDocumentRef に載せる必要がある．したがって，依存元の SPDX ファイルを生成する前にその SPDX ファイルが生成済みでなくてはならないのである．

循環依存の処理　まず循環依存について説明する．循環依存とはパッケージの依存先を辿っていくと依存元のパッケージが現れるような依存関係のことである．循環依存の例として libc6 と libgcc-s1 の 2 つのパッケージの依存関係を表示した結果を以下の 2 つの枠内で示す．

libc6 の依存関係表示の結果

```
$ apt depends libc6
libc6
  Depends: libgcc-s1
  Depends: libcrypt1 (>= 1:4.4.10-10ubuntu4)
  Breaks: busybox (<< 1.30.1-6)
  Breaks: fakeroot (<< 1.25.3-1.1ubuntu2~)
  Breaks: <hurd> (<< 1:0.9.git20170910-1)
  Breaks: ioquake3 (<< 1.36+u20200211.f2c61c1~dfsg-2~)
  Breaks: iraf-fitsutil (<< 2018.07.06-4)
～中略～
  Breaks: nscd (<< 2.35)
  Breaks: openarena (<< 0.8.8+dfsg-4~)
  Breaks: openssh-server (<< 1:8.2p1-4)
  Breaks: r-cran-later (<< 0.7.5+dfsg-2)
  Breaks: wcc (<< 0.0.2+dfsg-3)
  Recommends: libidn2-0 (>= 2.0.5~)
  Recommends: libnss-nis
  Recommends: libnss-nisplus
  Suggests: glibc-doc
 |Suggests: debconf
  Suggests: <debconf-2.0>
    cdebconf
    debconf
  Suggests: locales
  Replaces: <libc6-amd64>
```

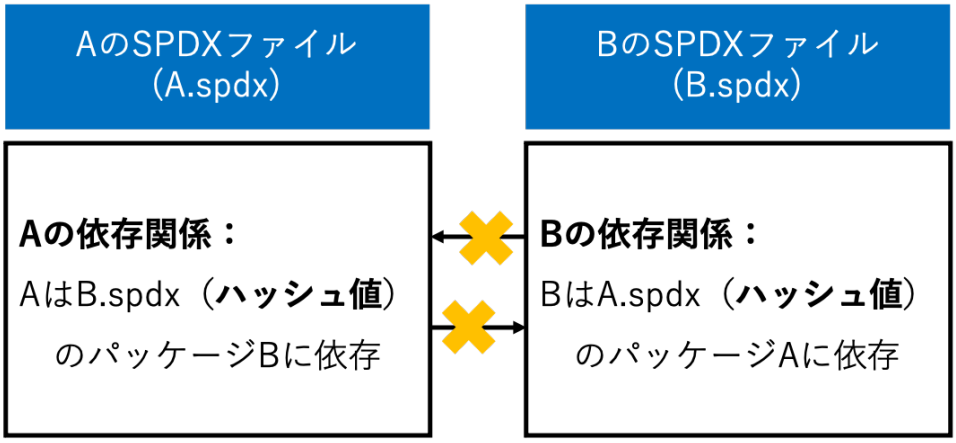

図 6: 循環依存とデッドロック

libc6 の依存関係表示の結果

```
$ apt depends libgcc-s1
libgcc-s1
  Depends: gcc-12-base (= 12.1.0-2ubuntu1~22.04)
  Depends: libc6 (>= 2.35)
  Replaces: <libgcc1> (<< 1:10)
```

これらは循環依存の中でも最も単純な形である．libc6 の依存先 (Depends) に libgcc-s1 があり，その libgcc-s1 の依存先 (Depends) に libc6 があるというのがわかる．

解析中のパッケージが循環依存に含まれているかどうかを判定し，循環依存のパッケージの情報をすべて 1 つの SPDX ファイルにまとめなければならない理由について説明する．依存元のパッケージと依存先のパッケージの SPDX ファイルを別にする場合，依存元のパッケージの情報を記載した SPDX ファイルの ExternalDocumentRef に依存先のパッケージの情報を記載した SPDX ファイルのハッシュ値を載せる必要がある．しかし，依存先のパッケージの依存先に依存元のパッケージがある場合にはそのようなことはできなくなる．なぜならハッシュ値を計算する場合には依存先の SPDX ファイルが完成している必要があるが，その依存先の SPDX ファイルを完成させるためには依存元の SPDX ファイルのハッシュ値が必要となり，デッドロックが発生するからである．この様子を図 6 で示す．パッケージ A とパッケージ B が互いに依存しあう循環依存の関係にあるとき，SPDX ファイルを分割しようとするとデッドロックが発生することがわかる．

また，同じ SPDX ファイル内のパッケージに対する依存関係であってもそのパッケージの

SPDXID が必要になるが，こちらは SPDXID の命名規則を統一し，依存先のパッケージの解析が終わらなくても依存先の SPDXID が計算できるようにすることでデッドロックの発生を防いでいる．

ここから循環依存の処理方法について説明する．パッケージを依存関係を深さ優先探索で辿りながら 1 つずつ解析していくことを考える．すると，循環依存のパターンは今解析しているパッケージをパッケージ A，依存先をパッケージ B とすると 5 つのパターンに分けられる．また循環依存のような依存関係の構造では本来，上の階層も下の階層もないが，ここでは今解析しているパッケージより先に解析した方を上の階層，後から解析した方を下の階層と呼ぶ．

まず 1 つ目のパターンはパッケージ B がパッケージ A の上の階層ですでに解析済みの場合である．これが判別できるようにするためには，辿ってきたパッケージの情報を保持しておき，その情報と照らし合わせる必要がある．この場合，循環依存先のパッケージ名としてパッケージ B の名前を上の階層に伝える．

次に 2 つ目のパターンはパッケージ B 以下の階層のパッケージがパッケージ A に依存している場合である．これは下の階層から伝えられた循環依存先のパッケージ名がパッケージ A 自身であることで判断できる．この場合，上の階層に伝える循環依存先のパッケージ名の情報からはパッケージ A の名前を除く．

3 つ目のパターンはパッケージ A の依存先は問題なくとも下の階層のパッケージが上の階層のパッケージに依存している場合である．下の階層から伝えられた上の階層に伝える循環依存先のパッケージ名があることで判断できる．この場合，上の階層に伝える循環依存先のパッケージ名の情報はそのままで上の階層に渡す．

4 つ目のパターンはパッケージ B がパッケージ A の別の依存先かつ循環依存している依存関係の一部で，既に解析済みの場合である．これはパッケージ A の依存先の中で SPDX ファイルをまだ生成していないパッケージとその推移的な依存先のリストを計算しておき，そのリストに含まれるかどうかで判断する．この場合はすでに解析済みのためパッケージ B の解析には移らず，パッケージ A の依存関係にパッケージ B を追加する．

そして最も複雑な処理を要する 5 つ目のパターンはパッケージ B がパッケージ A の上の階層のパッケージのパッケージ A が含まれない別の依存先の中で解析済みであり，なおかつその依存先が循環依存の中にある場合である．これは解析済みのパッケージのリストを保持しておくことで判断できる．そして上の階層に伝える循環依存先のパッケージ名の情報にはパッケージ B の名前を加え，パッケージ B を含む別の依存先を持つ上の階層のパッケージに処理が戻った際に依存先の中で SPDX ファイルをまだ生成していないパッケージとその推移的な依存先のリストに含まれるパッケージ名を上の階層に伝える循環依存先のパッケージ名から削除することで処理する．

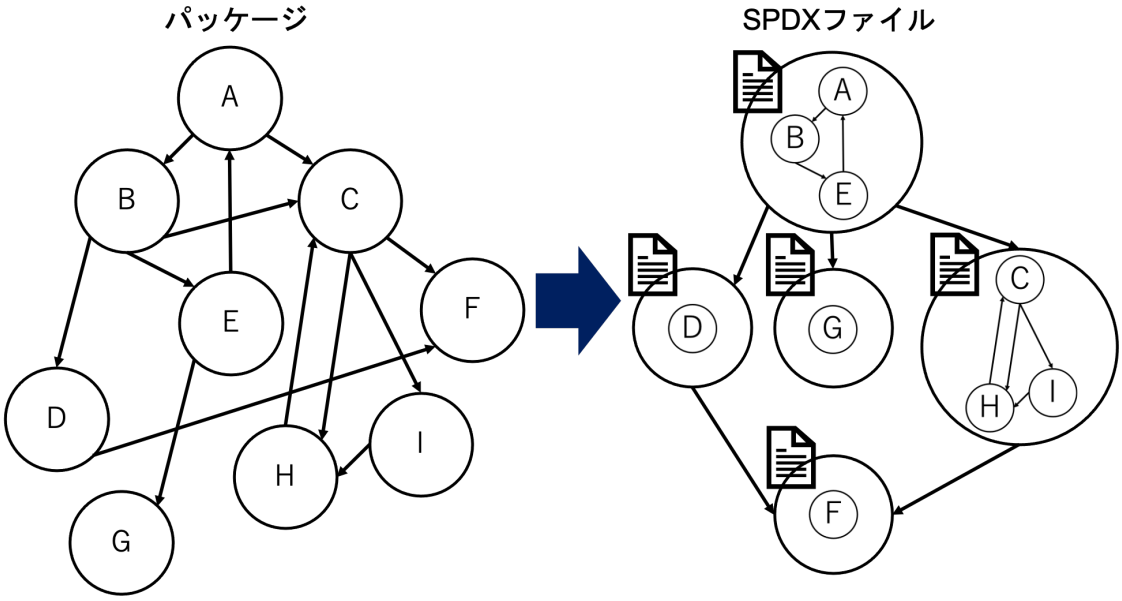

図 7: パッケージ間の依存関係と SPDX ファイル間の依存関係

また循環依存を処理して 1 つのパッケージにまとめたことで結果的に依存関係が整理されるという側面もある．パッケージ間の依存関係と生成した SPDX ファイル間の依存関係の例を図 7 に示す．複雑であった循環依存の部分が 1 つにまとめられることで依存関係の木構造がシンプルになることがわかる．この例では 3 つのパッケージ間の循環依存が 2 つ含まれる依存関係を示しているが実際には循環依存に含まれるパッケージの数はさらに多く，複雑になることもあり，SPDX ファイルを作成することにによる依存関係の簡略化の効果は大きくなる．実際にどの程度の循環依存が含まれているかや，その循環依存の大きさについては 4.1.1 節にて示す．

3.4 パッケージ情報の出力機能

生成された SPDX ファイル群は基本的にパッケージごとに SPDX ファイルが分かれているため，依存関係を辿る際には ExternalDocumentRef に書かれている SPDX ファイルを辿るだけでよい．また，循環依存するパッケージ群は 1 つの SPDX ファイルにまとめられているため，依存関係を辿った先で同じパッケージの SPDX ファイルに再び遭遇することも少なくなる．そのため，インストール済みのすべてのパッケージの情報が 1 つの SPDX ファイルにまとめられている場合に比べれば依存関係は辿りやすい．また，SPDX ファイルに載っている情報はすべてインストール済みのパッケージの情報なのでコマンドを何度も叩いたり，control ファイルに何度もアクセスしたりする場合と比べても，その都度インストール済みか

どうかを確認しなくて良い分，楽に依存関係を辿ることができる．しかしながら，厳密に脆弱性の影響を考慮したい場合，すなわち「脆弱性を持つパッケージの依存関係を逆に辿る」もしくは「インストール済みのすべてのパッケージの依存関係を推移的に辿り，その中に脆弱性を持つパッケージがないかを確かめる」のどちらかをしたい場合は手動では多少手間がかかる．SPDX ファイルの管理・解析をするツールがあればその手間を省くことができるが，ない場合を考慮して提案ツールでは生成した SPDX ファイルから依存関係を含むパッケージの情報を取り出せるようにした．具体的にはパッケージを 1 つ指定すると指定したパッケージの依存関係を逆に辿って現れるパッケージ群とパッケージの依存関係を推移的に辿って現れるパッケージ群を一覧表示する．提案ツールで SPDX ファイルにした時点で依存関係が整理されているため処理の内容は単純である．具体的には指定したパッケージ情報を記載した SPDX ファイルの ExternalDocumentRef に書かれている SPDX ファイルを辿って現れる SPDX ファイルに含まれるすべてのパッケージをリスト表示する．また合わせてパッケージのバージョンも出力する．

OS	Windows10 wsl2 上の Ubuntu22.04
CPU	Intel(R) Xeon(R) CPU E5-1603 v4 @ 2.80GHz 2.80 GHz
追加したパッケージ	python3-pip, pipx
総パッケージ数	517

表 4: テスト環境

4 実証・評価

4.1 動作確認

動作確認には Windows10 の WSL2 に新しくインストールした Ubuntu22.04 を利用した．テスト環境の詳細は表 4 に示す．提案ツールをインストールして実行するために python3-pip と pipx というパッケージを追加でインストールしている．提案ツールはインストール済みのすべてのパッケージを解析するように実行した．

4.1.1 実行結果

提案ツールを実行した結果を表 5 に示す．循環依存の数 13 に比べてパッケージの依存関係の中に循環依存が現れる回数の合計が 562 と非常に多い．パッケージの数が 517 であることからパッケージの依存関係を解析すると平均で 1 回以上は循環依存が現れることになる．したがって，基本的なライブラリにあたるパッケージが循環依存に含まれている可能性が高い．また，循環依存は大きなもので 251 個ものパッケージを含んでおり，非常に複雑になっていることがわかる．したがって，3.3.2 節で述べた循環依存を 1 つの SPDX ファイルにまとめることによってもたらされる依存関係を整理する効果は大きい．また，この循環依存をまとめた SPDX ファイルが 5,680kbyte と最も大きなサイズとなっているが，この実行環境でさえ出力データの合計からすれば半分程度に抑えられており，インストールしたパッケージが多くなるほど出力データの合計が大きくなることを考慮すれば解析結果をすべて 1 つのファイルにまとめるよりはファイルを分けることによって手動によるデータのアクセスのしやすくなっていると言える．依存関係において Replaces や Provides が参照された回数の合計は 16 回と非常に少なかった．これは Debian パッケージの依存関係が仮想パッケージ以外を優先している結果だと考えられる．この実行にかかった時間は 3,849.49 秒，時間に直すと 1 時間程度であった．この時間はファイル解析の時間と copyright ファイルからライセンスとコピーライトを解析するという方法でのライセンス解析の時間を含んでいる．なお，ファイル解析，ライセンス・コピーライト解析を行わずに SPDX ファイルを生成した場合にかかった時間は 6.90 秒であった

出力した SPDX ファイルの数	252
SPDX ファイルの最大サイズ	5680 kbyte
SPDX ファイルの最小サイズ	3 kbyte
SPDX ファイルの平均サイズ	46 kbyte
出力データの合計サイズ	11,626 kbyte
解析したパッケージの数	517
解析したファイルの合計数	30511
循環依存数	13
最も大きな循環依存に含まれるパッケージの数	251
循環依存が依存関係に現れる回数の合計	562
Replaces,Provides を参照した回数の合計	16
実行時間	3,849.49 秒

表 5: テスト結果

4.1.2 検証ツールを利用した確認

インストール済みのすべてのパッケージを解析した結果として生成された SPDX ファイル群を SPDX 公式のオンラインツールの Validator で文法的な間違いがないかを検証した．その結果，ファイルサイズがオンラインツールでは実行できないほど大きくなった SPDX ファイル 1 つを除いてすべて文法的な間違いがないことが確認できた．また，1 つ除いた SPDX ファイルについてもファイル情報を一部削除してサイズを削減するとオンラインツールで検証することができ，文法的な問題がないと確認できた．

4.2 簡易ツールとの比較

提案ツールはインストール済みの依存関係かつバージョンの制約を満たしている依存関係を正確に取得する．そのため，生成された SPDX ファイルに記載されている依存関係は必要最低限のものになっている．実際に apt depends コマンドを繰り返して依存関係を取得する簡易なツールを作成し，提案ツールが生成する SPDX ファイルから取得できる依存関係とどの程度違いがあるのかを調べた．既存ツールに Debian パッケージを対象に依存関係を解析できる SPDX ファイル生成ツールがあればそのツールが生成する SPDX ファイルから取得できる依存関係と比較するべきだが，本研究では見つけられなかったため簡易ツールを作成し

過大評価する依存関係	4,992,049
過小評価する依存関係	0

表 6: 提案ツールと比べて簡易ツールが過大・過少評価する依存関係

て比較している．

4.2.1 比較結果

簡易ツールと提案ツールのそれぞれから得られる各パッケージの推移的な依存関係を比較し，過大評価するパッケージ，すなわち実際にはシステムに存在しない依存関係であるにも関わらず過剰に依存関係として検出してしまうパッケージの数がどの程度あるのかと，過小評価するパッケージ，すなわち実際には依存関係があるにも関わらず依存関係がないと判断してしまうようなパッケージがどの程度あるのかを調べた．その結果を表 6 にて示す．

簡易ツールは実際にシステムに存在するかどうかとバージョン制約の確認をしていないため，非常に多くのパッケージを報告してしまう．これではもしこの依存関係を元に脆弱性の影響を調べる際にシステムに存在しないパッケージの報告を大量にしてしまう．したがって本当に対処が必要な脆弱性への対応が遅れてしまう可能性がある．また，簡易ツールが繰り返し実行している apt depends はパッケージをインストールできる最新バージョンのパッケージの依存関係を示しているためインストールされているバージョンの違いによっては依存関係に違いが生じることもあるかと考え，過小評価した依存関係についても調べたが見つかることはなかった．

4.3 手動による依存関係解析との比較

提案ツールを用いて依存関係を解析する場合，実行するコマンドは SPDX ファイルを生成するコマンドとパッケージ情報を出力するコマンドだけで良い．もし，SPDX ファイルの解析ができるツールがあれば SPDX ファイルを生成するコマンドだけで良い．これが Debian パッケージの推移的な依存関係の解析において提案ツールがない場合と比べてどの程度の作業工数の短縮につながっているのかをコマンドの実行数の違いで比較する．提案ツールがない場合は apt depends コマンドと dpkg -s コマンドのみを使うことを想定しており，apt depends コマンド 1 回で，1 つのパッケージの依存関係すべてを確認し, dpkg -s コマンド 1 回で 1 つのパッケージがシステムに存在するかどうかとバージョンの制約を満たしているかどうかを確認するものとする．具体的には以下のような調査を行う．まず，インストール済みの 517 のすべてのパッケージに対してそれぞれ自身以外のパッケージとの組を作るそしてそのすべての組のそれぞれのパッケージに対して一方のパッケージが他方のパッケージを推移

的な依存先として持つかどうかを判断するのに何回のコマンドの実行を要するかを数える．コマンドの実行数は提案ツールを改造することで計測した．

4.3.1 調査結果

調べた結果，すべての組み合わせの判断に要したコマンドの実行回数の平均は 194.05 回であった．この結果からも推移的な依存関係に含まれているかどうかを手動で検証するには非常に多くの手間を要することがわかる．またコマンドの実行回数がここまで大きな数字になっている原因として推移的な依存先として含んでいない時の判断をする場合には最後まで調べる必要があるからだと考えられる．対して，提案ツールを用いる場合は初めの 1 回こそ SPDX ファイルの生成が必要なものの，以降の推移的な依存関係の調査はすべてコマンド 1 回で実行することができる．また，実行時間も提案ツールは SPDX ファイルの生成を 1 時間程度で行うことができ，推移的な依存関係の出力には 1 秒も要しない．

5　まとめと今後の課題

依存関係を推移的に解析する SPDX ファイル自動生成ツールを開発した．提案ツールは Debian パッケージを対象としており，主に control ファイルや copyright ファイルの情報を解析することで SPDX ファイルの生成に必要な情報を入手している．

推移的な依存関係を解析するということはライセンスや脆弱性といったリスクに対処する上で重要であるが，Debian パッケージでは実際にインストール済みのパッケージ間における推移的な依存関係の取得において手間がかかる．そこで提案ツールを用いることでその手間を低減することができることを確認した．また，提案ツールから得られる依存関係はパッケージの存在とバージョンの制約を確認した後のものであるため，余分に依存関係となるパッケージを報告することがない．これは脆弱性を検証する際に有効に働く．

今後の課題としては Debian パッケージ以外のパッケージも対象にとることができるように取り組んでいきたい．

謝辞

本研究を行うにあたり，御多忙の中懇切なる御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座 肥後 芳樹 教授に，心から感謝の意を評します.

本研究の全過程を通して，研究の起点の設定，論文の執筆など，手厚く御指導，御助言を賜りました南山大学大学院理工学研究科　ソフトウェア工学専攻 井上 克郎 教授に心から感謝の意を表します.

本研究の全過程を通して，研究の方向性の検討，問題点の指摘，論文の執筆など御指導，御助言を賜りました福知山公立大学情報学部 眞鍋 雄貴 講師に心から感謝の意を表します.

本研究の全過程を通して，論文の執筆や発表内容の指摘，添削など御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座 松下 誠 准教授，ならびに神田 哲也 助教に心から感謝の意を表します.

最後に，日々御助言や御協力頂き，激励を賜りました肥後研究室の皆様ならびに事務職員 軽部 瑞穂氏に心から感謝致します.

参考文献

[1] Chris DiGiamo Andrew Archer, Doug Bienstock and Glenn Edwards. Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor — mandiant. https://www.mandiant.com/resources/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor, 2020. (Accessed on 06/26/2022).

[2] R. Michael Azzi. Cpr: how jacobsen v. katzer resuscitated the open source movement. *University of Illinois Law Review*, p. 1271, 2010.

[3] Ericka Chickowski. Log4j highlights need for better handle on software dependencies. https://www.darkreading.com/application-security/log4j-highlights-need-for-better-handle-on-software-dependencies, 1 2022.

[4] Free Software Foundation Europe. Fsfe compliance workshop discovers gpl violation by fantec, welte wins in court - fsfe. https://fsfe.org/news/2013/news-20130626-01.en.html, 6 2013.

[5] International Organization for Standardization. Iso - iso/iec 5962:2021 - information technology — spdx specification v2.2.1. https://www.iso.org/standard/81870.html, 8 2021.

[6] Linux Foundation. Tools - software package data exchange (spdx). https://spdx.dev/resources/tools/.

[7] Linux Foundation and its Contributors. *Software Package Data Exchange (SPDX™) Specification Version: 1.0*, 2010.

[8] Linux Foundation and its Contributors. *A common software package data exchange format*, 2.0 edition, 2015.

[9] U.S. Department of Homeland Security (DHS). Review of the december 2021 log4j event. https://www.cisa.gov/sites/default/files/publications/CSRB-Report-on-Log4-July-11-2022_508.pdf, 7 2022.

[10] United States Executive Ofice of the President[Joe Biden]. Executive order 14028: Improving the nation's cybersecurity — the white house. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/, 5 2021.

[11] National Telecommunications and Information Administration. The minimum elements for a software bill of materials (sbom). https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf, 7 2021.

[12] James Williams and Anand Dabirsiaghi. The unfortunate reality of insecure libraries. https://cdn2.hubspot.net/hub/203759/file-1100864196-pdf/docs/contrast_-_insecure_libraries_2014.pdf, 3 2012.

[13] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. *ICSE*.

[14] 経済産業省 商務情報政策局サイバーセキュリティ課. 最近の産業サイバーセキュリティに関する動向について. https://www.meti.go.jp/shingikai/mono_info_service/sangyo_cyber/wg_seido/wg_uchu_sangyo/pdf/003_03_00.pdf, 11 2021.

[15] 経済産業省 商務情報政策局サイバーセキュリティ課. サイバー・フィジカル・セキュリティ確保に向けたソフトウェア管理手法等検討タスクフォースの検討の方向性. https://www.meti.go.jp/shingikai/mono_info_service/sangyo_cyber/wg_seido/wg_uchu_sangyo/pdf/006_03_00.pdf, 3 2022.