情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

Javaでのインジェクション脆弱性防止規約の遵守を検証する定数伝播解析法の提案

山　岡　裕　司[†1]

インジェクション脆弱性は報告が多数あるアプリケーションセキュリティ問題であり，それを効率的に防止する手法を確立すべきである．

本稿では，バイトコード・ベリファイヤを応用した，Javaプログラムのインジェクション脆弱性を検出する定数伝播解析法を提案する．この解析法は高速で偽陰性がなく，対象プログラムが脆弱性防止規約を遵守していることの検証に有用である．また，脆弱性発見法としても有用なことを示す評価結果を得た．

Constant Propagation Analysis as Coding Convention Checker against Injection Vulnerabilities for Java.

Yuji Yamaoka[†1]

Injection vulnerabilities are application security problems that are reported frequently, and we should establish the method for reducing them efficiently.

In this paper, we propose a constant propagation analysis that detects injection vulnerabilities based on bytecode verifier technique for Java program. Our analysis is fast and has no false negative, and therefore it is useful as coding convention checker. Moreover, we obtained an encouraging evaluation result that indicates that our analysis is also useful as injection vulnerabilities detector.

†1 株式会社富士通研究所
FUJITSU LABORATORIES LTD.

1. はじめに

近年，最も多く報告されているアプリケーションセキュリティ問題に，インジェクション脆弱性がある．インジェクション脆弱性とは，プログラムへの入力を，十分なセキュリティ的検査／変換をせずに出力することによる脆弱性である．代表的なものに，SQLインジェクションやXSS（クロスサイト・スクリプティング）と呼ばれるものがある．

独立行政法人情報処理推進機構（IPA）によると，2008年第4四半期に届出のあったウェブサイトの脆弱性全体のうち，SQLインジェクションは10%，XSSは46%を占める[1)]．また，2005年の調査によると，SQLインジェクションによる被害は，直接被害額だけで1サイトあたり約4,800万円〜1億円程度に上ると算出されている[2)]．

これらの背景を鑑みるに，インジェクション脆弱性を効率的に防止する手法の確立は急務である．

以降ではインジェクション脆弱性の代表例として，SQLインジェクションを説明に用いる．SQLインジェクションは，プログラムへの入力によってSQL文の構文木が想定外のものになってしまい，危険なSQL文を発行してしまう脆弱性である．

たとえば，**図1**はSQLインジェクション脆弱性の疑いがあるJava[⋆1]コードである．(a)行でSQL文を組み立てているが，引数`column`の値によって（たとえば，空白文字を含めることによって）組み立てられるSQL文の構文木が様々なものに変化しうる．

インジェクション脆弱性を防止するのは，概念的には難しくない．入力が含まれている

```
PreparedStatement prepareStatement(Connection connection, String column, String value)
throws SQLException {
    String sql = "SELECT id FROM table WHERE " + column + "=?"; // (a)
    PreparedStatement statement = connection.prepareStatement(sql); // (b)
    statement.setString(1, value);
    return statement;
}
```

図1 SQLインジェクション脆弱性の疑いがあるコード
Fig. 1 Suspicious code about the SQL injection vulnerability.

⋆1 Javaは米国Sun Microsystems, Inc.の米国およびその他の国における商標または登録商標である．

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

変数などを出力する前に，十分なセキュリティ的検査／変換をすれば大抵のインジェクション脆弱性は防止できる．たとえば，図 1 では (a) 行の前に `column` について検査をするのが良い．

なお，図 1 では引数 `value` については，バインド機構と呼ばれるもの（PreparedStatement）を利用することで，SQL インジェクション脆弱性を防止している．バインド機構を利用できる場合に積極的に利用することも良い防止方法の 1 つである．

しかし，現実的には，つい検査を忘れてしまうとか，そもそも検査をすべきことを知らない／気づかないといったことがあり，多くの脆弱性が生じてしまうと考えられる．また，検査の仕方に一貫性がないと，どこでどのような検査をしているかがわかりづらくなるため，プログラム改修時にも脆弱性が生じやすくなる．

インジェクション脆弱性の防止を支援する技術の 1 つに，プログラムの静的解析がある．静的解析は，プログラムを実際に動かすことなく解析する作業である．

簡潔で身近な静的解析として，grep などのパターンマッチング技術が挙げられる．対象手続き（出力用の手続きのことで，手続きとはメソッド，コンストラクター，静的初期化子の総称である）を呼び出している箇所を正規表現などで検索し，検査をしていないコードを探す足がかりにするという利用法が考えられる．たとえば，SQL インジェクションに関係する対象手続きとして `java.sql.Connection#prepareStatement(String)` というメソッドがある．このメソッド呼び出し箇所をプログラム全体から検索すると，図 1 の (b) 行も検出される．図 1 ではその後，人手により引数の `sql`（あるいはその一部の `column` など）について検査されているか調べる．

パターンマッチング技術は偽陰性（検出漏れ）がない[⋆1]が，偽陽性（過剰検出）が大変多い．たとえば，図 1 の (a) 行で，仮に `sql` に文字列リテラルが代入されていたとすると，明らかに脆弱性ではないが，パターンマッチング技術はそれでも (b) 行を検出する．パターンを少しも工夫しないと偽陽性率は 100%であり，工夫して偽陽性率を下げるのも難しい．

本稿では，偽陰性がない，Java プログラムのインジェクション脆弱性を検出する定数伝播解析法を提案する．本解析法は，ベリファイヤ（Java バイトコード・ベリファイヤ）を応用したもので，高速である．さらに，無害化メソッドなど，特別に定数扱いすべき式などの情報を追加しやすいアルゴリズムとなっている．

⋆1 本稿では，リフレクションによる対象手続きの直接呼び出しや，動的なクラス定義などによる脆弱性は対象外とする．

本解析法は，脆弱性防止規約を遵守していることの検証に有用だと考える．脆弱性防止規約とは，その規約を遵守してコーディングすればインジェクション脆弱性が生じないようなコーディング規約のことである．本解析法は偽陰性がないため，規約を遵守していない箇所を漏れなく検出することができる．また，この規約は次の性質を持つよう定めることができる．

- それを遵守していれば本解析法に検出されないで済む．
- 遵守するためのコーディング方法にある程度自由度がある．

つまり，上手く規約を定めることで，偽陰性も偽陽性もない検査ができるため，効率的に脆弱性を防止できるようになる．規約を定めることは，検査の仕方に一貫性を持たせる効果も期待できる．そのため，プログラムの新規開発時にも改修時にも，漏れなく効率的にインジェクション脆弱性を防止する効果が期待できる．

以降本稿では，次の順に説明する．2 節では，ベリファイヤを中心に Java の静的解析について述べる．3 節では，本解析法のアルゴリズムを述べる．4 節では，本解析法の実装例と，脆弱性防止規約例について述べる．5 節では，実装例での評価結果について述べる．そして，6 節で関連研究について述べ，7 節で結びとする．

2. Java の静的解析

Java は Java VM と呼ばれる仮想マシン用の言語で，近年では特に有名になった．

Java は正確には，Java プログラミング言語と，クラスファイルと呼ばれる 2 つの形式がある．Java VM が解釈・処理するのはクラスファイルである．Java プログラミング言語はコンパイラによってクラスファイルに変換できる．また，クラスファイルも Java プログラミング言語に変換できる場合が多い．

本稿で提案する解析法はクラスファイルを対象にしており，Java とは原則的にクラスファイルのことを指すものとする．ただし，例などでは簡単のため Java プログラミング言語での記述を中心にする．

Java VM は Java の手続きを処理するとき，その手続き用の Java フレームを使う．Java フレームは手続き用の記憶領域で，局所変数についての記憶領域と，オペランド・スタックについての記憶領域からなる．

オペランド・スタックは，計算の途中結果などを保持するのに使われるスタックである．たとえば，変数への代入を表現するコードは，クラスファイルでは，代入値をスタック上にプッシュする命令と，スタック・トップをポップして代入先に格納する命令とで表現される．

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

Java VM には，信頼できないクラスファイルを安全に実行するための機構がある．そのうちの 1 つに，ベリファイヤがある．ベリファイヤはクラスファイルを実行する前に，クラスファイルの妥当性を静的解析する．

ベリファイヤの主な目的の 1 つは，型の整合性の検証である．たとえば，引数が文字列型のシグネチャを持つ手続きに対し，整数型のオブジェクトを引数にして呼び出す可能性がある場合，ベリファイヤはエラーとし，そのクラスファイルは実行されない．

図 2 はベリファイヤのアルゴリズムの主要部分である．ベリファイヤは Java VM をシミュレートするように，制御の流れに沿いながら Java フレームの確認と更新をしていく．Java VM は 1 つの手続きを処理している間は 1 つの Java フレームを使い回すが，ベリファイヤは命令毎に Java フレームを記憶しておく．また，Java VM は Java フレームの要素として実行時の実際の値を記憶するが，ベリファイヤは値ではなく型を記憶する．

ベリファイヤは，クラスファイル中にループや例外による制御ジャンプがあっても，あらゆるデータフローで問題がないことを検証する．たとえば，図 2 から読み取れるように，ループが検出された場合，融合によって Java フレームが変化したときにのみ「変更」ビットをセットすることで，同じ命令を適切な回数だけ解析する．そのため，検証漏れがない．

本稿で提案する解析法は，この検証漏れのないベリファイヤの技術を応用することで，偽陰性のない解析を実現した．

3. 解　析　法

本解析法は，先述のベリファイヤを応用したものである．

本解析法は，Java フレームの要素に非定数情報と別名情報を持たせ，それらが各命令によってどう更新されていくかをシミュレーションすることで，対象手続き引数（対象手続きの脆弱性に関係する引数）に非定数が渡される可能性を解析する方法である．**表 1** はベリファイヤとの主な違いを表にしたものである．

非定数情報は，対象手続き引数に渡されるときに非定数である可能性を判断するのに使う情報である．非定数である可能性があるか否かを示す論理値で表現するものとし，非定数で

表 1　Java バイトコード・ベリファイヤとの主な違い
Table 1　Main differences from the Java bytecode verifier.

	Java フレームの要素	解析内容
本解析法	非定数情報と別名情報	対象手続き引数に非定数が渡されている可能性
ベリファイヤ	型情報	型不整合の可能性

各命令には「変更」（*changed*）ビットがあり，この命令に着目する必要があるかどうかが示される．まず，最初の命令に対してのみ「変更」ビットがセットされる．そして，(略) 以下のループを実行する：

(1)　「変更」ビットがセットされている仮想マシン命令を選択する．「変更」ビットがセットされている命令が残っていない場合，そのメソッドは問題なく検証されたということになる．さもなければ，選択された命令の「変更」ビットをオフにする．

(2)　以下を行うことにより，該当命令のオペランド・スタックやローカル変数配列に対する影響のモデル化を行う．

- 命令がオペランド・スタックの値を使用するものである場合，スタック中に十分な数の値が存在し，スタック・トップにおける値が適切な型になっていることを裏付ける．(略)
- 命令がローカル変数を使用するものである場合，指定したローカル変数に適切な型の値が保持されていることを裏付ける．(略)
- 命令がオペランド・スタックに値をプッシュするものである場合，(略) 指定された型をモデル化したオペランド・スタックの先頭に追加する．
- 命令がローカル変数を更新するものである場合，そのローカル変数に新たな型が保持されたことを記録する．

(3)　現在の命令の次に実行するべき命令を決定する．(略)

(4)　現在の命令終了時における，オペランド・スタックやローカル変数配列の状態を，次に実行するべき命令に融合させる．例外ハンドラへの制御の移行という特殊なケースでは，オペランド・スタックに例外ハンドラ情報によって示された例外型の単独オブジェクトが保持されるよう設定される．

- 次に実行するべき命令の実行が初回である場合，該当命令を実行する直前のオペランド・スタックやローカル変数の状態として，ステップ 2 と 3 で算出したオペランド・スタックやローカル変数の値を記録する．次に実行するべき命令の「変更」ビットをセットする．
- 次に実行するべき命令が以前に実行されていた場合，ステップ 2 と 3 で算出したオペランド・スタックやローカル変数の値とすでにある値を融合する．値に対する何らかの変更があった場合には「変更」ビットをセットする．

(5)　ステップ 1 から続行する．

図 2　Java バイトコード・ベリファイヤのアルゴリズムの主要部分（文献 3) より一部省略して引用）
Fig. 2　The core algorithm of the Java bytecode verifier (Quoted from the reference 3)).

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

ある可能性がある場合を 1，可能性がない場合は 0 とする．つまり，本解析法では，対象手続き引数に渡される要素の非定数情報が 1 であったときにその旨を出力する．

別名情報は，ある要素を非定数化する（非定数情報を 1 に変更する）とき，別名関係のある別の要素も非定数化するのに使う情報である．別名情報は適当な ID の集合で表現するものとし，同じ ID を含む要素同士は別名関係があるとみなす．

以下の説明で，要素は次のように表記する．

$$(b, \{ids\ldots\}) \tag{1}$$

ここで，b は非定数情報，$\{ids\ldots\}$ は別名情報である．$ids\ldots$ は ID の列とする．たとえば，非定数で ID が O の要素は $(1, \{O\})$ と書く．

本解析法では，偽陰性をなくすため，別名関係や非定数についての少しの可能性も見逃さないようにする．そのため，手続き内解析にとどめ，メンバー（フィールド，メソッド，コンストラクター）参照についてはシグネチャより詳細な情報を解析しない．つまり，仮引数，フィールド，メソッドの返値などは原則的に非定数扱いする．

本解析法のシミュレーションのアルゴリズムは，ベリファイヤのアルゴリズムである図 2 の，2「モデル化」と，4「融合および例外オブジェクト」を変更したものである．まず，後者の「融合および例外オブジェクト」から説明する．

3.1 融合および例外オブジェクト

例外ハンドラで `catch` した例外オブジェクトは，型などに関わらず $(1, \{O\})$ とする．ID が常に O なのは，実際は無関係なオブジェクト同士に別名情報があることを示しかねないが，不変オブジェクトではないことが表現できれば十分との理由からである．そして，本解析法は手続き内解析のため，例外オブジェクトは非定数扱いとする．

要素同士の融合は，非定数情報については論理和，別名情報については和集合，の計算結果とする．これは，別名関係や非定数についての可能性が少しでもあればその情報を残すことを意味する．

ところで，融合の方法については注意しなくてはならない点がある．図 2 はループ構造となっているため，停止条件として使われる「変更」ビットがいつか必ず全てオフになるようにすべきである．そのための簡単な方法は，全ての要素の集合を有限な束（lattice）とし，融合をその結び（join）とすることである．ID の集合が有限ならば，上記の融合方法はこれに当てはめることができる．そのため，ID を生成する「モデル化」は，有限個の ID しか生成しない方法にしている．

3.2 モ デ ル 化

「モデル化」は，各命令についての影響をモデル化するが，本稿では主要な次の命令についてのみ詳しく説明する．

オブジェクト生成命令　ニーモニック anewarray，multianewarray，new，newarray に相当する命令．

非 static フィールド取得の命令　ニーモニック getfield に相当する命令．なお，配列要素取得の命令（aaload，baload，caload，daload，faload，iaload，laload，saload）も同じように処理する（配列の index は使用しない）．

非 static フィールド設定の命令　ニーモニック putfield に相当する命令．なお，配列要素設定の命令（aastore，bastore，castore，dastore，fastore，iastore，lastore，sastore）も同じように処理する（配列の index は使用しない）．

手続き呼び出しの命令　ニーモニック invokeinterface，invokespecial，invokestatic，invokevirtual に相当する命令．

ここで，以下での説明を簡明にするため，あらかじめ次の用語を定義しておく．

非定数化　要素の非定数情報を 1 にし，同時にその要素と別名関係にある全ての要素の非定数情報を 1 にすること．

既定不変型　基本型，基本型のラッパークラス型，String 型，Class 型のいずれかを示す総称．これらは，そのインスタンスは不変であると，Java の仕様で定められていると考えてよい．

対象手続き／対象手続き引数　本解析の目的は特定の手続きの特定の引数に非定数が渡される可能性がある箇所を検出することである．このとき，その特定の手続きを対象手続き，対象手続きの特定の引数を対象手続き引数と呼ぶ．対象手続き引数は，利用者などがあらかじめ定義しておく．

無害化メソッド　その返値を定数とみなして問題ないメソッド．対象手続き引数には，それに対応する無害化メソッドを定義できる場合が多い．つまり，ある変数をそのまま対象手続き引数に渡すと脆弱性となるが，その変数を無害化メソッドに通した結果を対象手続き引数に渡す場合には問題ない，という場合が多い．無害化メソッドも，利用者などがあらかじめ定義しておく．

なお，本解析法では，生成後（コンストラクター呼び出し完了後）の既定不変型の要素は，別名情報を空集合にする．これは，既定不変型の要素は別名により非定数化される可能性が実際にないからである．

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

3.2.1 オブジェクト生成命令

オブジェクト生成命令では，定数として要素を生成する．

具体的には，命令に対応する要素数をスタックからポップし，$(0, \{P <$ バイトコード位置 $>\})$ をプッシュする．ここで，$<$ バイトコード位置 $>$ とはこの命令のバイトコード位置（整数値）である．よって，ID $P <$ バイトコード位置 $>$ が無限に生成されることはない．

3.2.2 非 static フィールド取得の命令

非 static フィールド取得の命令では，原則的にフィールドとそれを持つオブジェクトとを同一視する．

具体的には，スタック・トップを *objectref* としてポップし，原則的にそれをそのまま *value* としてプッシュする．ただし，フィールドの型が既定不変型であった場合は，別名情報を空集合にしてプッシュする．

3.2.3 非 static フィールド設定の命令

非 static フィールド設定の命令では，原則的にフィールドとそれを持つオブジェクトとの別名関係を設定する．また，どちらかが非定数の場合，もう一方も非定数化する．

具体的には，まずスタック・トップを *value* としてポップし，さらに次のスタック・トップを *objectref* としてポップする．次に，原則的に次の処理を独立におこなう．

(1) *objectref* と別名関係がある（*objectref* の別名情報のいずれかの元（別名）を含む）Java フレーム中の各要素について，その要素と *value* とを融合した結果を新しい要素として設定する．

(2) *value* と別名関係がある Java フレーム中の各要素について，その要素と *objectref* とを融合した結果を新しい要素として設定する．

そして，これらの処理で得た Java フレーム同士を融合する（同じ場所の要素同士を全て融合する）．ただし，フィールドの型が既定不変型であった場合は，2 の処理は省略する．

3.2.4 手続き呼び出しの命令

手続き呼び出しの命令では，原則的に全ての引数と返値との全ての組み合わせに別名関係を設定し，非定数にする．

具体例として invokespecial について説明する．invokeinterface，invokestatic，invokevirtual についても，ほぼ同様のことをする．invokespecial に対する処理手順は次のようになる．

(1) スタック・トップを *objectref* としてポップし，さらに手続きの引数分だけの要素をスタックから $argN(N = 1, 2, \ldots)$ としてポップする．

(2) この手続きが対象手続きであった場合は，対象手続き引数に対応する $argN$ が非定数か調べ，非定数であった場合はその旨の出力をする．

(3) *objectref* および全ての $argN$ を融合した結果を *joined* として算出する．すなわち，融合を加算として表現すると *joined* は次のように表せる．

$$joined = objectref + \sum_{N} argN \tag{2}$$

そして，*joined* の別名情報のいずれかの元（別名）を含む Java フレーム中の各要素について，その要素の非定数情報を 1 に変更し，さらに別名情報を $\{P <$ バイトコード位置 $>\}$ に変更する．

(4) この手続きが返値を返す場合，原則的に $(1, \{P <$ バイトコード位置 $>\})$ をスタックにプッシュする．ただし，この手続きが無害化メソッドであった場合は $(0, \{P <$ バイトコード位置 $>\})$ をプッシュする．さらに，返値の型が既定不変型の場合は別名関係を空集合に変更したものをプッシュする．

なお，対象手続き引数と無害化メソッドは，利用者などがあらかじめ定義したものを使う．

上記は一般的な手続きを扱うときの処理であるが，解析精度を向上させるため，一部の代表的な既知の手続きについては特殊な処理をする．その手続きの動作に応じた，より正確な処理をするのである．

たとえば，特殊な処理をすべき手続きには，既定不変型のコンストラクターがある．既定不変型のコンストラクターでは引数のうち *objectref*（仮引数 `this` に相当）以外には副作用がなく，その副作用も引数からしか受けない（変更可能な static 変数に依存していない）．また，既定不変型のコンストラクター呼び出し後は，生成されたインスタンスに対応する要素の別名情報を空集合にすべきである．具体的には，いずれかの引数が非定数である場合にのみ *objectref* を非定数化し，*objectref* とその別名の要素は別名情報を空集合にすれば良い．

他にも，特殊な処理をすべき手続きには，たとえば StringBuilder や StringBuffer の各手続きがある．インジェクション脆弱性は文字列と深い関わりがあるが，これらのクラスは文字列の加工に使われる最も基本的なクラスであり，特殊な処理の対象とする効果が高い．この特殊な処理により，対象プログラムで StringBuilder などによって同一手続き内で定数を組み合わせて文字列を作成している場合に，作成される文字列も定数だと正しく解析することができる．

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

4. 実装および脆弱性防止規約

評価のため，本解析法を実装した．

言語は Java を使い，ライブラリとして Apache Software Foundation の BCEL[4] を利用した．ソースコード量は，コメント少なめで約 8,000 行，約 300KBytes となった．

3.2.4 節で述べた特殊な処理として，既定不変型，StringBuilder，StringBuffer の各手続きについて実装した．

その他の特徴は次の通りである．

- getstatic の処理で，対象が final 修飾されていた場合（static final フィールドであった場合），そのフィールドの静的初期化子を先に解析し，そのフィールドに設定される値を解析することで，そのフィールドが非定数か否かを決めた．
- 効率化のため，図 2 のアルゴリズムを開始する前に，その手続き内で対象手続きが呼ばれる可能性があるかを解析し，ない場合にはその手続きの解析を終了させた．

この実装で検証できる脆弱性防止規約として，対象手続き引数と無害化メソッドを別途定義するだけで，たとえば次のような SQL インジェクション脆弱性を防止する規約を定めることができる．

- 対象手続き `java.sql.Connection#prepareStatement(String)` の第 1 引数に「定数」以外が渡されるようなコーディングをしてはならない．
- 「定数」とは，次のいずれかである．
  - 無害化メソッド `myproject.Sanitizer#assertAlnum(String)` の返値．
  - リテラル．
  - 定数が設定される static final フィールド．
  - StringBuilder/StringBuffer のデフォルトコンストラクターで生成したインスタンス．
  - String/StringBuilder/StringBuffer 型の定数に対する `toString()` の返値．
  - StringBuilder/StringBuffer による定数同士の連結結果．
  - 定数同士の演算結果．
  - 定数を代入した局所変数．
- なお，仮引数，フィールド，無害化メソッド以外のメソッドの返値，などは「定数」扱いしない．

無害化メソッド `myproject.Sanitizer#assertAlnum(String)` は，たとえば**図 3** のように実装されたメソッドであると考えられる．このコードの意味は，仮引数 `s` が英数字のみから構成される 1 文字以上の文字列であれば仮引数を返値とし，それ以外はエラーとする，となる．

```
package myproject;

public class Sanitizer {

    public static String assertAlnum(String s) {

        if (s.matches("\\p{Alnum}+")) return s;

        throw new IllegalArgumentException(s);

    }

}
```

図 3 無害化メソッドの実装例
Fig. 3 An implementation example of the sanitizing method.

上記規約を遵守したコードは，たとえば**図 4** のようなものである．このコードでは，(a) 行で `java.sql.Connection#prepareStatement(String)` を呼び出しているが，その引数は「定数」となっている．そのため，脆弱性ではなく，この実装に検出されることもない．もし「定数」とならないように変更した場合は，この実装により規約違反として検出される．

このように，規約といってもコードの書き方にある程度自由度を持たせることができるため，少ない規約数で多くの種類のコードに対応できる．

5. 評　　価

4 節で述べた本解析法の実装の，精度と速度性能についての評価結果について述べる．

精度の評価では，米スタンフォード大学 SUIF グループのベンチマーク Stanford SecuriBench Micro 1.08[5] およびツール LAPSE 2.5.6[6] を使用した．

Stanford SecuriBench Micro は，Java プログラムからインジェクション脆弱性を静的に検出する技術のためのベンチマークで，脆弱性のある様々な小さなコードの集まりである．脆弱性のある対象手続き呼び出し箇所が明示されているのはもちろんのこと，対象手続きを呼び出しているが脆弱性がない箇所も明示されているのが特長である．この特長のため，偽陰性率と偽陽性率の両方を容易に算出できる．

LAPSE は，Java プログラムからインジェクション脆弱性を静的に検出する（またその支援をする）ツールである．

表 2 は本解析法と LAPSE を，後述の微修正を施した Stanford SecuriBench Micro に

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

```
PreparedStatement prepareStatement(Connection connection,
SortedMap<String, String> columnValueMap) throws SQLException {
    StringBuilder builder = new StringBuilder();
    for (String column : columnValueMap.keySet()) {
        String string = (builder.length() == 0 ? "WHERE" : "AND");
        builder.append(" ").append(string).append(" ");
        builder.append(Sanitizer.assertAlnum(column)).append("=?");
    }
    builder.insert(0, "SELECT id FROM table");
    PreparedStatement statement = connection.prepareStatement(builder.toString()); // (a)
    int index = 0;
    for (Entry<String, String> entry : columnValueMap.entrySet()) {
        statement.setString(++index, entry.getValue());
    }
    return statement;
}
```

図 4 規約を遵守したコードの例
Fig. 4 An example of code that keep the convention.

適用した結果である．本解析法の偽陰性率は 0%，偽陽性率は約 77%である．LAPSE の偽陰性率は約 18%，偽陽性率は約 87%である．偽陰性率・偽陽性率共に低いことから，この適用では，本解析法は LAPSE よりも精度が良いといえる．

なお，Stanford SecuriBench Micro には明らかに誤りであると考えられる箇所がいくつかあり，そのままでは評価に支障をきたすため，次の 3 箇所を修正して使用した．

(1) Basic26.java の 31 行目：脆弱性であるとの判定（BAD）を書き込んだ．

(2) Collections3.java の 27 行目：『`l12`』を『`l11`』に変更した．

表 2 Stanford SecuriBench Micro 1.08 に対する検出精度の比較
Table 2 A comparison of detection accuracy for Stanford SecuriBench Micro 1.08.

	偽陰性率	偽陽性率
本解析法	$0/137 = 0\%$	$40/52 \approx 77\%$
LAPSE	$25/137 \approx 18\%$	$45/52 \approx 87\%$

(3) Collections13.java の 34 行目：『`new String(s1)`』を『`"added", new String(s1)`』に変更した．

これらの結果，Stanford SecuriBench Micro には，脆弱性候補（対象手続き呼び出し箇所）が 189 箇所，そのうち実際脆弱性であるのが 137 箇所（脆弱性でないのが 52 箇所），あることになる．

このように，本解析法は脆弱性発見法として有用なことがうかがえるが，偽陽性率は高い．そのため，適切な規約をコーディング時から遵守するように開発することが望ましい．もし，規約遵守の漏れがあっても，本解析法により漏れなくその箇所を検出することができる．

速度性能の評価としては，我々が独自に用意した大きめの現実的な 1 つのプログラムに適用してみるだけにとどめた．なぜなら，本解析法は Java バイトコード・ベリファイヤを少し変更したものであり，バイトコード・ベリファイヤに比べて顕著な性能劣化はないはずだからである．対象プログラム 5,685 クラスファイル（約 28MBytes，ソースコード量 100 万行以上）に対し，Microsoft Windows[⋆1] XP Professional，Intel Pentium[⋆2] 4 CPU 3.4GHz 上の Sun Microsystems[⋆3]JRE 1.6.0_13（最大メモリ 512MBytes）上で，`java.io.File#<init>(String)` の第 1 引数を対象手続き引数として適用したところ，約 4 分で完了した（38 箇所を検出した）．十分実用的な性能があるといえる．

6. 関 連 研 究

本稿の関連研究について述べる．

インジェクション脆弱性軽減を目的とした Java の静的解析の研究に，文献 7) がある．精度の比較評価で使用した LAPSE は，この研究成果の一部である．この研究では，実際の 9 種類の大きなオープンソースアプリケーションを静的解析し，29 箇所でインジェクション脆弱性を発見している．その際，偽陽性が，1 種類のアプリケーションからのみ 12 箇所出ている．しかし，この研究では偽陰性について評価していない．

我々は，脆弱性検出技術では特に偽陰性を重視すべきだと考えている．脆弱性は 1 つでも残っていると，それが悪用されることにより大きな被害をもたらすことがある．セキュリ

⋆1 Microsoft，Windows は米国 Microsoft Corporation の米国およびその他の国における商標または登録商標である．
⋆2 Intel，Pentium は米国 Intel Corporation の米国およびその他の国における商標または登録商標である．
⋆3 Sun Microsystems は米国 Sun Microsystems の米国およびその他の国における商標または登録商標である．

Vol.2009-SE-164 No.13
Vol.2009-EMB-13 No.13
Vol.2009-CSEC-45 No.13
2009/5/29

ティへの投資意欲を低下させないためにも，手間ひまをかければ確実に脆弱性をなくせる技術は必要だろう．

同様の目的に対し動的解析技術で取り組んでいる研究に，文献 8) がある．この研究では，クラスファイルを自動変換することで，文字列データに汚染情報を追加して持たせるようにする方法を提案している．プログラムへの入力と考えられる文字列データは汚染されているとし，そこから派生するデータにも汚染を伝播させ，対象手続き引数に汚染データが渡されそうになったらそれを阻止・警告する．

動的解析は一般的に，解析結果の妥当性を人が判断しやすいという利点があるが，実際に動かさなくてはならず，入力も用意しなくてはならないという欠点がある．特に，網羅的に解析するには，コードが網羅的に実行されるような入力を用意しなければならず，難しい場合がある．また，本解析法はコンパイルできるようになった（クラスファイルにできた）時点で適用できるが，動的解析はそれより後の動かせるようになる時点まで適用できないため，開発工程の手戻りの観点からも静的解析は有利である．

7. おわりに

本稿では，Java バイトコード・ベリファイヤを応用した，Java プログラムのインジェクション脆弱性を検出する，偽陰性のない高速な解析法を提案した．本解析法は，パターンマッチングでは難しかった偽陽性の軽減をある程度達成している．

本解析法は，脆弱性検出技術としての有用性を示す評価結果を得られたが，脆弱性防止規約の遵守を検証する技術としても有用である．

今後の課題には，引数の定数化と，手続き間解析への展望がある．

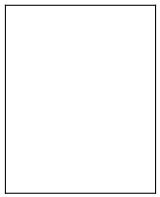
本解析法では，無害化メソッドを簡単に導入できることを示したが，無害化メソッドは返値を定数化するだけで，引数の定数化は実現できていない．引数の定数化が実現できると，コーディングにより自由度を与える規約が定められるようになるだろう．

また，本解析法は原則的に手続き内解析であった．Java プログラムでは，呼び出し元は特定できないことが多いが，呼び出し先は特定できることも多い．特定できる呼び出し先を解析した結果を使うようにすれば，偽陽性を減らせると考える．

参 考 文 献

1) 情報処理推進機構：ソフトウエア等の脆弱性関連情報に関する届出状況 [2008 年第 4 四半期（10 月〜12 月）],
http://www.ipa.go.jp/security/vuln/report/vuln2008q4.html
. 2009 年 4 月 13 日参照.
2) 情報処理推進機構：「企業における情報セキュリティ事象被害額調査」及び「国内におけるコンピュータウイルス被害状況調査」[2005 年],
http://www.ipa.go.jp/security/fy17/reports/virus-survey/index.html
. 2009 年 4 月 13 日参照.
3) ティム・リンドホルム，フランク・イェリン：Java 仮想マシン仕様，ピアソン・エデュケーション，2 edition (2001). 村上雅章 訳.
4) Apache Software Foundation: BCEL,
http://jakarta.apache.org/bcel/
. 2009 年 4 月 13 日参照.
5) Stanford SUIF Compiler Group: Stanford SecuriBench Micro,
http://suif.stanford.edu/~livshits/work/securibench-micro/
. 2009 年 4 月 13 日参照.
6) Stanford SUIF Compiler Group: LAPSE,
http://suif.stanford.edu/~livshits/work/lapse/
. 2009 年 4 月 13 日参照.
7) Livshits, V.B. and Lam, M.S.: Finding security vulnerabilities in java applications with static analysis, *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, USENIX Association, pp.18–18 (2005).
8) Haldar, V., Chandra, D. and Franz, M.: Dynamic Taint Propagation for Java, *In Proceedings of the 21st Annual Computer Security Applications Conference*, pp. 303–311 (2005).

(平成？年？月？日受付)
(平成？年？月？日採録)



山岡　裕司（正会員）
昭和 53 年生．平成 15 年東京大学大学院情報理工学系研究科数理情報学専攻修士課程修了．同年富士通株式会社入社．ソフトウェアセキュリティの研究に従事．