正規表現にマッチする機密情報が出力されないことを保証する Wasm バイナリ静的検査手法

角村 駿[1,a)]　大塚 真太朗[1,b)]　江本 健斗[1,c)]

概要：近年，複数の小さなサービスの集まりとして単一の Web サービスを構成するマイクロサービスアーキテクチャの採用が増加している．このアーキテクチャはスケールが容易であるものの，サービス間通信の増加によって情報漏洩のリスクが高まるという問題がある．本研究は，マイクロサービスの WebAssembly バイナリに対し，特定の正規表現パターンにマッチする機密情報文字列が出力され得ないことを保証する静的検査手法を提案する．本手法は，対象バイナリの抽象実行から出力文字列の式木を構築し，正規表現から構築した DFA に受理される出力文字列となる入力文字列の存在を SMT ソルバに問うことで実現する．

キーワード：静的検証, SMT ソルバ, 正規表現, Wasm

Static Verification that a Wasm Binary Does not Output Secret Information Matching a Given Regex

Kakumura Shun[1,a)]　Otsuka Shintaro[1,b)]　Emoto Kento[1,c)]

Abstract: Recently, microservice architectures have been widely adopted for large-scale web services, in which an entire service is composed of small, independent services (microservices). While this architecture improves scalability in both development and operation, it increases the risk of information leakage due to the growing volume of data communication among microservices. This research proposes a static verification method to certify that a given Wasm program does not output any sensitive information matching a specified regex. Given a Wasm binary, the method constructs a computation tree representing its outputs via the abstract interpretation, and then queries an SMT solver to determine whether the DFA derived from the given regex can reach an accepting state by the outputs computed through the computation tree.

Keywords: static verification, SMT solver, regular expression, Wasm

1. はじめに

近年，複数の小さなサービス（マイクロサービス）の集まりとして大規模な Web サービスを構成するマイクロサービスアーキテクチャの採用が増加している [5–8]．このアーキテクチャは開発と運用の両面においてスケールが容易であるものの，サービス間通信の増加によって情報漏洩のリスクが高まるという問題がある．

情報漏洩への対応として，しばしば，機密情報の匿名化・仮名化の処理が行われる．例えば，Google Cloud の DLP（Data Loss Prevention）のサービスである Cloud DLP [2] では，機密情報のパターンを正規表現で指定し，文字列ストリーム中のパターンにマッチする部分を匿名化・仮名化することができる．

各マイクロサービスのコードに対し，その入力から出力に至るデータフローが必ず機密情報の匿名化・仮名化処理を通過していることをコンパイラで確認すれば，サービスをデプロイする前に情報漏洩のリスクがないことをある程

[1] 九州工業大学
Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[a)] kakumura@pl.ai.kyutech.ac.jp
[b)] otsuka@pl.ai.kyutech.ac.jp
[c)] emoto@pl.ai.kyutech.ac.jp

度保証することができる．例えば，全てのデータ型に対して 1 ビットの「匿名化・仮名化処理済み」の情報を付加した型システムを考えればよい．しかし，この単純なアプローチでは，入力ストリームに対して「機密情報を指定する正規表現にマッチしない形に可逆に変換する」という処理を適用し，そのストリームを匿名化・仮名化処理に通し，その後に逆変換の処理を適用すれば，検査をパスしつつ情報漏洩を起こせてしまう．例えば，クレジットカード番号を機密情報だと思えば，数字 4 文字の連続の繰り返しとして `[0-9]{4}-[0-9]{4}-[0-9]{4}-[0-9]{4}` のような正規表現での指定を行うことになるが，このパターンは「0–9 の数字を A–J のアルファベットに置換する」という可逆の変換を適用すれば，匿名化・仮名化処理を通過して安全であると誤判定させつつ機密情報を出力できてしまう．

本研究は，検査対象の Wasm バイナリと機密情報のパターンを表す正規表現とを入力とし，その Wasm バイナリの出力がその正規表現にマッチする文字列を絶対に含まないことを保証する，静的検査手法（すなわち，プログラムの実行前に安全性を証明する手法）を提案する．本手法は，抽象実行と SMT ソルバの使用に基づく．具体的には，まず，Wasm バイナリを解析しやすい形に変換して抽象実行し，出力ストリームの各バイトが入力ストリームの各バイトからどう計算されるかの式木を構築する．そして，与えられた正規表現から DFA を構成し，「その DFA が出力ストリームの式木から計算される値で受理状態に至る」という論理式が何らかの入力ストリームにより充足可能であるかを SMT ソルバに問う．この論理式が充足不可能であれば，検査対象の Wasm バイナリは正規表現にマッチする機密情報を絶対に出力しないことを保証できる．また，Web サービスのテキストフィールドではしばしば入力項目の事前検査を正規表現で行うため，それに対応した解析として入力ストリームに対する正規表現での制限も同様に導入する．本手法は “力技” であるが，実験により，小さなプログラムに対して現実的な時間で検査が終了することが示された．

本論文の構成は以下のとおりである．まず，第 2 節にて既存の事柄についての導入を行う．次に，第 3 節にて提案手法を導入する．そして，第 4 節にて提案手法の評価を示す．最後に，第 5 節にて関連研究を述べ，第 6 節にて本論文をまとめる．

2. 準備

本節では，本論文で用いる既存結果について導入する．

2.1 Wasm

Wasm（WebAssembly）[10] は，W3C の勧告するスタックベースの命令セットアーキテクチャである．C++ や Rust を含む様々なプログラミング言語からコンパイルされ，各種のウェブブラウザやその他の環境で実行可能である．Wasm を対象とすることにより，開発言語に依存しない解析手法を構築できる．Wasm は，通常のバイナリ形式のほか，相互変換可能なテキスト形式（wat: WebAssembly Text Format）をもつ．

```
1  (module
2   (import "wasi_snapshot_preview1" "fd_write"
3     (func $fd_write (param
4         i32 ;; file descriptor
5         i32 ;; IoVec pointer
6         i32 ;; num of IoVec
7         i32 ;; address to store # of written bytes
8        ) (result
9         i32 ;; return value
10       )))
11  (memory (export "memory") 1)
12  (data (i32.const 0) "Hello␣World!\n")
13  (data (i32.const 16) ;; IoVec
14    "\00\00\00\00" ;; iov_base = 0
15    "\0d\00\00\00" ;; iov_len = 13
16  )
17  (func (export "_start") ;; entry point
18    (local $i i32)
19    i32.const 0
20    local.set $i
21    (loop $loop
22      i32.const 1 ;; fd=1 (stdout)
23      i32.const 16
24      i32.const 1
25      i32.const 24
26      call $fd_write ;; fd_write(1,16,1,24)
27      drop ;; discards the result
28      local.get $i
29      i32.const 1
30      i32.add
31      local.tee $i ;; $i = $i + 1
32      i32.const 5
33      i32.lt_u ;; $i < 5 ?
34      br_if $loop ;; goto $loop if true
35    )))
```

図 1 wat 形式の Wasm の例：“Hello World” を 5 回出力

図 1 に Wasm の例として “Hello World!\n” を 5 回出力するプログラムを示す．各行の `;;` より後はコメントである．

ファイルシステムやネットワークの操作などは，ランタイムが提供する関数を import して使用する．その様な関数セットとして，WASI（WebAssembly System Interface）[11] が提案されている．本研究では WASI の Preview1 を利用する．図 1 の 2–10 行目は WASI Preview1 の `fd_write` 関数を利用するための import である．

Wasm はスタックベースであるため，演算や関数の実行は引数を `i32.const` などでスタックに積んで行う．図 1 の 18 行目でローカル変数 `$i` を宣言した後，19・20 行目は `$i = 0` の代入を，22–26 行目は `fd_write(1,16,1,24)` の関数呼び出しを 28–31 行目は `$i = $i + 1` の加算と代

```
1 (declare-const x Int)
2 (define-fun even ((z Int)) Bool (= (mod z 2) 0))
3 (assert (and (>= x 7) (even x)))
4 (check-sat)
5 (get-model)
```

図 2 Z3 への入力（SMT ファイル）の例

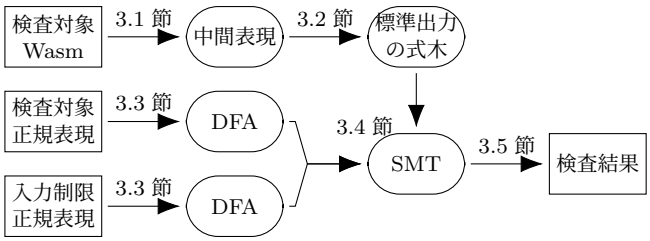

図 3 提案手法の概観

入を行っている．32–34 行目は，$i < 5 の比較を行い，真であれば 21 行目のループ先頭へ戻る．ほか，Wasm にはアドレスを指定してバイト単位で読み書き可能な線形メモリ空間やグローバル変数がある．

2.2 SMT ソルバ Z3

SMT（Satisfiability Modulo Theories）ソルバは，与えられた数学的な論理式の充足可能性を判定するソフトウェアである．本研究では，高性能な SMT ソルバとして知られる，Microsoft Research の開発する Z3 [4] を用いる．

Z3 への入力（SMT ファイル）は図 2 のようなテキストファイルである．図 2 は，1 行目で未決定の変数 x を宣言し，2 行目で偶数判定の関数 even を定義し，3 行目で x に関する論理式（7 以上であり，かつ，偶数である）を assert で宣言し，4 行目でその論理式が充足可能であるか（すなわち，条件を満たす x が存在するか）を Z3 に問う．この場合，例えば x が 8 であれば条件を満たすため，Z3 からは充足可能であることを意味する sat という出力を得る．もし充足可能でなければ，unsat という出力を得る．充足可能である場合，6 行目の (get-model) により，条件を満たす具体的な x の値も出力できる．

図 2 では整数型のみを用いたが，他にも IEEE754 規格の浮動小数点数や配列なども利用できる．

3. 提案手法

本節では，本研究の提案手法を導入する．本手法は，検査対象の Wasm バイナリ（以下，検査対象 Wasm）と，出力に現れないことを証明したい機密情報のパターンを表す正規表現（検査対象正規表現）を入力にとり，検査対象 Wasm の出力に検査対象正規表現にマッチする文字列が含まれ得ないことを証明する．なお，検査対象 Wasm の入出力は標準入出力で行われると仮定する．

図 3 に提案手法の概観を示す．以降，図 4 を検査対象 Wasm として用い，各部を説明する．このプログラムは，標準入力から 128 バイトの入力をバッファにコピーし，先頭のバイトをその値に依存した定数値（64 以下で ‘0’，それ以外で ‘A’）に書き換え，その次のバイトをその値に依存しない定数値 ‘X’ に書き換え，最後にバッファの中身を全て標準出力に出力する．

```
1  (module
2  ;; fd_read, fd_write のimport は紙面の都合で略
3    (memory (export "memory") 1)
4    (data (i32.const 128)
5        "\00\00\00\00"
6        "\80\00\00\00"
7    )
8    (func (export "_start")
9  ;; メモリの 0 番地に標準入力の 128 バイトをコピー
10     i32.const 0
11     i32.const 128
12     i32.const 1
13     i32.const 126
14     call $fd_read
15     drop
16 ;; 読まれたバイト数（126 番地）を保存（132 番地）
17     i32.const 132
18     i32.const 126
19     i32.load
20     i32.store
21 ;; 0 番地の値は 64 以下か？
22     i32.const 0
23     i32.load8_u
24     i32.const 64
25     i32.le_u
26     (if (then   ;; 真なら 0 番地の値を'0'に
27       i32.const 0
28       i32.const 48
29       i32.store8
30     ) (else     ;; 偽なら 0 番地の値を'A'に
31       i32.const 0
32       i32.const 65
33       i32.store8
34     ))
35 ;; 1 番地の値を'X'に置き換える
36     i32.const 1
37     i32.const 88
38     i32.store8
39 ;; 読まれたバイト数分を 0 番地から標準出力へ
40     i32.const 1
41     i32.const 128
42     i32.const 1
43     i32.const 126
44     call $fd_write
45     drop
46   ))
```

図 4 提案手法の説明のための Wasm 例（wat）

3.1 検査対象 Wasm の解析

まず，検査対象 Wasm の解析を行い，後続の処理のための前処理として，主にスタックを介した処理記述から変数

```
_start() {
;; メモリの 0 番地に標準入力の 128 バイトをコピー
    i32 v0 = 0;
    i32 v1 = 128;
    i32 v2 = 1;
    i32 v3 = 126;
    i32 v4 = $fd_read(v0, v1, v2, v3);
    drop(v4);
;; 読まれたバイト数（126 番地）を保存（132 番地）
    i32 v5 = 132;
    i32 v6 = 126;
    i32 v7 = i32.load(v6);
    i32.store(v5, v7);
;; 0 番地の値は64 以下か？
    i32 v8 = 0;
    i32 v9 = i32.load8_u(v8);
    i32 v10 = 64;
    i32 v11 = i32.le_u(v9, v10);
    if (v11) {   ;; 真なら 0 番地の値を'0'に
        i32 v12 = 0;
        i32 v13 = 48;
        i32.store8(v12, v13);
    } else {     ;; 偽なら 0 番地の値を'A'に
        i32 v14 = 0;
        i32 v15 = 65;
        i32.store8(v14, v15);
    }
;; 1 番地の値を'X'に置き換える
    i32 v16 = 1;
    i32 v17 = 88;
    i32.store8(v16, v17);
;; 読まれたバイト数分を 0 番地から標準出力へ
    i32 v18 = 1;
    i32 v19 = 128;
    i32 v20 = 1;
    i32 v21 = 126;
    i32 v22 = $fd_write(v18, v19, v20, v21);
    drop(v22);
}
```

図 5 図 4 の検査対象 Wasm の解析結果（内部表現）

を用いた記述に変更する．図 4 の Wasm を解析した結果の中間表現の pretty print を図 5 に示す．具体的な処理はスタックへの push に応じて新たな変数を導入しつつ，演算子等がスタックを pop する際に対応する変数を引数に渡す形となる．

3.2 抽象実行による式木の構築

次に，前節の解析結果をインタプリタで抽象実行 [3] し，標準出力を表す式木を構築する．図 5 を抽象実行した結果を図 6 に示す．最初の行は出力が得られる条件が `True`（常に得られる）であることを意味し，以降の行はこのときの標準出力の各バイト（配列 `stdout` の要素として表示）が標準入力（配列 `stdin` の要素として表示）からどう計算されるかの式木である．例えば，図 6 の `stdout[0]` は，標準出力の最初のバイトが標準入力の最初のバイト `stdin[0]` の値に応じて計算されることを表している．

```
precondition = True
stdout[0] = if (stdin[0] <= 64) then 48 else 65
stdout[1] = 88
stdout[2] = stdin[2]
stdout[3] = stdin[3]
    ...
stdout[127] = stdin[127]
```

図 6 図 5 の抽象実行結果：標準出力の各バイトの式木

抽象実行のためのインタプリタは，前節の解析で導入された変数と，Wasm の線形メモリ・ローカル変数・グローバル変数の状態を保持しつつプログラムを実行し，標準出力の式木を構築する．この抽象実行では，標準入力の各バイトを値の定まってない変数（単純な式木）として扱い，式木との演算は式木を返すように，値同士の計算は値を返すようにしてプログラムを実行する．これにより，あらかじめ計算できる部分を計算したコンパクトな式木を構築する．外部から import している関数の呼び出しについては，その仕様に従ってインタプリタの内部状態を直接書き替える形で実現する．また，条件分岐についてはその条件が式木である場合，then 節と else 節の両方を独立に実行し，その結果として変化のあったメモリや変数の情報を if 式としてマージする．

3.3 DFA の構築

検査対象正規表現の前後に任意文字列を表す正規表現 `.*` を連結し，その正規表現に対応する DFA（決定性有限オートマトン）を構築する．これは，後段の処理で出力全体と正規表現とのマッチングを行うからである．例えば，検査対象正規表現が 16 個の数字の並びを表す `[0-9]{16}` の場合，正規表現 `.*[0-9]{16}.*` の DFA を構築する．

DFA の構築は標準的な手法を用いる．すなわち，正規表現からまず NFA を構築し，部分集合構成法で DFA に変換した後に DFA の最小化を行う．

構築した DFA は，状態遷移を表す写像 State → Char → State と，受理状態を表す写像 State → Bool で表現される．

入力側の制限も考える場合，すなわち，特定の正規表現にマッチした入力しか与えられない場合の検査を行う際には，その入力側の正規表現については直接 DFA を構築して用いる．

3.4 SMT ファイルの生成

抽象実行で構築された式木と正規表現から構築した DFA を元に，Z3 で検証するための SMT ファイルを生成する．この SMT ファイルは，DFA の定義，標準出力による DFA の状態遷移の定義，検査の定義からなる。

DFA の定義は，構築した DFA の状態遷移と受理状態を表す写像の関数定義 `next` と `accept` である．具体的には，図 7 のように，状態遷移は Z3 の理解する if 式（`ite`）を

```
1  ; 状態遷移の写像を埋め込み
2  (define-fun next ((state Int) (char (_ BitVec 8)))
       Int
3      ; 状態0 なら，読んだ文字が数字なら状態2 へ，それ以
           外は0 へ
4      (ite (= state 0) (ite (and (bvule #x30 char) (
           bvule char #x39)) 2 0)
5      ; 状態1 なら，無条件に状態1 へ
6      (ite (= state 1) 1
7      ; 状態2 なら，読んだ文字が数字なら状態3 へ，それ以
           外は0 へ
8      (ite (= state 2) (ite (and (bvule #x30 char) (
           bvule char #x39)) 3 0)
9      ; 中略
10     ; 最後，どの分岐でもないときの状態保持
11     state)))
12 )
13 ; 受理状態の写像の埋め込み
14 (define-fun accept ((state Int)) Bool
15     ; 状態が受理状態 1
16     (or (= state 1))
17 )
```

図 7 SMT ファイル：DFA の定義の例

用いて写像を定義し，受理状態の判定は状態（state）が受理状態の番号であるかの等式を or で連結して定義する．

標準出力による DFA の状態遷移の定義は，第 3.2 節で構築した標準出力の式木を DFA の状態遷移関数に与える形で，標準出力の各バイトにより遷移した状態をひとつひとつの変数として連続的に定義する．

図 8 に図 6 の式木での状態遷移の定義を示す．8 行目の state0_0 が初期状態（0 番目の状態）を表す変数であり，具体的な値として DFA の初期状態番号 0 を与えている．続く 9 行目で次の 1 番目の状態が変数 state0_1 として定義されている．具体的には，ひとつ前の状態である state0_0 と，標準出力の最初のバイトに対応する式木（12 行目）を引数として状態遷移関数 next を呼び出すことで得られる値という定義になっている．ここで，式木の中の標準入力を表す部分は，2 行目で宣言している配列 stdin の要素へのアクセス（select stdin 0）になっている．同様に，14 行目は 2 番目の状態を変数 state0_2 として定義し，以降同様に続く．また，出力の式木の生成の前提条件が True ではない場合，それぞれの条件に対応した状態遷移が同様に生成される．

最後に，検査部分の定義として「標準出力を DFA に入力した際の最後の状態が受理状態である」という論理式を assert で宣言し，その充足可能性を（check-sat）で問う（図 9）．

以上の部分を結合した単一 SMT ファイルが Z3 による検証に用いられることになる．

もし，入力側にも正規表現での制限を入れる場合には，同様にして DFA を標準入力で駆動した結果が受理状態であるという論理式を追加する．例えば，入力側

```
1  ; 標準入力を配列として宣言
2  (declare-const stdin (Array Int (_ BitVec 8)))
3  (define-const precondition0 Bool
4    ; ある出力が得られる条件の式木
5    ; 定数True
6    true
7  )
8  (define-const state0_0 Int 0)
9  (define-const state0_1 Int (next state0_0
10   ; 標準出力の 0 文字目を表す式木
11   ; 標準入力の 0 文字目が 64 以下で'0'、それ以外で'A'
12   (ite (bvule (select stdin 0) #x40) #x30 #x41)
13 ))
14 (define-const state0_2 Int (next state0_1
15   ; 標準出力の 1 文字目を表す式木
16   ; 常に'X'
17   #x58
18 ))
19 (define-const state0_3 Int (next state0_1
20   ; 標準出力の 2 文字目を表す式木
21   ; 標準入力の 2 文字目
22   (select stdin 2)
23 ))
24 ; 以降，3 文字目以降が続く（略）
25 ; 標準出力のリストが 2 つ以上ある場合は続けて生成
```

図 8 SMT ファイル：標準出力による状態遷移の定義（図 6 に対応）

```
1  (assert (or
2      (and precondition0 (accept state0_127))
3      ; 出力木の場合分けが複数ある場合は続けて生成
4  ))
5  (check-sat)
```

図 9 SMT ファイル：検査の定義

の DFA の受理状態の判定関数を acceptI とし，入力での状態遷移の最後の状態を stateI_127 とすれば，図 9 の 2 行目を（and precondition0 (accept state0_127) (acceptI stateI_127)）として入力側の条件を論理式に含める．

3.5 Z3 による検証

前節で生成した SMT ファイルを Z3 に読み込ませ，検証を行う．

結果が sat であれば，ある入力に対して検査対象正規表現にマッチする出力が検査対象 Wasm から得られることを意味する．すなわち，プログラムは危険でああり，検査は不合格である．

他方，unsat が出力されれば，どの様な入力に対しても検査対象 Wasm から検査対象正規表現にマッチする標準出力が得られないことを意味するため，プログラムは安全であり検査は合格である．

3.6 制約

本手法で扱える検査対象 Wasm には以下の制約がある．

```
void main() {
  char buf[128];
  fd_read(stdin, buf, 128);
  fd_write(stdout, buf, 128);
}
```

図 10 標準入力をそのまま出力するプログラム

表 1 計算時間の測定に用いた計算機環境

CPU	Intel(R) Core(TM) Ultra 7 265K (3.90 GHz)
RAM	128 GB (32 GB x4, DDR5-5600)
OS	Ubuntu 24.04.2 LTS / WSL / Winodws11 Pro 25H2
z3	z3 4.15.7

本質的な制約として，標準入力の値によらず十分短い時間で停止する Wasm プログラムしか扱えない．例えば，無限にループするようなプログラムは抽象実行を終えられず，また，現状では標準入力に応じて繰り返し回数が変化するループには対応しない．

実行時にアクセスするメモリアドレスが標準入力に依存する場合にも対応しない．

標準入力から指定したバイト数が読み込まれなかった場合を考慮していない．この点については，通常のプログラムでは実際の読み込みバイト数の確認などを行うため，それによって検査結果が間違うことは稀だと考えられる．

3.7 高速化の工夫

以上では，生成される SMT ファイルにおける標準入力の扱いを配列（`Array`）としていた（図 8）が，配列の扱いが難しいためかこのままでは Z3 による検証にかかる時間が大きいという問題が生じる．そこで，この問題の解消として，この配列の各バイトを新たな変数として定義する工夫を導入する．具体的には，例えば配列の 29 番目の要素に対応する変数を `(declare-const rand29 (_ BitVec 8))` と定義する．このような変数の定義を入力の長さ分列挙し，配列変数の代わりに用いることで，Z3 による検証の高速化が期待できる．

4. 評価実験

検査対象 Wasm を複数用意し，検査結果が正しいことの確認と，実行時間の測定を行った．用意した検査対象 Wasm は，標準入力から読み込んだ文字をそのまま出力するもの（図 10 に対応），標準入力から読み込んだ文字のうち半角数字全てを 'X' に書き替えて出力するもの（図 11 に対応），標準入力から 32 文字読み込み前半と後半を互い違いに入れ替えて出力し再び元に戻して出力するもの（図 12 に対応）の 3 つである．また，実行時間の測定には，表 1 に示す仕様の計算機を用いた．

図 10〜12 のプログラムに対応する 3 つの Wasm に対し，正規表現 `[0-9]{16}` を検査対象正規表現としてそれぞれ

```
void main() {
  char buf[128];
  fd_read(stdin, buf, 128);
  for (int i = 0; i < 128; i++) {
    if ('0' <= buf[i] && buf[i] <= '9') {
      buf[i] = 'X';
    }
  }
  fd_write(stdout, buf, 128);
}
```

図 11 標準入力の数字を X に書き替えて出力するプログラム

```
void main() {
  char buf[32];
  fd_read(stdin, buf, 32);
  for (int i = 0; i < 8; i++) {
    swap(buf[2 * i], buf[16 + 2 * i]);
  }
  fd_write(stdout, buf, 32);
  for (int i = 0; i < 8; i++) {
    swap(buf[2 * i], buf[16 + 2 * i]);
  }
  fd_write(stdout, buf, 32);
}
```

図 12 標準入力からの 32 文字をシャッフルして出力したのち，再びシャッフルして戻して出力するプログラム

検査を行った結果，想定通りに図 11 は検査に合格し，他は検査不合格となった．これは，図 11 の出力は数字を含まないため検査対象正規表現にマッチする文字列を絶対に含まないのに対し，他は標準入力に適当な文字列を与えることで検査対象正規表現にマッチする文字列を出力できるからである．

同様に，表 2 に，クレジットカード番号を表す正規表現 `([0-9]{4}-){3}[0-9]{4}`，メールアドレスを表す正規表現 `[a-zA-Z]+@([a-zA-Z]+\.)*[a-zA-Z]+`，パスポート番号を表す正規表現 `[A-Z]{2}[0-9]{7}`，いくつかの特定のワードを表す正規表現 `RM-ORANGE|RM-YELLOW|RM-GREEN`（例として Google DLP API のドキュメントにある病室名の例を用いた）に対しても検査を行った結果を示す．

表 2 には，検査にかかった時間もあわせて記載している（9 回の中央値）．また，与えられた正規表現から構築された DFA の状態数も記載している（正規表現の前後に任意の文字列にマッチする正規表現 `.*` が付けられてから DFA が構築されることに注意する）．

数字を必ず含む正規表現 `([0-9]{4}-){3}[0-9]{4}` と `[A-Z]{2}[0-9]{7}` に対する検査結果は，正規表現 `[0-9]{16}` と同様であり，図 11 のみが合格で他は不合格となった．正規表現 `[a-zA-Z]+@([a-zA-Z]+\.)*[a-zA-Z]+` と `RM-ORANGE|RM-YELLOW|RM-GREEN` に対しては，どのプログラムもこれにマッチする出力を出し得るので，すべて不合格という結果となった．

表 2 検査の結果と実行時間（秒）：入力制限無し

検査対象正規表現（DFA 状態数）	図 10	図 11	図 12
`[0-9]{16}`	不合格	合格	不合格
（17 状態）	1.58	3.19	0.74
`([0-9]{4}-){3}[0-9]{4}`	不合格	合格	不合格
（20 状態）	3.25	4.28	1.13
`[a-zA-Z]+@([a-zA-Z]+\.)*[a-zA-Z]+`	不合格	不合格	不合格
（4 状態）	0.23	0.82	0.06
`[A-Z]{2}[0-9]{7}`	不合格	合格	不合格
（10 状態）	0.70	2.64	0.18
`RM-ORANGE|RM-YELLOW|RM-GREEN`	不合格	不合格	不合格
（19 状態）	2.16	5.32	0.94

表 3 検査の結果と実行時間（秒）：入力を `.*` で制限

検査対象正規表現（DFA 状態数）	図 10	図 11	図 12
`[0-9]{16}`	不合格	合格	不合格
（17 状態）	1.63	3.35	0.58
`([0-9]{4}-){3}[0-9]{4}`	不合格	合格	不合格
（20 状態）	3.15	5.33	1.09
`[a-zA-Z]+@([a-zA-Z]+\.)*[a-zA-Z]+`	不合格	不合格	不合格
（4 状態）	0.27	0.83	0.06
`[A-Z]{2}[0-9]{7}`	不合格	合格	不合格
（10 状態）	0.72	2.53	0.29
`RM-ORANGE|RM-YELLOW|RM-GREEN`	不合格	不合格	不合格
（19 状態）	2.50	5.77	0.76

計算時間を見てみると，いずれも現実的な時間で検査ができている．また，DFA の状態数が増えると計算時間も増す傾向にあるが，検査されるプログラムにより時間の差が見られる．

次に，サービスへの入力が予め正規表現で制限されている状況に対応した検査として，入力側にも正規表現での制限を入れた場合の結果を示す．入力側の正規表現を `.*` とした場合の結果を表 3 に，`[0-9]*` とした場合の結果を表 4 に示す．前者は実質の制限はなく，入力側の DFA を導入したことによる計算時間の変化を確認するものである．後者は数値しか入力として与えられないことを意味した状況である．

表 3 と表 2 と比較すると，入力に実質の制限がないため，検査の結果に変化はない．実行時間については，多くの場合で入力側 DFA の駆動に伴う若干の増加が確認できる．

入力を数字列のみに制限した場合（表 4）は，出力され得ない文字を必ず含む正規表現（`([0-9]{4}-){3}[0-9]{4}` や `[A-Z]{2}[0-9]{7}` や `RM-ORANGE|RM-YELLOW|RM-GREEN` にマッチする出力は行われないため，これらに対する検査が合格になっている．計算時間については，多くの場合で短縮される結果となった．これは，入力側の DFA（状態数 3）を駆動する分の処理の増加はあるものの，それによる探索空間の削減の効果が大きい結果であると考えられる．

表 4 検査の結果と実行時間（秒）：入力を `[0-9]*` で制限

検査対象正規表現（DFA 状態数）	図 10	図 11	図 12
`[0-9]{16}`	不合格	合格	不合格
（17 状態）	1.46	2.55	0.37
`([0-9]{4}-){3}[0-9]{4}`	合格	合格	合格
（20 状態）	2.08	3.29	0.47
`[a-zA-Z]+@([a-zA-Z]+\.)*[a-zA-Z]+`	合格	合格	合格
（4 状態）	0.50	0.82	0.09
`[A-Z]{2}[0-9]{7}`	合格	合格	合格
（10 状態）	0.93	1.54	0.20
`RM-ORANGE|RM-YELLOW|RM-GREEN`	合格	合格	合格
（19 状態）	2.17	4.01	0.45

5. 関連研究

文字列の型として正規表現を考える，文字列処理のためのラムダ計算の拡張が提案されている [9]．この型検査の実装は困難であるとされているが，ある程度の制限のもとで現実的な型検査器の実装ができたならば，本研究の意図した静的検査が型レベルで効率的に行えることになり，より大規模なプログラムの検査も期待できる．

本論文の提案手法で現状扱える正規表現は，純粋な正規表現である．しかし，現実的なプログラミングでは先読みや後方参照などの拡張を用いることも多い．先読み付きの正規表現は依然として正規表現と同等であることが示されているため [1, 12]，先読みについては本研究の手法でそのまま扱える．ただし，実際の計算時間がどの程度になるかは明らかではない．後方参照については正規表現の表現力を超えるため，DFA を用いている提案手法では扱えない．DFA よりも高度なオートマトンを駆動する形に手法を拡張する必要がある．

6. おわりに

本研究は，Wasm バイナリに対し，正規表現で指定した機密情報文字列を出力し得ないことを保証する静的検査手法を提案した．本手法は，Wasm の抽象実行と SMT ソルバによる充足可能性の判定に基づく．実験により，本手法は “力技” であるものの，小さなプログラムに対して現実的な時間で検査を終了することが示された．

今後の課題として，DFA や式木の SMT ファイルへの埋め込みの工夫や，SMT ソルバで正規表現に関する理論を直接用いることでの高速化が挙げられる．また，対応する Wasm 命令の追加，WASI Preview2 やそれ以降への対応，日本語を含む Unicode への対応も挙げられる．

謝辞

本研究は，JST，CREST，JPMJCR21M4 の支援を受けたものである．また，本研究は JSPS 科研費 JP24K14898 の助成を受けたものである．

参考文献

[1] Berglund, M., van der Merwe, B. and van Litsenborgh, S.: Regular Expressions with Lookahead, *Journal of Universal Computer Science*, Vol. 27, No. 4, pp. 324–340 (2021).
[2] Cloud, G.: Sensitive Data Protection, `https://cloud.google.com/security/products/sensitive-data-protection`. Accessed: 2026-02-11.
[3] Cousot, P. and Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, ACM, pp. 238–252 (1977).
[4] De Moura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, Springer, pp. 337–340 (2008).
[5] Guo, L., Moorthy, A., Chen, L.-H., Carvalho, V., Mavlankar, A., Opalach, A., Prakash, A., Swanson, K., Tweneboah, J., Venkatrav, S. and Zhu, L.: Rebuilding Netflix Video Processing Pipeline with Microservices, `https://netflixtechblog.com/4e5e6310e359` (2024). Accessed: 2026-02-11.
[6] Lewis, J. and Fowler, M.: Microservices, `https://martinfowler.com/articles/microservices.html` (2014). Accessed: 2026-02-11.
[7] Newman, S.: *Building Microservices, 2nd Edition*, O'Reilly Media, Inc. (2021).
[8] Schwarz, M. and Neverov, A.: Up: Portable Microservices Ready for the Cloud, `https://www.uber.com/en-JP/blog/up-portable-microservices-ready-for-the-cloud/` (2023). Accessed: 2026-02-11.
[9] Tabuchi, N., Sumii, E. and Yonezawa, A.: Regular Expression Types for Strings in a Text Processing Language, *Electronic Notes in Theoretical Computer Science*, Vol. 75, pp. 95–113 (2003).
[10] W3C: WebAssembly, `https://webassembly.org/`. Accessed: 2026-02-11.
[11] W3C: WebAssembly System Interface, `https://wasi.dev/`. Accessed: 2026-02-11.
[12] 森畑明昌：先読み付き正規表現の有限状態オートマトンへの変換，コンピュータ ソフトウェア， Vol. 29, No. 1, pp. 1_147–1_158 (2012).