社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－SE－158 (5)
2007／11／2

# 文字列解析を用いたアクセス権推論の精度向上

小金山 美賀† 田渕 直† 立石 孝彰†

Java[TM]2 実行環境には，実行中のコードが特定のシステムリソースに対して，アクセスが許されているかどうかを判断するセキュリティ機構が備わっている．セキュリティ機構を適切に利用するためには，各プログラムポイントのリソースに対して，どのような操作が許されているのかを定義するセキュリティ・ポリシーが必要である．このようなセキュリティ・ポリシーを，与えられたプログラムから静的に推論するプログラム解析アルゴリズムが提案されている．しかし，提案手法では，文字列操作に関するメソッドによって生成される文字列の値を推論することができないため，対象リソースが文字列操作を経て生成された文字列の値によって定まる場合，リソースが特定できず，任意のリソースに対してアクセス可能なポリシーと推論することになる．そこで本稿では，オープンソースライブラリに含まれる，セキュリティ・ポリシーの推論に関連するプログラム変数に対して文字列解析を適用し，必要のないセキュリティ・ポリシーが削減されることを示す．

# Improving Precision of Access Rights Analysis with String Analysis

Mika Koganeyama † ,Naoshi Tabuchi † and Takaaki Tateishi†

The Java[TM]2 runtime system has a security mechanism which guarantees the code under execution has appropriate access permissions to a certain system resource. Use of this security mechanism requires security policies in order to specify what operations are permitted on each such resource at each program point. Previous work proposed a program analysis algorithm to statically infer a semi-optimal policy set from given program text. However, the proposed method cannot calculate the optimal policy when the target resource is determined by string values at run-time, since it does not keep track of all potential string values generated through built-in or user-defined methods. This results in generating excessive security policies, where actually unnecessary resource accesses are permitted. To overcome such limitations, we apply static string analysis to program variables relevant to security-sensitive operations. This paper shows that string analysis can infer string values in open-source libraries, thus unnecessary conservativeness of the existing analysis can be reduced.

## 1. はじめに

### 1.1 Java2 セキュリティ機構とアクセス権解析

Java[TM]2 実行環境には，システムリソースに対する適切なアクセス権限が，実行中のコードに与えられているかを判断するセキュリティ機構が備わっている．この機構を利用するためには，開発者が事前に，各リソースに対してどのような操作が許されるのかを定義する必要がある．

Java ランタイムライブラリにおけるアクセス権限は，抽象クラス java.security.Permission をルートとしたクラス階層でモデル化されている．Permission オブジェクトは，java.security.Permission のサブクラスのインスタンスである．例えば，java.security.Permission のサブクラスである java.io.FilePermission クラスの Permission オブジェクトは，以下のようにインスタンス化される．

```
FilePermission perm =
    new FilePermission("/tmp/abc", "read");
```

上記のように，FilePermission コンストラクタは，第一引数，第二引数にそれぞれ，ファイルリソースのパス名，そのリソースに対する操作を与える．上記の例では，リソース"/tmp/abc"ファイルに対して“読み込み”という操作を許可することを意味する．

Java2 のセキュリティ機構では，このようなアクセス権限をアプリケーションのコードと分離して，セキュリティ・ポリシーとして定義することができる．しかし，セキュリティ・ポリシーを定義するためには，アプリケーションや関連するライブラリのコードがどのようなリソースに対して，どのような操作によってアクセスするのかを把握する必要がある．そのためには，実際にコードを実行し，アクセス権の認証に失敗した部分について検討し，セキュリティ・ポリシーの編集を行い，再度コードを実行する，ということを繰り返す必要がある．また，このようなテスト実行に基づくアプローチは網羅性に欠けるため，必要なアクセス権限を見つけることができず，セ

† 日本 IBM 東京基礎研究所
Tokyo Research Laboratory, IBM Japan

キュリティ・ポリシーの設定が不十分となることがある．

Koved, Pistoia ら[1]は，データフロー解析を用いて，与えられたソースプログラムから，適切なセキュリティ・ポリシーを推論する手法を提案している．しかし，この手法では，FilePermission オブジェクトのように，セキュリティ・ポリシーの設定にファイルパス名といった文字列値に依存する情報が必要である場合，適切なセキュリティ・ポリシーを推論できないことが多い．なぜなら，この手法では，組込みライブラリやユーザ定義のメソッドによって生成される文字列を推論することができないためである．文字列を適切に推論できないと，生成されたセキュリティ・ポリシーには，例えば「任意のファイルの読み込みを許可」といった，明らかにセキュリティ上安全ではない，不必要なアクセス権が多く含まれる可能性がある．

本稿では，上記の課題を解決することを目的として，実行時に生成され得る文字列値を推論するために，アクセス権限の設定に必要なプログラム変数に対して静的な文字列解析を行う．

### 1.2 関連研究

前述したように，Koved, Pistoia ら[1]は，データフロー解析を用いて，静的にセキュリティ・ポリシーを推論する手法を提案している．本稿では，オープンソースライブラリのコードに彼らの手法を適用し，適切なアクセス権限を推論することができなかったプログラムポイントを特定する．

南出[2]は，PHP プログラム中の文字列を文脈自由文法によって近似し，推論する文字列解析器を開発している．南出の文字列解析器は，有限状態のトランスデューサ，すなわち出力付有限オートマトンを用いることで，str_replace 関数のような文字列操作を行う組込み関数の振る舞いを近似している．我々の用いる文字列解析器は，彼のアルゴリズムを Java 言語を対象とするように移植したものである．

また，Christensen, Møller and Schwartzbach[3),4)]も The Java String Analyzer(JSA) と呼ばれる Java の文字列解析器を開発している．

### 1.3 構　　成

以降の本稿の構成は以下のとおりである．2 章にて，従来の手法[1]では，適切なセキュリティ・ポリシーを推論できなかったプログラム例をあげるとともに，我々が用いた文字列解析のアルゴリズムの概略を説明する．3 章にて，文字列解析器の適用対象となる変数や操作を特定する．また，文字列解析器の実装について述べるとともに，いくつかの変数に対して文字列解析を適用し，その結果を示す．

## 2. セキュリティ・ポリシー推論と文字列解析

### 2.1 文字列依存の Permission オブジェクト

FilePermission クラスのインスタンス化に必要なファイルパス名のように，Java2 セキュリティ機構における Permission オブジェクトは，文字列値に依存していることが多い．また，その文字列値は実行中に動的に生成され，ローカル変数やオブジェクトフィールドに代入されることも少なくない．文字列値に依存する Permission オブジェクトについて，以下の例を用いて説明する．

```
1:  public class SampleClass {
2:    void methodA() {
3:      String filename = "methodA_filename";
4:      boolean b = methodB(filename);
5:    }
6:
7:    boolean methodB(String filename) {
8:      String filenameB =
9:           filename.substring(9) + ".txt";
10:     File file = new File(filenameB);
11:     if (file.exists()) {
12:       return true;
13:     } else {
14:       return false;
15:     }
16:   }
17: }
```

上記プログラムの 11 行目で，methodB メソッドにおいて file.exists() を呼び出すことによって，アクセス権認証メソッド java.lang.SecurityManager.checkRead (String file) が呼び出される．checkRead(String file) メソッドは，ファイルパス名を引数として受け取り，コールスタック上のすべてのコードがそのファイルの読み込みを許可されているかを調べる．さらに，checkRead() メソッド内で，java.lang.SecurityManager .checkPermission(Permission perm) メソッドを呼び出す．checkPermission(Permission perm) メソッドの引数として渡される Permission オブジェクトは，ファイルパス名に相当する文字列値と，ファイル読み込みアクションを意味する文字列定数"read"を用いて生成される．よって，file.exists() のアクセス権認証は，file オブジェクトが保持するファイルパス名に依存する．file オブジェクトは，上記プログラムの 10 行目でローカル変数 filenameB の値，すなわち"filename.txt"を用いて初期化される．

我々は，前述した Koved, Pistoia ら[1]の提案した手法を実装した SWORD4J (Software Workbench Development Environment for Java)[5]を上記プログラムに適用した．その結果，SWORD4J ではローカル変数 filenameB の値である"filename.txt"を推論することができず，以下のようなセキュリティ・ポリシーを出力した．

```
java.io.FilePermission "???file???", "read"
```

上記 java.io.FilePermission の後に続く2つの文字列は，ファイル名とそのファイルに対して許可されるアクションを示す．ここでは，ファイル名が"???file???"となっており，特定のファイル名が記述されていない．これは，SWORD4J がプログラム中から実際のファイル名を特定することができなかったことを意味する．そしてこの結果は，セキュリティ・ポリシーとして「アプリケーションは，任意のファイルを読み込み可能である」というきわめて保守的な推論が行われたことを意味する．このようなセキュリティ・ポリシーを適用することは，明らかに最小権限の原則[6]に反する．一方で，1.1 節でも述べたように，SWORD4J のようなツールを利用しなければ，テスト実行に基づいてセキュリティ・ポリシーを生成する必要があり，網羅性に欠ける．

SWORD4J が，変数 filenameB の値を特定することができなかった原因は，変数 filenameB の値は，9 行目の filename.substring(9) メソッド呼び出しと，文字列連結によって生成された文字列であるためと考えられる．したがって，プログラム中の文字列変数の値や，その変数に対する操作を推論することができれば，「任意のファイルに対して読み込み権限がある」といったセキュリティ上安全でない，不必要なセキュリティ・ポリシーを削減することができる．

### 2.2 文字列解析

本稿において，文字列解析とは，与えられたプログラム中の変数やプログラムから出力される値を近似する静的プログラム解析の一種をさす．本稿では，南出[2]による手法を Java に移植したアルゴリズムを用いる．南出のアルゴリズムでは，文脈自由言語 (CFL: Context-Free Language) によって文字列の近似が行われる，すなわち，プログラム中の各変数に対して CFL が推論される．推論される言語は，実行時にその変数が取り得るすべての値を含む，文字列の集合である．ここで，2.1 節で用いた SampleClass クラスのプログラムにおける変数 filenameB の値，すなわち filename.substring(9) の返り値と文字列定数".txt"の連結により生成された値を文字列解析によって推論する．

1.2 節で述べたように，南出のアルゴリズムは出力付有限オートマトンを採用し，メソッド呼び出しによる返り値を CFL で近似している．例えば，引数として整数 9 を与えられた String.substring() メソッドは，9 状態のトランスデューサによって図 1 のようにモデル化される．図 1 の状態 0 と 9 はそれぞれ初期状態と終了状態である．$A$ は任意の 1 文字を，$\epsilon$ は，空列を表している．$A/\epsilon$ でラベル付けされた遷移は，入力された 1 文字をトランスデューサが $\epsilon$ に変換することを表す．また，$A/A$ でラベル付けされた遷移は，入力された 1 文字と同じ文字を出力することを表す．

このトランスデューサを文字列"methodA_filename"に

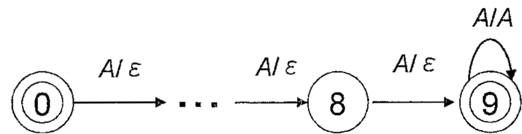

図 1 flename.substring(9) のトランスデューサ

適用すると，出力として単一要素の集合 {"filename"} が得られる*．その結果，変数 filenameB は，以下のような文法によって近似される．

$$S \rightarrow AB$$
$$A \rightarrow \texttt{filename}$$
$$B \rightarrow \texttt{.txt}$$

以下本稿において，文法の非終端記号 $S$ は開始記号を意味するものとする．

## 3. 文字列解析の適用

本稿では，オープンソースとして公開されている Jakarta Commons IO[7] のライブラリを解析対象コードとして使用する．以下，3.1 節にて，ライブラリコード内のセキュリティに関与する操作に関連する変数の例をあげ，解析対象となる変数のパターンを特定する．3.2 節にて，本稿で用いる文字列解析器の実装の概略を述べる．最後に，3.3 節にて，文字列解析を実際に適用した変数をあげ，その適用結果を示す．

### 3.1 解析対象変数の特定

はじめに，Commons IO ライブラリのコードに SWORD4J を適用し，セキュリティに関与する操作のうち，適切なアクセス権限を推論することができなかった箇所を特定した．その結果，146 箇所のセキュリティに関与する操作の中で，SWORD4J が適切にアクセス権限を推論できなかった操作は 117 箇所で，そのほとんどが FilePermission に関連していた．アクセス権限を推論できなかったほとんどの操作が，java.io.File クラスのメソッドであった．File クラスのメソッドに対するアクセス権認証は，多くの場合ファイルパス名に依存しているため，File オブジェクトが保持するファイルパス名を推論することによって，必要なアクセス権限を推論することができる．

例えば，Commons IO ライブラリの org.apache.commons.io.FileUtils クラスには，図 2 に示す openInputStream(File file) メソッドが定義されている．図 2 の 2，3，6，12 行目に記述されている 4 つの操作，file.exists()，file.isDirectory()，file.canRead()，new FileInputStream(file) は，セキュリティに関与する操作である．これらの操作に対する具体的なアクセス権限は，すべて File クラスのオブジェクト file のパス名に依存する．変数 file は，

* トランスデューサは非決定的になりうるため，一般に一つの入力に対して，複数の出力が得られる可能性がある．

```
1:  public static FileInputStream openInputStream(File file) throws IOException {
2:    if (file.exists()) {
3:      if (file.isDirectory()) {
4:        throw new IOException("File '" + file + "' exists but is a directory");
5:      }
6:      if (file.canRead() == false) {
7:        throw new IOException("File '" + file + "' cannot be read");
8:      }
9:    } else {
10:     throw new FileNotFoundException("File '" + file + "' does not exist");
11:   }
12:   return new FileInputStream(file);
13: }
```

図 2 org.apache.commons.io.FileUtils.openInputStream() メソッド

openInputStream() メソッドの引数に渡された値であるため，実際の値は，openInputStream() メソッドを呼び出したメソッドにより生成されている．

一例として，ライブラリの単体テストコードの中に，openInputStream() メソッドを呼び出す以下のようなコードが存在していた．

```
String fileName =
    "LineIterator-" + lineCount + "-test.txt";
File testFile =
     new File(getTestDirectory(), fileName);
...
FileUtils.openInputStream(testFile);
```

testFile オブジェクトは，File コンストラクタによって生成され，その第一引数は親ディレクトリのパス名，第二引数はファイル名である．第二引数のファイル名，すなわち変数 fileName の値は，他の場所で定義された int 型変数 lineCount と 2 つの文字列の連結によって生成される．よって，変数 testFile，すなわち openInputStream メソッドの引数である変数 file のファイルパス名は，File コンストラクタに与えられた 2 つの引数と，システム固有に定義されたパスの区切り文字の連結によって生成される．

File オブジェクトのファイルパス名は，上記の例と同様にコンストラクタの引数によって特定することができ，オブジェクトのライフサイクルにおいて不変である．したがって，コンストラクタの引数に与えられる値を推論することにより，File オブジェクトのファイルパス名を推論することができる．File コンストラクタは，文字列や，他の File オブジェクトや両方を引数として取ることから，我々は，File オブジェクト☆やそのコンストラクタの引数に対して文字列解析を行う．

### 3.2 文字列解析の実装

本稿で利用する文字列解析器は，プログラムの静的解析用オープンソースライブラリである WALA[8] 上で我々が開発したものである．文字列解析器は，南出が提案したアルゴリズム[2] に基づいて，Java を対象言語とするように移植したものであり，java.lang パッケージと java.io.File クラスの主要な文字列操作に関連するメソッドに対応するトランスデューサを実装している．

また，文字列解析を適用する際にいくつかの考慮すべき点があり，我々はそれらの点を実装上で以下のように処理している．

**実行環境依存の値**

文字列値の中には，実行時にしか取得できず，静的には利用できない値もある．java.lang.getProperty(String key) によって取得するシステムプロパティがその代表的な例である．getProperty(String key) メソッドは，プロパティの名前である key を引数として与えられ，外部ファイルまたは別の方法によって，プロパティの名前に対応する値を返す．

このような動的に定められる値については，特定の文字列を解析の事前条件として与えておくこととする．例えば，getProperty(String key) の引数に "java.io.tmpdir"が与えられると，文字列 C:/test/tmpdir"が返されるよう，事前に定義しておく．なお，上記の具体的な文字列について，特別な意味はない．

**File オブジェクトの取り扱い**

3.1 節で述べたように，本稿の目的における解析では，File オブジェクトが保持するファイルパス名を推論する必要がある．そのためには，文字列解析器がオブジェクトのフィールドの値を取得しなければならない．すなわち，オブジェクトの状態を考慮に入れる必要がある．しかし，File オブジェクトのファイルパス名は，そのオブジェクトのライフサイクルの中で不変であるため，ファイルパス名のみを考慮する限りでは，File クラスはステートレスなクラスとみなすことができる．java.lang.String クラスも不変であり，ステートレスであることから，java.io.File クラスは，java.lang.String の特殊な

☆ 本稿では，File クラスを String クラスの特殊なケースとみなして解析する．詳細は 3.2 節を参照のこと．

ケースであるとみなす．よって，本稿の文字列解析器では，java.io.File クラスの以下の 3 つのコンストラクタについて，java.lang.String クラスの変形型とみなして解析する．

```
public File(File parent, String child)
public File(String parent, String child)
public File(String pathname)
```

### 3.3 適用結果

**適用実験 1**

第一の適用実験として，図 2 に示す FileUtils.openInputStream(File file) メソッド内の変数について文字列解析を適用する．はじめに，対象ライブラリコード内の openInputStream() メソッドを呼び出している場所を確認すると，ライブラリの単体テストコードから 13 箇所のメソッド呼び出し部分を特定した．次に，FileUtils クラスと上記単体テストコードとそれらに依存するすべてのクラスを抽出する．そして，すべての単体テストを呼び出すエントリーポイントとして，main() メソッドを解析対象コードに追加することにより，すべてのメソッド呼び出し部分が静的コールグラフ内に現れる．

抽出したコードを SWORD4J と文字列解析に適用した結果，SWORD4J は openInputStraem() メソッド内の file オブジェクトに対する適切なアクセス権の集合を推論することができなかった．一方，文字列解析の結果，file オブジェクトのファイルパス名について，以下のような推論結果が得られた．

$$
\begin{array}{lll}
S & \rightarrow & AB \\
A & \rightarrow & CD \\
B & \rightarrow & EFG \mid \texttt{read.txt} \mid \texttt{read.obj} \mid \texttt{test2.txt} \mid \\
 & & \texttt{LineIterator} - \texttt{closeEarly.txt} \mid \\
 & & \texttt{LineIterator} - \texttt{invalidEncoding.txt} \mid \\
 & & \texttt{dummy} - \texttt{missing} - \texttt{file.txt} \mid \\
 & & \texttt{LineIterator} - \texttt{nextOnly.txt} \mid \\
 & & \texttt{LineIterator} - \texttt{validEncoding.txt} \\
C & \rightarrow & \texttt{C:/test/directory/} \\
D & \rightarrow & \texttt{test/io} \\
E & \rightarrow & \texttt{LineInterator}- \\
F & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \\
G & \rightarrow & -\texttt{test.txt}
\end{array}
$$

推論された文法は，"C:/test/directory/test/io/read.txt"などを含む 12 種類の文字列である．単体テストケースは 13 種類であったが，そのうち 2 つが同じファイルにアクセスしていたため，重複をはぶいた推論結果が出力されている．

**適用実験 2**

第二の実験として，以下に示す LockableFileWriter.createLock() メソッド内の変数に文字列解析を適用する．

```
1: private void createLock()
2:                throws IOException {
3:  synchronized
4:          (LockableFileWriter.class) {
5:   if (!lockFile.createNewFile()) {
6:    throw new IOException(
7:    "Can't write file, lock "
8:    + lockFile.getAbsolutePath()
9:    + " exists");
10:  }
11:  lockFile.deleteOnExit();
12: }
13:}
```

セキュリティに関与する操作は，上記 5 行目の lockFile.createNewFile() メソッドで，変数 lockFile のファイルパス名を推論する．lockFile は，クラスのフィールドで，図 3 に示すコンストラクタ内で初期化される値である．コンストラクタは，File オブジェクト file とロックファイルの存在場所を特定するディレクトリのパス名を引数として取る．ここで，図 3 の 11，12 行目をみると，lockDir が null ならば，デフォルト値として"java.io.tmpdir"に対応するシステムプロパティの値を用いている．本解析では，システムプロパティのデフォルト値として"C:/test/tmpdir"と事前に定義している．また，図 3 の 8，9 行目をみると，file オブジェクトのパス名がディレクトリであれば，IOException が投げられている．

LockableFileWriter オブジェクトがインスタンス化されている単体テストケースは，7 箇所特定された．そのうち 4 つのケースでは，file オブジェクトのファイル名として"testlockfile"とが与えられており，1 つのケースでは，別の名前"io"が与えられていた．実際は，後者のケースの"io"はディレクトリ名を指すため，IOException が投げられる．残りの 2 つのケースでは，意図的に NullPointerException を投げるようにするため，file の値として null が与えられている．よって，本実験では解析の対象外とする．

第一の実験と同様の手順で，前述した 5 つの単体テストケースを呼び出すエントリーポイントとして main() メソッドを対象コードに追加し，文字列解析を適用し，createLock() メソッド内の lockFile オブジェクトのパス名を推論した．その結果，以下のような推論結果が得られた．

$$
\begin{array}{lll}
S & \rightarrow & ABC \\
A & \rightarrow & \texttt{C:/test/tmpdir/} \\
B & \rightarrow & \texttt{io} \mid \texttt{testlockfile} \\
C & \rightarrow & .\texttt{lck}
\end{array}
$$

".lck"は，LockableFileWriter クラスのどこかで定義された文字列定数である．推論結果には，"C:/test/tmpdir/io.lck"が含まれているが，"C:/test/tmpdir/io"は，ディレクトリ名であるため，実際は"C:/test/tmpdir/io.lck"というファイルは存在しない．よって，

```
1:  public LockableFileWriter(File file, String encoding, boolean append,
2:                                String lockDir) throws IOException {
3:    super();
4:    file = file.getAbsoluteFile();
5:    if (file.getParentFile() != null) {
6:      FileUtils.forceMkdir(file.getParentFile());
7:    }
8:    if (file.isDirectory()) {
9:      throw new IOException("File specified is a directory");
10:   }
11:   if (lockDir == null) {
12:     lockDir = System.getProperty("java.io.tmpdir");
13:   }
14:   File lockDirFile = new File(lockDir);
15:   FileUtils.forceMkdir(lockDirFile);
16:   testLockDir(lockDirFile);
17:   lockFile = new File(lockDirFile, file.getName() + LCK);
18:   createLock();
19:   out = initWriter(file, encoding, append);
20: }
```

図 3 org.apache.commons.io.output.LockableFileWriter コンストラクタ

実行時に"C:/test/tmpdir/io.lck"にアクセスすることはなく，createLock() メソッドが呼び出される前に，IOException が投げられる．本稿で用いた文字列解析器はフローを考慮していないため，推論結果には上記に述べたような余分な文字列が含まれることがある．したがって，最小権限の原則にいまだ反することになるが，任意のファイルにアクセスを許可するというセキュリティ・ポリシーに比べて，はるかに適切なポリシーの設定が可能であるといえる．

**適用実験 3**

第 3 の実験は，配列を含むものである．下記のように，org.apache.commons.io.FileUtils.toURLs (File[] files) メソッドは，引数に File クラスの配列を受け取り，その配列を URL クラスの配列に変換している．

```
1:public static URL[] toURLs(
2:   File[] files) throws IOException {
3: URL[] urls = new URL[files.length];
4: for (int i=0; i<urls.length; i++) {
5:  urls[i] = files[i].toURL();
6: }
7: return urls;
8:}
```

上記 5 行目のループ内で呼び出される File.toURL() は，セキュリティに関与する操作である．また，toURLs メソッドは，下記のようなテストケースから呼び出される．

```
File[] files = new File[] {
  new File(getTestDirectory(),
                          "file1.txt"),
  new File(getTestDirectory(),
                          "file2.txt"),
};
URL[] urls = FileUtils.toURLs(files);
```

toURLs() メソッド内の files の値を文字列解析によって推論したところ，以下のようになった．

$$S \rightarrow ABC$$
$$A \rightarrow \texttt{C:/test/tmpdir/}$$
$$B \rightarrow \texttt{test/io/}$$
$$C \rightarrow \texttt{file1.txt} \mid \texttt{file2.txt}$$

本稿で用いる文字列解析器は，配列に代入されるオブジェクトの順序を考慮しないため，どの配列要素に対して，どのオブジェクトが入るかまでを推論することはできない．つまり，file[0] と file[1] の値は，いずれも"C:/test/tmpdir/test/io/(file1.txt | file2.txt)"と推論される．実際は，例えば file[0] は，"C:/test/tmpdir/test/io/file1.txt"となるため，解析結果には不要な推論が含まれている．しかし，実行時にはアクセスされないファイルパス名が含まれているというわけではないので，セキュリティ・ポリシーの設定という観点では，本解析結果は妥当であるといえる．

**適用結果のまとめ**

表 1 は，各適用実験における SWORD4J と文字列解析の実行時間（秒）を示している．実行環境は，Intel Core2 Duo 2.0GHz プロセッサ，メモリ 3.0G で，Java™2 Platform Standard Edition Runtime Environment Version 5.0 Update 11 である．

なお，実行時間はプロファイリングツールなどを利用して厳密に測定したものではなく，表1は単に相対的な比較を行い，文字列解析が妥当な実行時間で解析できることを示すことを目的としている．

| 適用実験 | 実行時間（秒） | |
|---|---|---|
| | SWORD4J | String analysis |
| 1 | 240 | 18 |
| 2 | 210 | 15 |
| 3 | 240 | 18 |

表1 SWORD4J と文字列解析の実行時間

表2は，各適用実験において文字列解析によって推論された文字列値の数を示している．SWORD4J のカラムに示す記号"???"は，文字列値が推論できなかったことを示す．つまり，SWORD4J で解析した場合，セキュリティ・ポリシーは，「任意のリソースにアクセス可能である」と推論されることになる．一方文字列解析では，適用実験1から3において，それぞれ 12, 2, 2 個の文字列値を推論した．前述したように，適用実験2と3における解析結果には，依然として不要な推論が含まれているが，文字列解析によって，「無数の可能性」から「2つだけの可能性」へと，不要な保守性が大きく削減されているといえる．

| 適用実験 | 推論された文字列セットの数 | |
|---|---|---|
| | SWORD4J | String analysis |
| 1 | ??? | 12 |
| 2 | ??? | 2 |
| 3 | ??? | 2 |

表2 SWORD4J と文字列解析によって推論された文字列セットの数

3.1 節で述べたように，Commons IO ライブラリ内の 146 箇所のセキュリティに関与する操作のうち，SWORD4J は `FilePermission` に関連する 117 箇所の操作に対して，適切なアクセス権限を推論することができなかった．本稿では，その 117 箇所の操作のうち3箇所について文字列解析を適用し，その有効性を示している．残りのセキュリティに関与する操作についても，議論や実験を行う必要があるのは当然であるが，本稿で対象とした3箇所と同じようなタイプの操作がほとんどであるため，同様の手順で実験を行うことで，残りの操作についても同様の結果を得られることが期待できる．

本実験から，文字列解析によって，完全ではないにせよ多くの不要な保守性を削減することで，より適切なセキュリティ・ポリシーを推論できることが分かる．

## 4. 議　　論

### 4.1 無限個の値を取り得る変数

本実験において，文字列解析器は，各解析対象変数に対して有限セットの値を推論した．しかし，一般には，ある変数の取り得る値が無数に存在する場合がある．例えば，以下のようなメソッドについて考える．

```
1:  public File freshFile() {
2:   unsigned int i = 0;
3:   File file;
4:   do {
5:    file = new File("log_"
6:                    + i
7:                    + ".txt");
8:    i++;
9:   } while (file.exists());
10:  return file;
11: }
```

上記9行目の変数 `file` のファイルパス名は，(`unsigned int` 型変数 `i` の値の範囲に制約がないとみなせば)，実行時に取り得る値は有限の範囲に収まらない．変数 `file` の値の文字列解析器による推論結果は，以下の文法で表すことができる．ただし下の文法は，文字列解析器の出力を可読性を考慮して整形したものである．

$$
\begin{aligned}
S &\rightarrow L\,I\,T \\
L &\rightarrow \texttt{log_} \\
I &\rightarrow D \mid \texttt{1}\ Ds \mid \ldots \mid \texttt{9}\ Ds \\
D &\rightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
Ds &\rightarrow D \mid D\ Ds \\
T &\rightarrow \texttt{.txt}
\end{aligned}
$$

上記の推論結果は，$S$ を開始記号として導出される文字列が，`log_0.txt`, `log_1.txt`, ..., `log_100.txt`, ... と無限に存在することを示す．

Java2 セキュリティ機構で利用するセキュリティ・ポリシーでは，リソース名を文脈自由文法で記述することはできず，正規表現風のワイルドカードを用いる程度の記述しかできない．したがって，上記のように，推論された文法の言語が無限言語となるような場合は，1) 既存の手法によって文脈自由言語を正規言語に近似し，2) 正規言語を正規表現として表すことで取り扱うことができる．例えば，上記の例のファイルパス名は，`"log_[1-9][0-9]*\.txt"` のような POSIX 風の正規表現によって表される．

ここで近似を行う理由は，与えられた仕様に対して言語の包含関係を検査するため**ではない**ことに注意されたい．アクセス権解析は，チェックされるべき何らかの仕様を必要としているのではなく，適切な仕様そのものを求めるための解析である．このため，正規表現への近似は，適切な Java2 セキュリティ・ポリシーを記述することができない場合だけに必要とされる．セキュリティ・ポリシーのリソース名の記述として文脈自由文法が利用できるのであれば，近似は必要ではない．

### 4.2 他のプログラムへの適用

本稿では，Jakarta Commons IO ライブラリを文字列解析対象プログラムとした．本解析の他のプログラムに対する適用性を判断するため，Eclipse 3.2 の一部である `org.eclipse.update` パッケージのソースプログラムに

対して，SWORD4J と適用した．その結果，124 箇所のセキュリティに関与する操作のうち，FilePermission に関連する 85 箇所の操作に対して，適切なアクセス権を設定することができなかった．

例えば，org.eclipse.update.internal.configurator パッケージに含まれる ConfigurationActivator クラスには，writePlatformConfigurationTimeStamp() というメソッドが存在する．このメソッド内で，以下のように java.io.FileOutputStream クラスがインスタンス化されている．

```
stream = new DataOutputStream
  (new FileOutputStream(
                configArea
                + File.separator
                + NAME_SPACE
                + File.separator
                + LAST_CONFIG_STAMP));
```

FileOutputStream オブジェクトのファイルパス名は，5 つの文字列の連結によって生成される．File.separator はシステムプロパティで，NAME_SPACE と LAST_CONFIG_STAMP は static final フィールドなどで定義された文字列定数である．したがって，これらを文字列解析で推論するのは容易である．一方，変数 configArea は上記よりも多少複雑で，以下のように値が代入されている．

```
String configArea =
  configLocation.getURL().getFile();
```

変数 configLocation は，org.eclipse.osgi.service.datalocation.Location インタフェース型のローカル変数である．インタフェース型の変数であるため，インタフェース型を実装したクラスのオブジェクトが代入されている単体テストケースのコードなど作成し，そのコードを解析に含める必要がある．あるいは，プログラムの事前定義として，getURL() メソッドの返り値に固定の値を設定しておくなどの方法がある．

## 5. まとめ

本稿では，Jakarta Commons IO ライブラリのコード内の 3 箇所のセキュリティに関与する操作に関連する変数に対して文字列解析を適用し，セキュリティ・ポリシーの推論における不要な保守性を大幅に削減できることを示した．文字列解析を適用することで，ファイル名のようなセキュリティ・ポリシーの設定の際に必要にとなる文字列値を推論することができる．その結果，従来の手法では適切に推論できなかったセキュリティ・ポリシーを妥当な実行時間で推論することが可能であると分かった．

## 謝　辞

本稿の適用実験を行うにあたり，有益なコメントを下さった IBM T.J. Watson Research Center の Marco Pistoia 氏と Julian Dolby 氏に感謝いたします。

## 参考文献

1) L.Koved, M.Pistoia, and A.Kershenbaum. Access rights analysis for java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 359–372, New York, NY, USA, 2002. ACM Press.
2) Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM Press.
3) A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS'03: Proceedings of International Static Analysis Symposium, volume 2695 of LNCS*, pages 1–18. Springer-Verlag, 2003.
4) Java String Analyzer. http://www.brics.dk/JSA/.
5) SWORD4J: Software Workbench Development Environment for Java. http://www.alphaworks.ibm.com/tech/sword4j.
6) J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
7) Jakarta Commons IO library. http://jakarta.apache.org/commons/io/.
8) WALA: T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.