Secure Compiler Seminar (2006/6/6)

米澤研 M2　佐藤秀明

今回の内容

- 言語処理系の技術をウェブアプリケーションに適用
  - 動的生成ページの妥当性を概算
  - Sanitize が必要な箇所を検知
  - 実行されるコマンドの安全性をチェック

動的生成ページの妥当性概算

題材

- Static Approximation of Dynamically Generated Web Pages
  - Yasuhiko Minamide.
  - In Proceedings of the 14th International Conference on World Wide Web (WWW 2005).
- プログラムが生成するhtmlの文法を静的に検査
  - プログラムを解析
  - 文脈自由文法(CFG)で近似的に表現

String Analyzer

- プログラムの出力する文字列を CFG で表現

```
<?php
for ($i = 0; $i < $n; $i++)
  $x = "0".$x."1";
echo $x;
?>
```

PHP source code

```
$x : /abc|xyz/
$n : int
```

input specification
（プログラムに対する入力）

string analyzer

$$
\begin{array}{lcl}
X & \rightarrow & abc \\
X & \rightarrow & xyz \\
X & \rightarrow & 0X1
\end{array}
$$

CFG

CFG の生成手順 (1/3)

```
$i = 0; $S = "";
echo "abc";
while ($i < 10) {
   $S = $S."xy";
   $i = $i + 1;
}
$T = str_replace("x","xx",$S);
echo $T;
```

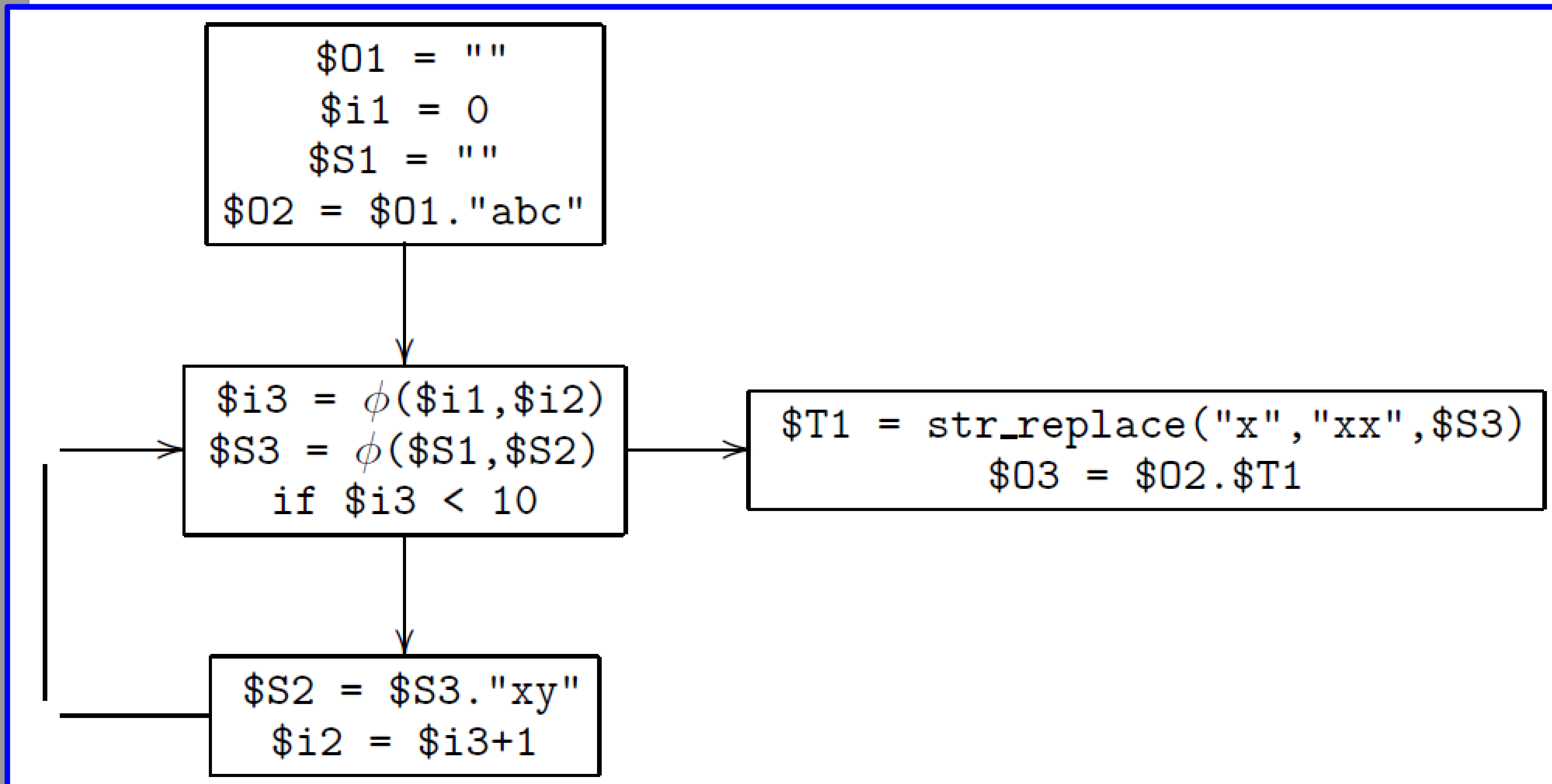
```
$O = "";
$i = 0; $S = "";
$O = $O."abc";
while ($i < 10) {
   $S = $S."xy";
   $i = $i + 1;
}
$T = str_replace("x","xx",$S);
$O = $O.$T;
```

元のプログラム

文字列出力

特殊変数への連結

CFG の生成手順 (2/3)



静的単一代入 (SSA) 形式に変換

CFG の生成手順 (3/3)

$$
\begin{aligned}
O_1 &\rightarrow \epsilon \\
S_1 &\rightarrow \epsilon \\
O_2 &\rightarrow O_1abc \\
S_3 &\rightarrow S_1 \\
S_3 &\rightarrow S_2 \\
S_2 &\rightarrow S_3xy \\
T_1 &\rightarrow \texttt{str_replace}(x, xx, S_3) \\
O_3 &\rightarrow O_2T_1
\end{aligned}
$$

$S3 = ϕ($S1,$S2)

文字列操作について
CFG を抽出

文字列操作関数を
どうやって CFG に
落とすか？

文字列操作関数の除去 (1/2)

1. 関数の引数を表す CFG を特定
2. 関数自体を transducer で表現
   - Transducer: 出力機能つきオートマトン
3. 1. と 2. から関数の返り値を表す CFG を求める
   - アルゴリズムは既知

- ただし関数が CFG で閉じている場合のみ適用可能
  - strtolower(), str_replace(), trim(), ...
- 閉じていない場合は返り値を CFG で近似

文字列操作関数の除去 (2/2)

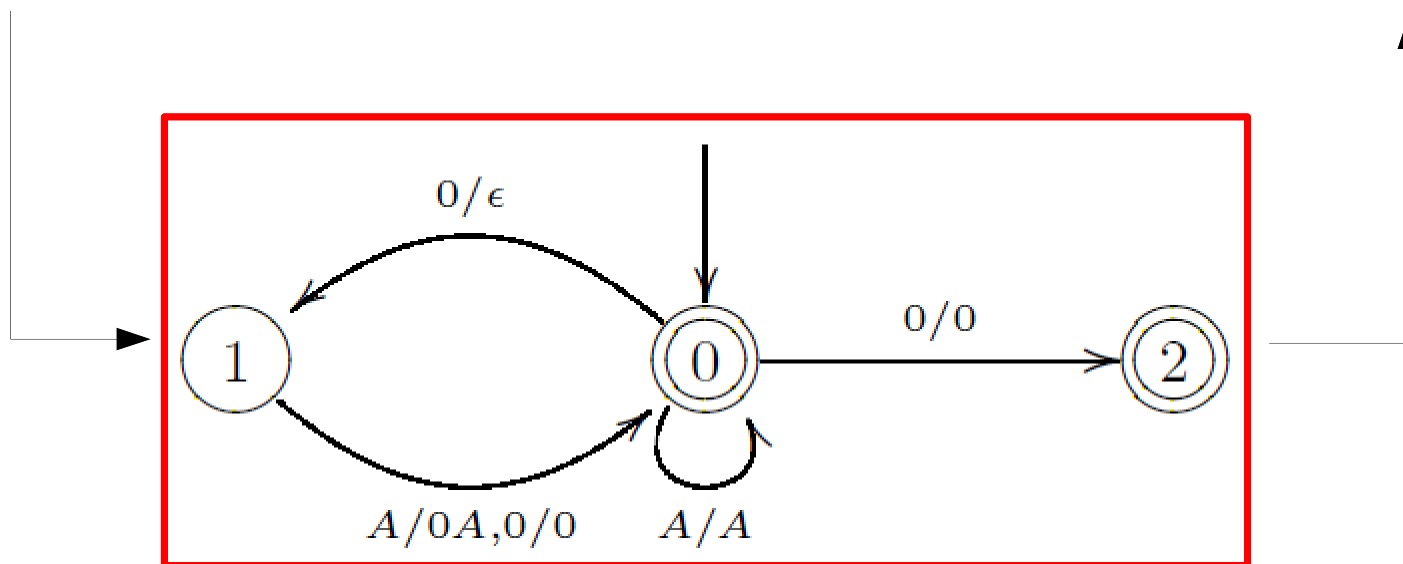
- 返り値を表す CFG を求める例

$$\begin{array}{lcl} X & \to & abc \\ X & \to & xyz \\ X & \to & 0X1 \end{array}$$

引数

$$\begin{array}{lcl} S & \to & abc \mid xyz \mid X1 \\ X & \to & 0abc \mid 0xyz \mid 0S1 \end{array}$$

返り値



str_replace(“00”, “0”, $x) を表す transducer

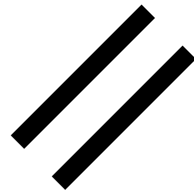
返り値の近似 (1/2)

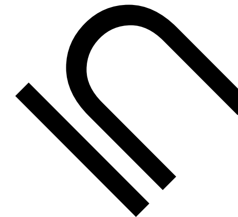
- 関数を表現するのに pushdown transducer が必要
→ stack の状態は忘れることにする
  - strip_tags()
- もっと複雑な関数の場合→正規表現で近似
  - md5() : [0-9a-f]{32} と表現しておく
  - str_shuffle(), str_repeat(), crypt(), sha1(), ...
- Rich な正規表現関数
→正規表現の構造から近似 CFG を構成
  - 置換操作など : マッチする CFG を transducer で抽出

返り値の近似 (2/2)

- 関数の循環適用が存在
  →出力し得る全文字集合の「0 個以上出現」で近似
  - かなり荒い近似だが滅多にないので許容

$$
\begin{array}{lcl}
S & \rightarrow & 01 \\
S & \rightarrow & \texttt{str_replace}(0, x0y, T) \\
T & \rightarrow & \texttt{str_replace}(1, 1z, S)
\end{array}
$$





$\{x^n 0 y^n 1 z^n \mid n \geq 0\}$

文脈自由でない

$\{x, y, z, 0, 1\}^*$

文脈自由

HTML の Validation

- (生成する HTML の CFG)⊆(正しい HTML の CFG) を確認したい
  - 一般には undecidable
- 生成されるタグのネスト回数は普通は有限
  → decidable にできる！
  1. 正しい HTML のネスト回数を i 回に制限した文法を構成
     - この文法は regular
  2. i=1,2,3,… として包含関係を順にチェック
     - (生成する HTML の CFG)⊆(ネストを i 回に制限した文法)
- 今回はタグのネスト構造のみチェック
  - 計算量の都合

評価 :CFG の構成

- 非常に長い時間がかかるものも
  - 文字列操作関数が 17 連続で適用されていた

Program	# lines	# nonterminals	# productions	Time (sec)
webchess	2224	300	450	0.36
schoolmate	8085	7985	9505	39.92
faqforge	843	180	443	0.16
phpwims	726	82	226	0.13
timeclock	462	656	1233	0.15
tagit	890	858365	6961180	4933.17

評価 :Validation

- 時間がかかる
  - 制限された文法を繰り返し生成
- バグはうまく見つけられた
  - めったに実行されない部分から発見
    →静的解析の有効性

Program	Depth	Bugs	Opt tags	Time (sec)
webchess	9	1	6	123.33
faqforge	10	30	0	45.64
phpwims	9	7	0	63.93
timeclock	14	11	0	145.61
schoolmate	17	14	3	7580.69

Sanitize が必要な箇所の検知

題材

- Static Detection of Security Vulnerabilities in Scripting Languages
  - Yichen Xie and Alex Aiken.
  - To appear in the 15th USENIX Security Symposium (USENIX Security 2006).
- sanitize されていないデータの流れを追跡
  - ユーザ入力がそのまま query に使用される箇所を発見
  - PHP に特化した手法

PHP の特徴 (1): SQL Injection 攻撃

- プログラマの意図しない SQL 文を実行される
  - 例：全ユーザのパスワードが変更される！

```
$rows=mysql_query("UPDATE users SET
    pass='$pass' WHERE userid='$userid'");
```

userid パラメタ： ' OR '1' = '1

```
UPDATE users SET pass='...'
WHERE userid='' OR '1'='1'
```

  - cf. Java だと攻撃されにくいが面倒

```
PreparedStatement s = con.prepareStatement
  ("UPDATE users SET pass = ? WHERE userid = ?");
s.setString(1, pass); s.setInt(2, userid);
int rows = s.executeUpdate();
```

PHP の特徴 (2): 暗黙的なキャスト

- 文字列型とそれ以外の型が seamless に融合

```
if ($userid < 0) exit;
$query = "SELECT * from users
          WHERE userid = '$userid'";
```

- 問題 : 分かりづらい

```
if ($userid == 0) echo $userid;
```

  - 期待 : "0" しか出力されない
  - 実際 : $userid が数字でないなら全て出力される
    - 数字でない文字列は全て 0 にキャストして比較
- SQL Injection に悪用される危険性

PHP の特徴 (3): extract()

- HTTP GET/POST リクエストのパラメタを一括展開

```
$_GET['param1'];
$_GET['param2'];
$_GET['param3'];
      ...
```

```
extract(_GET, EXTR_OVERWRITE);
$param1;
$param2;
$param3;
      ...
```

- 問題：初期化されていない変数を攻撃可能

```
extract($_GET, EXTR_OVERWRITE);
for ($i=0;$i<=7;$i++)
  $new_pass .= chr(rand(97, 122));
mysql_query("UPDATE . . . $new_pass . . .");
```

ユーザが new_pass パラメタをつけていると…

方策

- 出自不明の変数は sanitize/initialize すべき
  - ユーザ入力とわかる変数
    - $_GET、$_POST、$_COOKIE など
  - それ以外の初期化されていない変数
    - extract() によって攻撃され得る変数
- Cast の振舞いを適切に把握すべき
  - ユーザ入力データが SQL query 関数へ流出

ユーザ入力が sanitize なしに
query 関数へ渡される可能性を
静的解析

解析の手順

入力：プログラム、
sanitize を行う関数のリスト、
SQL query を行う関数のリスト

1. プログラムから制御フローグラフ (CFG) を構成
2. 基本ブロックごとにプログラムの実行を simulate
   - データの伝搬を追跡
   - pre/post condition を計算
3. 各ブロック間 / 関数呼び出し間の情報を統合
4. unsanitized input が query 関数に流れる箇所を検知

出力：危険な箇所のリスト

基本ブロックの simulation

- プログラムを抽象的に評価
  - 実行時にしか決まらない要素は不定としておく
- state(Γ): location から value への写像
  - simulation の進行とともに更新される

$$\frac{\Gamma \vdash lv \overset{\text{Lv}}{\Rightarrow} l \qquad \Gamma \vdash e \overset{\text{E}}{\Rightarrow} v}{\Gamma \vdash lv \leftarrow e \overset{\text{S}}{\Rightarrow} \Gamma[l \mapsto v]} \text{ assignment}$$

$$\frac{\Gamma \vdash s_1 \overset{\text{S}}{\Rightarrow} \Gamma' \quad \Gamma' \vdash s_2 \overset{\text{S}}{\Rightarrow} \Gamma''}{\Gamma \vdash (s_1; s_2) \overset{\text{S}}{\Rightarrow} \Gamma''} \text{ seq}$$

String の扱い (1)

- state を用いて alias を管理
  - “$_0$” は初期値を表す

$$\Gamma = \{\text{hash} \Rightarrow \text{hash}_0, \text{key} \Rightarrow \text{key}_0, \text{_POST} \Rightarrow \text{_POST}_0, \text{_POST[userid]} \Rightarrow \text{_POST[userid]}_0\}$$

$$\$\text{hash} = \$\text{_POST};$$
$$\$\text{key} = \text{'userid'};$$

$$\Gamma = \{\text{hash} \Rightarrow \text{_POST}_0, \text{key} \Rightarrow [\text{'userid'}], \ldots\}$$

$$\$\text{userid} = \$\text{hash}[\$\text{key}];$$

$userid は _POST[userid]_0 を指す

String の扱い (2)

- concatenation: 連結する全 substring を把握

```
$a = 'a0';
$b = 'b0';
$c = $a.$b;
```

$\Gamma = \{c \Rightarrow [\text{'}a_0\text{'}, \text{'}b_0\text{'}], \ldots \}$

- contains: 含むかもしれない全 substring を把握

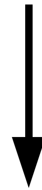
```
function f($a, $b) {
    if (...) return $a;
    else return $b;
}
$ret = f($x.$y, $z);
```

$\Gamma = \{ret \Rightarrow [contains(x, y, z)], \ldots \}$

Boolean の扱い

- untaint: sanitize 関数による効果を管理
  - If による分岐先の基本ブロックへ伝搬される情報

$\$ok = \text{is_numeric}(\$userid);$



$\Gamma = \{ok \Rightarrow untaint(\{userid\}, \{\}), \ldots\}$

untaint({true なら sanitize されている変数のリスト},
{false なら sanitize されている変数のリスト})

Cast の扱い

- PHP の cast rule を忠実に simulate

$$\mathsf{cast}(k, \mathsf{bool}) = \begin{cases} \mathsf{true} & \text{if } k \neq 0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathsf{cast}(\mathsf{true}, \mathsf{str}) = [\text{“}1\text{”}]$$
$$\mathsf{cast}(\mathsf{false}, \mathsf{str}) = []$$

$$\mathsf{cast}(v = [\beta_1, \ldots, \beta_n], \mathsf{bool})$$
$$= \begin{cases} \mathsf{true} & \text{if } (v \neq [\text{“}0\text{”}]) \wedge \bigvee_{i=1}^{n} \neg \mathsf{is_empty}(\beta_i) \\ \mathsf{false} & \text{if } (v = [\text{“}0\text{”}]) \vee \bigwedge_{i=1}^{n} \mathsf{is_empty}(\beta_i) \\ \bot & \text{otherwise} \end{cases}$$

$\cdots$

実験結果

- False Positive なし
  - Err Msgs: $_GET、$_POST などの sanitize 忘れ
  - Warn: それ以外の自由変数の sanitize 忘れ
    - extract() で攻撃され得る変数

	Err Msgs	Bugs (FP)		Warn
e107	16	16	(0)	23
News Pro	8	8	(0)	8
myBloggie	16	16	(0)	23
DCP Portal	39	39	(0)	55
PHP Webthings	20	20	(0)	6
Total	99	99	(0)	115

検知した extract() 脆弱性の一例

```
for ($i=0;$i<=7;$i++)
    $new_pass .= chr(rand(97, 122));
...
$result = dbquery("UPDATE ".$db_prefix."users
  SET user_password=md5('$new_pass')
  WHERE user_id='".$data['user_id']."'");
```

new_pass パラメタに

```
abc'), user_level =' 103', user_aim = ('
```

を渡す

```
UPDATE users SET user_password=md5('abc'),
      user_level='103', user_aim=('???????')
  WHERE user_id='userid'
```

パスワード 'abc' の Super Administrator 権限を設定

関連研究

- Securing Web Application Code by Static Analysis and Runtime Protection
  - Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee and Sy-Yen Kuo
  - In Proceedings of the 13th International Conference on World Wide Web (WWW 2004).
- sanitize されていない変数の流れを捕捉
  - type qualifier ベースのアルゴリズム
  - sanitize を行うコードを自動挿入

実行されるコマンドの安全性チェック

題材

- The Essence of Command Injection Attacks in Web Applications
  - Zhendong Su and Gary Wassermann.
  - In Proceedings of the 33rd Annual Symposium on Principles of Programming Languages (POPL 2006).
- SQL injection 攻撃を阻止
  - SQL 文を実行直前にチェック
  - 構文木ベースの判定アルゴリズム

Sanitize が不十分な例

```
String sanitizedName =
  replace(request.getParameter("name"),"'","''");
String sanitizedCardType =
  replace(request.getParameter("cardtype"),
          "'","''");
String query = "SELECT cardnum FROM accounts"
   + " WHERE uname='" + sanitizedName + "'"
   + " AND cardtype=" + sanitizedCardType + ";";
```

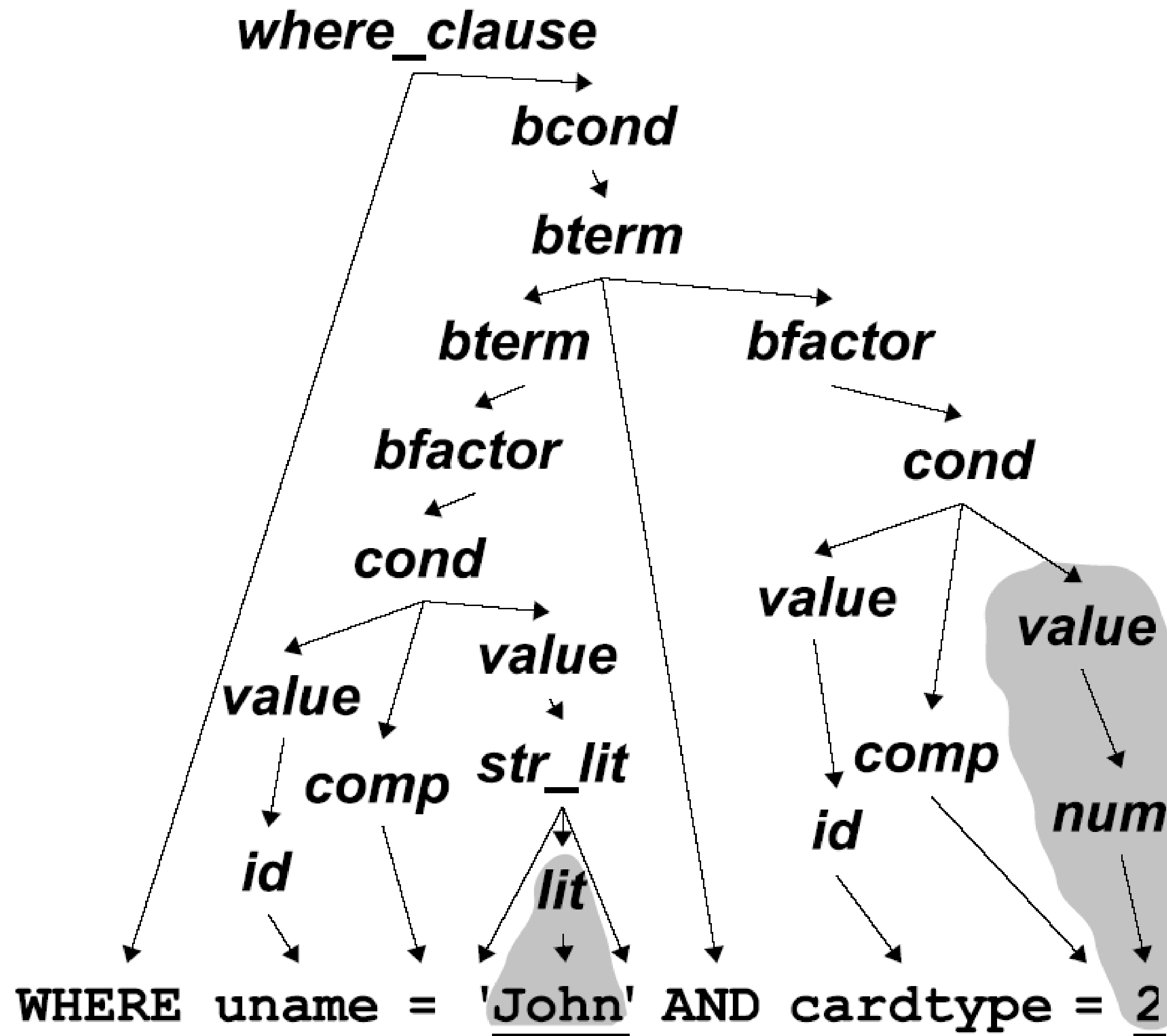
- cardtype に” 2 OR 1=1” を入れると攻撃できる
  - 整数だから quote をサボっている

問題の根源

- SQL 文を単なる string として扱っている
  - 文の構造を考慮していない
- 構文解析の必要性
  - プログラマの意図していない構文構造を警戒

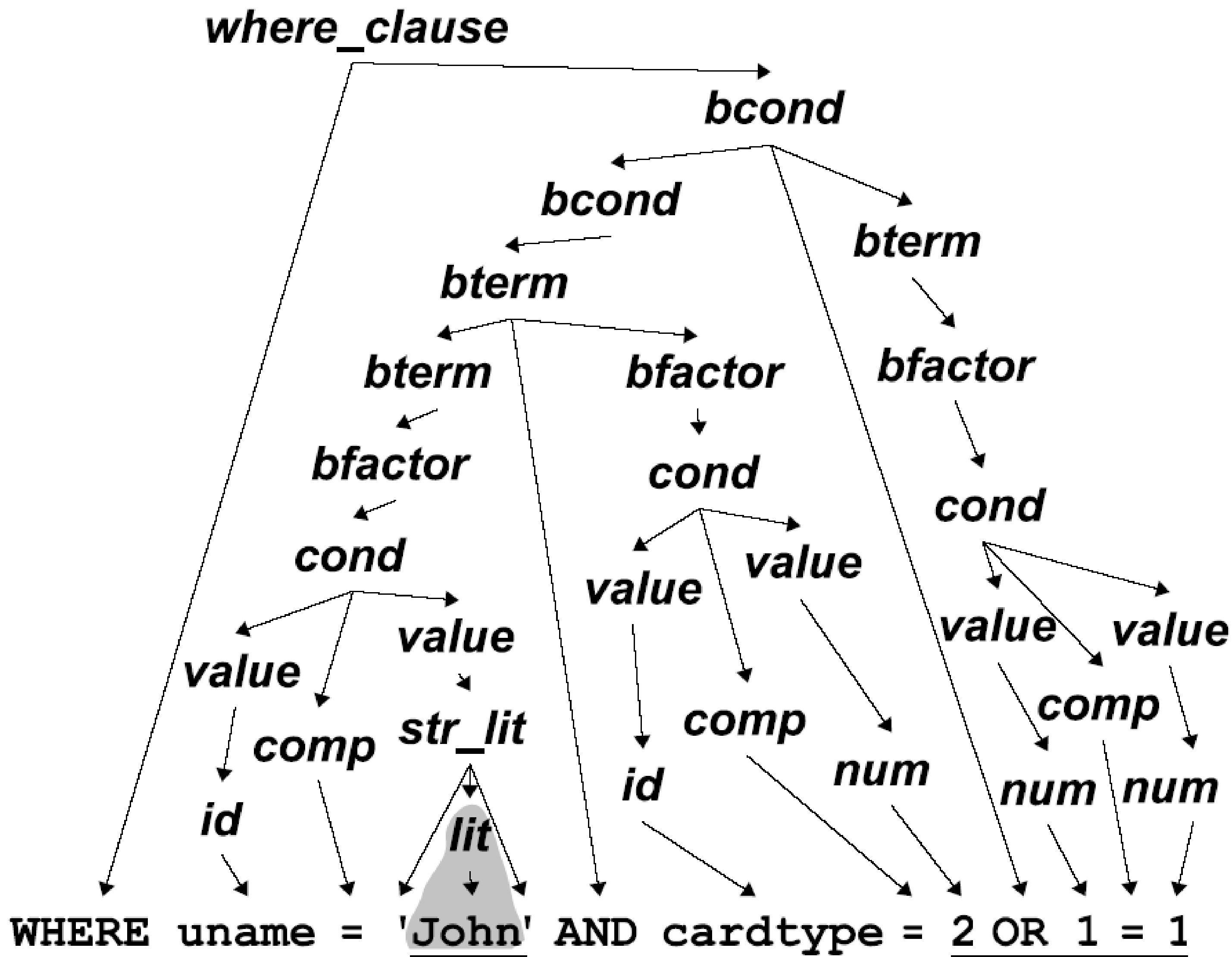
正しい構文の例

- ユーザ入力が subtree の中で閉じている



危険な構文の例

- ユーザ入力が隣の subtree へはみ出している



SQL Injection 攻撃の定義

- 以下の 2 条件を共に満たす SQL 文を攻撃と定義
  1. 全体として valid な構文木をもつ
  2. ユーザ入力が subtree の外へはみ出している
- この定義で十分だろう
  - 不十分な例：入力として“ < 5” を想定

    ```
    query = "SELECT * FROM tbl WHERE col " + input;
    ```

  - こんな例は現実世界には皆無

文法の拡張

- ユーザ入力を特殊記号【】で囲む
  - 【】付きの SQL 文を受理する文法を新たに生成

元の文法 G

```
sentence ::= ... input ...
input       ::= ...
```

ユーザ入力を parse する
subtree に注目

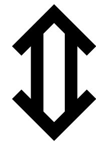
拡張した文法 G^a

```
sentence ::= ... inputᵃ ...
inputᵃ      ::= 【 input 】
input       ::= ...
```

アルゴリズムの正しさ

- 定理(健全性、完全性)

$s \in L(G)$ かつ s は SQL injection 攻撃でない



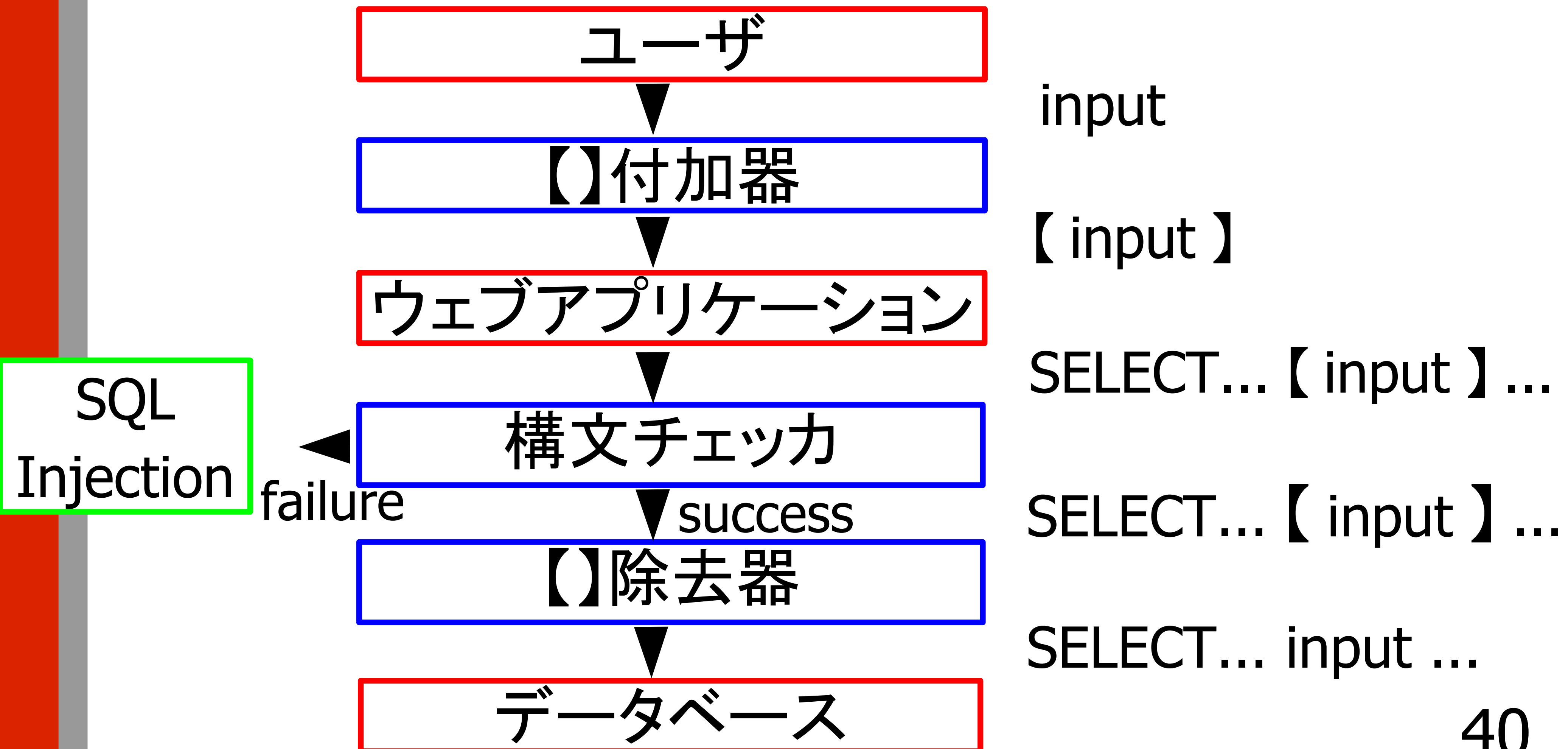
s のユーザ入力箇所を【】で囲んだ文 s^a について

$s^a \in L(G^a)$

  - 証明は省略

処理の手順

- ウェブアプリケーションの実装詳細は知らなくてよい
  - 【】は一般の文字と衝突しないと仮定



特殊記号【】の選び方

- 辞書にないアルファベット4文字の文字列を採用
  - ユーザが【】と同じ文字列を入力する確率を下げる
  - アルファベット以外はアプリ内のsanitizerを通りづらい
- 数字を通すsanitizerはsanitize後に【】を付加

評価の手順

- PHP とJSP の両バージョンで検証
- 【】付加 / 構文チェックの処理コードは手作業で追加
  - 将来的には自動化する予定
- 攻撃リクエストURL は自動生成
  - [Halfond and Orso]* の成果を利用

* William G. J. Halfond and Alessandro Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005).

評価対象のプログラム

- 修正個所は実際の入出力箇所より少ない
  - 同一関数を経由する入出力操作が多い
  - 自動化の際は flow analysis が有用か

Subject	LOC		Query Checks Added	Query Sites	Metachar Pairs Added	External Query Data
	PHP	JSP				
Employee Directory	2,801	3,114	5	16	4	39
Events	2,819	3,894	7	20	4	47
Classifieds	5,540	5,819	10	41	4	67
Portal	8,745	8,870	13	42	7	149
Bookstore	9,224	9,649	18	56	9	121

評価結果

- false positive/false negative なし
- 解析時間も現実的
  - 2-3 ミリ秒

Language	Subject	Queries		Timing (ms)	
		Legitimate (Attempted/allowed)	Attacks (Attempted/prevented)	Mean	Std Dev
PHP	Employee Directory	660 / 660	3937 / 3937	3.230	2.080
	Events	900 / 900	3605 / 3605	2.613	0.961
	Classifieds	576 / 576	3724 / 3724	2.478	1.049
	Portal	1080 / 1080	3685 / 3685	3.788	3.233
	Bookstore	608 / 608	3473 / 3473	2.806	1.625
JSP	Employee Directory	660 / 660	3937 / 3937	3.186	0.652
	Events	900 / 900	3605 / 3605	3.368	0.710
	Classifieds	576 / 576	3724 / 3724	3.134	0.548
	Portal	1080 / 1080	3685 / 3685	3.063	0.441
	Bookstore	608 / 608	3473 / 3473	2.897	0.257