

A Damage Identification Method Based on Syntax Analysis and Semantic Analysis for SQL Injection

メタデータ	言語: jpn 出版者: 公開日: 2020-06-12 キーワード (Ja): キーワード (En): 作成者: 黒木, 琴海, 鐘本, 楊, 青木, 一史, 三好, 潤, 野口, 靖浩, 西垣, 正勝 メールアドレス: 所属:
URL	http://hdl.handle.net/10297/00027516

構文解析と動的意味解析に基づく SQL インジェクション被害識別手法

A Damage Identification Method Based on Syntax Analysis and Semantic Analysis for SQL Injection

黒木 琴海* 鐘本 楊* 青木 一史* 三好 潤* 野口 靖浩†
Kotomi Kuroki Yo Kanemoto Kazufumi Aoki Jun Miyoshi Yasuhiro Noguchi
西垣 正勝†
Masakatsu Nishigaki

あらまし SQL インジェクションは未だに多発しており、個人情報の漏洩といった重大な被害に繋がっている。SQL インジェクションの検知には WAF (Web Application Firewall) が用いられる。WAF は SQL インジェクションを検知することは可能であるが、その攻撃がどのような被害をもたらすかといった分析まではできていない。そのため、被害の分析は人手で行う必要があり時間を要する。本稿では、分析時間の短縮を目的として、SQL インジェクションによる被害を識別する手法を提案する。提案手法では、HTTP リクエスト中に含まれる部分的な SQL 文のみを解析対象として、構文解析によって SQL 文の木構造を推定し、木構造をもとに SQL 文を部分的に実行し挙動を把握する動的意味解析を行った結果から被害を識別する。識別精度の評価によって、人工データセットに対して 83.1%、実環境のデータセットに対して 71.9%の精度で正しく被害を識別できたことを示す。

キーワード 被害識別, SQL インジェクション, 構文解析, 動的意味解析

1 はじめに

インターネットに公開されている Web アプリケーションは攻撃者からのアクセスが容易なため、日々大量の攻撃に晒されている。Web アプリケーションの中には、オンラインショッピングやインターネットバンキングなど、顧客の個人情報やクレジットカード情報といった重要な情報をデータベース (DB) に保存するものも多く存在している。そのため、これらの情報の漏洩を狙う SQL インジェクションが Web アプリケーションに対する攻撃の中でも特に攻撃数が多く深刻な影響をもたらしている。

SQL インジェクションの脅威から Web アプリケーションを守るために用いられているのが WAF (Web Application Firewall) である。WAF は、シグネチャと呼ばれる攻撃の特徴を記した正規表現と、HTTP リクエストの内容がマッチした場合にその HTTP リクエストを攻撃として検知し、遮断を行う。WAF によって検知した攻

撃を全て遮断することで攻撃による被害を防ぐことは可能である。しかしながら、WAF の検知は完璧ではなく現実には正常なリクエストを誤検知してしまう可能性がある。そのため、Web サーバの運用者は誤検知によって正常なリクエストが遮断され、Web アプリケーションの可用性が損なわれることを懸念して、WAF では遮断せず、検知のみを行うという運用を行う場合がある。

WAF で遮断せず、検知のみを行う場合には、Web サーバの運用者は検知された攻撃に対して、その攻撃の成否や、情報漏洩や改竄といった被害を明らかにすることで、その対処の必要性、対処の方法等を決定することができる。しかし、情報漏洩や改竄といった被害を明らかにする部分は人間による判断に頼っているのが現状である。そのため、SQL インジェクションの攻撃コードが複雑であったり、攻撃者によって難読化されてしまうと解析に手間取り、被害の判断に時間を要してしまうという課題が存在する。WAF は SQL インジェクション攻撃であることを検知できても、その攻撃がどのような被害に繋がるかは判断できない。

本稿では、HTTP リクエストに含まれる SQL 文の断片のみを分析対象として、SQL インジェクションがも

* NTT セキュアプラットフォーム研究所, 〒 180-8585 東京都武蔵野市緑町 3-9-11. NTT Secure Platform Laboratories, 3-9-11 Midori-cho, Musashino-shi, Tokyo 180-8585 Japan.

† 静岡大学情報学部, 〒 432-8011 静岡県浜松市中区城北 3 丁目 5-1. 3-5-1 Johoku, Naka-ku, Hamamatsu-shi, Shizuoka 432-8011 Japan.

表 1: 各被害をもたらす SQL インジェクション例

被害の種類	攻撃例
脆弱性の偵察	a' and SLEEP(5)-- a' UNION SELECT CONCAT(CHAR(116),CHAR(101),CHAR(115),CHAR(116))-- a' AND EXTRACTVALUE(1,CONCAT(0x5c73,(SELECT (ELT(1=1,1))),0x61))--
情報漏洩 (システム情報)	a' UNION SELECT VERSION()-- a' UNION SELECT table_name FROM information_schema.tables--
情報漏洩 (DB 内容)	a' UNION SELECT name,password FROM users--
改竄	a'; DROP TABLE users--

たらず被害を識別する手法を提案する。まず、SQL インジェクションによる被害を 4 つの種類に分類し、攻撃に含まれる SQL クエリに対して構文解析および動的意味解析を行うことで、攻撃が成功した際にどの種類の被害を引き起こすかを識別する。構文解析では攻撃者が挿入しようとしている SQL クエリを木構造に変換する。提案手法の特徴となる動的意味解析では、木構造に表現された SQL クエリを部分的に実行 (エミュレーション) しその挙動を把握する。最後に、挙動に関する情報が付与された木構造の SQL クエリに対して複数のルールに基づいて被害を識別する。提案手法により、Web サーバの運用者は攻撃者によって挿入された SQL クエリを自ら解析する時間を必要とせず、得られた被害情報に基づいて、被害有無の確認や対処に移ることが可能となる。

本研究の貢献は以下の通りである。

- 攻撃対象の DB や Web アプリケーションの情報やシステムの改変を必要としない SQL インジェクションの被害識別手法を設計した。提案手法では、攻撃者が HTTP リクエストに挿入した SQL クエリのみを解析対象として、その攻撃が成功した場合にどのような被害が発生するかを識別する。これによって、可用性の観点から監視対象のシステムへ手を加えたりシステムの情報を得ることが困難な場合や、多数のシステムを監視しなければならない場合などでも被害を推測することが可能となる。
- 識別精度の評価によって、提案手法が、人工的に作成したデータセットに対して 83.1%、実環境で得られたデータセットに対して 71.9%の攻撃を正しく識別できることを示した。

2 SQL インジェクションによる被害

2.1 被害の種類

著者らは過去の攻撃データや自ら運用している Web サーバに対する攻撃の分析を通じて SQL インジェクションによる被害の種類を以下の 4 つと定義した。

- 脆弱性の偵察
- 情報漏洩 (システム情報)

- 情報漏洩 (DB 内容)

- 改竄

各被害をもたらす SQL インジェクションの例を表 1 に示す。以下、それぞれの被害について詳しく解説する。

脆弱性の偵察 この被害は、攻撃対象の Web アプリケーションに対して、SQL インジェクションの脆弱性の有無を確認するための攻撃が行われることを指す。表 1 の 1 つ目の攻撃例では、“SLEEP(5)” という関数によって HTTP レスポンスを遅延させることで、SQL インジェクションの可否を判断している。2 つ目の攻撃例では、CONCAT(CHAR(116),CHAR(101),CHAR(115),CHAR(116)) を実行した結果である “test” という文字列が HTTP レスポンスに含まれるか否かで SQL インジェクションの可否を判断している。3 つ目の攻撃例では、特定のエラーが発生するような SQL クエリを挿入し、レスポンスにエラーメッセージが含まれるか否かで SQL インジェクションの可否を判断している。

情報漏洩 (システム情報) この被害は DB システムのバージョン情報やホスト名、DB システムのユーザ名、スキーマ情報など、DB システムに関する情報が漏洩することを指す。攻撃例としては、表 1 に挙げたような、“VERSION()” という関数によって DB システムのバージョン情報が漏洩したり、特定の DB システムにデフォルトで存在する “information_schema.tables” というテーブルから攻撃対象の DB に存在するテーブル名が漏洩したりといったものが存在する。

情報漏洩 (DB 内容) この被害は DB サーバに保存されているテーブルの内容が漏洩することを指す。ただし、DBMS にデフォルトで存在するシステムテーブルの漏洩は情報漏洩 (システム情報) 被害として定義するため、本被害ではサーバの運用者が作成したテーブルや Web アプリケーションによって作成されたテーブルの内容の漏洩を対象とする。攻撃例としては、表 1 に挙げたような、“users” というテーブルから “name” と “password” というカラムの内容を漏洩させるものが存在する。

改竄 この被害は、データベースの内容が追加・変更・削除される被害を指す。攻撃例としては表 1 に挙げるよ

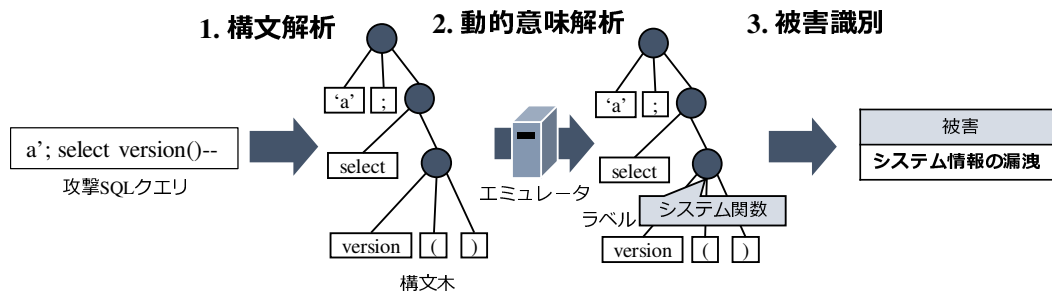


図 1: 提案手法の概要図

うな “users” というテーブルを消去するものが存在する。

2.2 被害識別における課題

WAF や既存の攻撃検知手法 [1,2] では攻撃を検知することが可能だが、その攻撃の被害を識別することはできない。正規表現によるシグネチャマッチを用いて SQL インジェクションの被害が発生したか否かを判別する手法として DIAVA [3] が存在する。DIAVA は、情報漏洩を引き起こす攻撃に対して、その攻撃が成功した際に出力される文字列を検知することで情報漏洩の有無を識別する手法である。しかし、この手法の課題は、漏洩した文字列が HTTP レスポンスに含まれないブラインド SQL インジェクションなどには対応ができないことである。また、漏洩した情報の種類や、脆弱性の偵察や改竄といった他の種類の被害については識別ができない。

この課題を解決する手法として、SQL クエリの実行時の挙動から被害を識別する方法が考えられる。実行時にアクセスしたテーブルや実行された関数等を取得することで被害の識別が可能である。攻撃対象の DB で実際に実行された SQL クエリの挙動を観測するか、攻撃対象の DB で実行される SQL クエリ全体と SQL の実行環境を用意できれば容易に挙動を把握することができる。しかしながら実際には、可用性の観点から監視対象のシステムに手を加えることが困難であったり、多数のシステムを同時に監視していて DB や Web アプリケーションの情報が容易に把握できなかったりする。そういった場合には、HTTP リクエストに含まれる、攻撃者によって挿入された断片的な SQL クエリのみを用いて被害を識別する必要がある。攻撃者によって挿入された断片的な SQL クエリは、実行される SQL クエリの一部でしかないため、そのまま実行することはできない。

本稿では、断片的な SQL クエリに対して構文解析を行って構文木に変換した上で部分木毎に実行した際の挙動から、その攻撃が成功した際の被害を識別する方法を提案する。これによって、攻撃対象のシステムの情報や改変を必要とせず、攻撃者によって挿入された断片的な SQL クエリのみをもとにした被害の識別を可能とする、

3 提案手法

図 1 に提案手法の概要を示す。提案手法では、攻撃として検知された HTTP リクエストから抽出した攻撃者が挿入する攻撃 SQL クエリを入力として想定している。HTTP リクエストが攻撃か否かの判別は WAF や既存の攻撃検知手法で実現可能である。また、WAF には検知した攻撃をハイライトする仕組みが実装されているものもあるため、この仕組みを応用することで攻撃者が挿入した攻撃 SQL クエリのみを HTTP リクエストから抽出することが可能であると考えられる。そのため、本稿では攻撃 SQL クエリは抽出できているものとして扱う。

提案手法の処理は 1. 構文解析, 2. 動的意味解析, 3. 識別の 3 つのステップに大別される。2.2 節で述べた通り、攻撃 SQL クエリは実際に実行される SQL クエリの一部でしかないため、そのまま実行することができない。そのため、まず 1. 構文解析で攻撃 SQL クエリを構文木に変換し、実行可能な単位に分割する。その後、2. 動的意味解析にて構文木をもとに部分木単位で実行した際の挙動から、構文木に対してラベル付けを行う。構文木に対して付与されたラベルをもとに、3. 識別にて被害の種類を識別する。以下、各節にて各ステップの処理を詳説する。

3.1 構文解析

この処理では入力された攻撃 SQL クエリに対してあらかじめ定義した構文ルールによって SQL クエリを解析し、SQL クエリを構文木として表現する。攻撃 SQL クエリは実行される SQL クエリの全体ではなくあくまで攻撃者が挿入したい部分のみであるため、このままでは構文解析した場合、括弧や引用符が不足している場合が多く解析できない場合が多い。そのため、まず、入力された SQL クエリが構文解析できるよう括弧および引用符の補完を行う。

括弧の補完では、攻撃 SQL クエリに對にならない括弧がある場合に対応する括弧を先頭または末尾に追加する。引用符の補完とは異なり、攻撃 SQL クエリ内の対応する括弧の有無の判断が容易なため、構文解析を行う

表 2: 構文解析におけるトークン一覧

トークン名	説明	例
SEMICOLON	クエリの終端を表す記号	;
SQL-WORD	SQL の構文に含まれる単語	SELECT AND OR FROM
NAME	アルファベット, 数字, 一部の記号で構成される変数名, 関数名等を表す文字列	abc123 @@HOSTNAME dbname.tbname
COMMA	要素を並べるための記号	,
R-OP	比較演算子	!= <> = >= <= > <
A-OP	算術演算子	+ - * / && ~ & ::
NUMBER	数値	10 0x6d 1.5 5e13
STRING	エスケープされた文字列	"text" '123'
ASTERISK	SQL におけるワイルドカード	*
L-PAREN	左括弧	(
R-PAREN	右括弧	)

```

<sql-query> ::= <statements>
<statements> ::= <statement> | [ <statement> ] ( <SEMICOLON> <statement> )+
<statement> ::= <items> | [ <items> ] ( <SQL-WORD> [ <items> [ <NAME> ] ] )+
<items> ::= <condition> ( <COMMA> <condition> )*
<condition> ::= <expression> [ <R-OP> <expression> ]
<expression> ::= <value> | [ <value> ] ( <A-OP> <value> )+
<value> ::= <function> | <parenthesis> | <NAME> | <NUMBER> | <STRING> | <ASTERISK>
<function> ::= <NAME> <L-PAREN> [ <statement> ] <R-PAREN>
<parenthesis> ::= <L-PAREN> [ <statement> ] <R-PAREN>

```

図 2: 提案手法における攻撃 SQL クエリの構文

前に必要な括弧を補完する。

引用符の補完では、攻撃 SQL クエリに単一引用符や二重引用符が欠けている場合に攻撃 SQL クエリの先頭と末尾に補完を行う。まず補完しない状態で一度構文解析を行い、構文エラーでかつ攻撃 SQL クエリに引用符が含まれている場合に、引用符が欠けていると判断して引用符の補完を行う。例えば a' OR 1=1--を入力された攻撃 SQL クエリとする。この場合、引用符が不足しているため攻撃クエリの先頭と末尾に引用符の補完を行い 'a' OR 1=1 --' として扱う。

括弧と引用符を補完した攻撃 SQL クエリに対して構文解析を行う。構文解析では、まず表 2 に示したトークンに攻撃 SQL クエリを分割した上で、定義した文法に従って構文木に変換する。定義した文法を EBNF (Extended Backus-Naur Form) で記述したものを図 2 に示す。図 2 において、表 2 に示したトークンが終端記号となる。

例として、攻撃 SQL クエリに対して構文解析を行い、得られた構文木を図 3 に示す。構文木は、ノードとエッジで構成される木構造で表される。構文木には葉ノードにあたる Token ノードと中間ノードにあたる Type ノードの 2 種類のノードが存在する。Token ノードは図 2 における終端記号で、表 2 で示したトークンを表す。Type ノードは図 2 における非終端記号を表す。

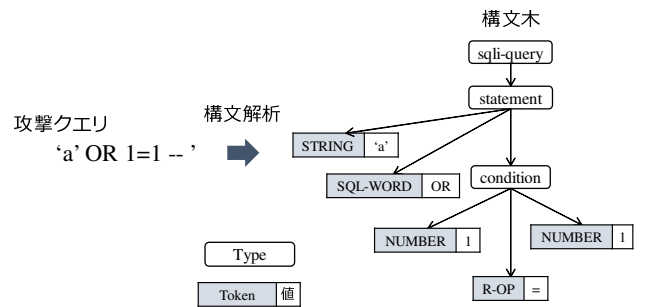


図 3: 構文解析例

3.2 動的意味解析

3.1 節にて得られた構文木に対して、構文木にその部分の意味を示す意味ラベルを付与する。このラベル付与は予め定めた複数の意味解析ルールに従って行われる。このラベル付与を構文木の葉から順に行うことで、意味付けされた構文木が得られる。意味付けする際に、提案手法では構文木を部分的に見た時に得られる部分木から得られた SQL クエリをエミュレータで実行 (エミュレーション) し、評価を行う。評価の際に得られた結果や処理の挙動などの情報をもとに意味解析ルールの条件に合致した場合のみ意味ラベルを付与する。ここで用いるエミュレータとはデフォルト状態でインストールされた DB

表 3: 意味解析ルール

No.	Type	子ノード数	条件			付与する意味ラベル
			1	2	3	
1	-	-	×	×	×	固定値
2	function	3	-	×	×	システム関数
3	-	-	-	-	○	システムテーブル
4	-	-	-	○	-	テーブル

表 4: 条件一覧

	内容
条件 1	部分木内に「固定値」以外のラベルが存在
条件 2	実行時にテーブル等が存在しないことを示すエラーが発生
条件 3	実行時に既存のテーブルを参照

サーバである。

より具体的には各 Type ノードに対して、以下の処理を行う。Type ノードを根とする部分木に含まれる Token の文字列を連結し、先頭に “SELECT”，末尾に “;” を追加したクエリをエミュレータで実行し、実行結果を得る。部分木の構造と実行結果をもとに、意味解析ルールに従って部分木の変換とラベル付けを行う。提案手法で定義した意味解析ルールを表 3 に示す。各条件に対して、満たす場合には “○”，満たさない場合には “×”，どちらでも良い場合には “-” と記載している。意味解析ルールは利用者が追加定義することも可能である。意味解析ルールは優先度の高い順に記述しており、優先度の高いルールから順に条件が一致するか判定を行い、一番最初に一致したルールに従って部分木に意味ラベル付けを行う。Type は注目している Type ノードの種類、子ノード数は注目している Type ノードの子となるノードの数、付与する意味ラベルは注目している Type ノードに付与するラベルを示している。

意味解析ルールに含まれる各条件の説明を表 4 に示す。条件 1 は、対象の Type ノードを根とする部分木内に、「固定値」以外、すなわち「システム関数」、「システムテーブル」、「テーブル」ラベルのいずれかが付与されたノードが存在する場合に満たされる。条件 2 は、デフォルトの DB には存在しないテーブルを指定したことによって、テーブルが存在しないというようなエラーが発生した場合に満たされる。条件 3 は、実行時に既存のテーブルを参照している場合に満たされる。エミュレータにはデフォルトで存在するシステムテーブルしか存在しないため、システムテーブルを参照していることを表す。

表 3 の各ルールについて説明する。No.1 のルールは、攻撃者が指定した特定の値を出力するような場合に適用される。例えば、“CHAR(116)” という関数の場合、部分木内に「固定値」以外のラベルが存在せず（条件 1）、実行時にテーブルへのアクセスを伴わず（条件 2）、エラーも発生しない（条件 3）ため、「固定値」ラベルが付与されることとなる。No.2 のルールは、“VERSION()”

や “DATABASE()” 等のシステム情報を取得するような関数の場合に適用される。こういったシステム情報を取得するような関数は引数がないという特徴があるため、Type が “function” で子ノード数が 3（関数名、左括弧、右括弧の 3 ノードのみ）という条件とした。例えば、“VERSION()” という関数は Type が function で、子ノードが “VERSION”，“(”，“)” の 3 つのトークンとなり、実行時にテーブルへのアクセスやエラーが発生しないため、このルールが適用されて「システム関数」ラベルが付与される。No.3 のルールは、DB にデフォルトで存在するシステムテーブルへのアクセスを行う場合に適用される。条件 3 の実行時に既存のテーブルを参照しているという条件を満たす場合、エミュレータにはシステムテーブルしか存在しないことから、システムテーブルを参照しているという判断ができる。例えば、“SELECT table_name FROM information_schema.tables” というクエリの場合、特定の DB システムにデフォルトで存在する “information_schema.tables” というテーブル名を指定して “table_name” というカラムのデータを取得しており、条件 3 を満たすため「システムテーブル」ラベルが付与される No.4 のルールは、システムテーブルではなく、DB の利用者や Web アプリケーションによって作成されたテーブルへのアクセスを行う場合に適用される。条件 2 にあるとおり、実行時にテーブル等が存在しないことを示すエラーが発生した場合、DB にデフォルトで存在するシステムテーブル以外のテーブルを参照しているという判断ができる。例えば、“SELECT name,password FROM users” というクエリの場合、“users” というテーブルはデフォルトでは存在しないため、テーブルが存在しないというエラーが発生する。そのため、条件 2 にあてはまり、「テーブル」ラベルが付与される。

以上の意味解析ルールにもとづき、構文木に対して SQL クエリの意味を示す意味ラベルを付与し、意味ラベル付構文木を得る。

表 5: 被害タイプルール

No.	条件	被害タイプ
1	意味ラベル付構文木内にテーブルラベルが含まれる	DB 内容の漏洩
2	意味ラベル付構文木内にシステム関数ラベルが含まれる	システム情報の漏洩
3	意味ラベル付構文木内にシステムテーブルラベルが含まれる	システム情報の漏洩
4	意味ラベル付構文木内のラベルが固定値ラベルのみ	脆弱性の偵察
5	意味ラベル付構文木内のいずれかの Token の値が “DROP”, “DELETE”, “INSERT”, ... のいずれか	改竄

3.3 被害識別

3.2 節の動的意味解析によって意味ラベル付けが完了した構文木に対して、どの被害の種類に属するものかの識別を行う。ここでいう被害の種類とは 2.1 節で定義した 4 つの被害である。識別には事前に定義した「被害タイプルール」を用いる。

3.2 節で得られた意味ラベル付構文木に対して、被害タイプルールを用いて被害種類の識別を行う。表 5 に提案手法で定義した被害タイプルールを示す。新たな被害が増えた場合に備え、被害タイプルールは利用者が追加可能である。被害タイプルールは条件と識別する被害タイプという 2 つの項目があり、条件はそのルールに当てはまるか否かを示している。条件を満たした場合にその条件に関連する被害タイプとして識別する。被害タイプルールは優先度の高い順に記述しており、優先度の高いルールから順に条件を満たすか判定を行い、一番最初に満たしたルールに従って被害タイプを識別する。例えば、表 5 の No.1 のルールの場合、3.2 節で得られた意味ラベル付構文木の中に、テーブルラベルが付与されたノードが存在した場合、被害タイプは DB 内容の漏洩であると識別する。

4 評価

提案手法の識別精度について評価を行った。評価に用いたデータセットは、検証用の環境で攻撃を再現して作成したデータセットと、実環境のトラフィックから作成したデータセットの 2 種類である。

4.1 人工データセット

識別精度評価を行うための人工データセットを作成した。データセットの作成にはオープンソースの SQL インジェクション脆弱性検査ツールである sqlmap¹を用いた。実験環境に Web サーバを用意し、sqlmap を利用して SQL インジェクションを行った。sqlmap では攻撃対象となる Web サーバに脆弱性が存在しなければ攻撃を止めてしまう機能が存在するため、sqlmap のソースコードに対して攻撃 SQL クエリを生成する箇所を変更し、攻撃対象の Web サーバに脆弱性がなくても sqlmap で用意されている攻撃 SQL クエリの prefix や suffix 等のあら

ゆる組み合わせの攻撃を行うようにした。sqlmap はオプションによって攻撃の被害を変更できるため、異なるオプションを指定することで被害種類が異なる SQL インジェクションを生成した。より具体的には、情報漏洩 (システム情報) を引き起こす攻撃を生成する際はバージョン情報、DBMS のユーザ情報、データベース・テーブル名、サーバのホスト名、テーブルのスキーマ情報をそれぞれ取得するようにオプションを指定した。情報漏洩 (DB 内容) を引き起こす攻撃を生成する際は特定のデータベース・テーブルを指定して内容を取得するようにオプションを指定した。脆弱性の偵察を引き起こす攻撃を生成する際はオプションを指定しなかった。評価に際しては改竄に関する識別精度は求めなかった。この理由は、sqlmap が DB の改竄を行う機能を有していないためである。

DBMS の種類により SQL クエリの構文や用意されている関数が異なる。本評価では広く利用されている MySQL を対象とした。他の DBMS に関しては構文解析の処理における構文とエミュレータを追加することで容易に対応可能であると考えた。上記の作業によって得た攻撃 SQL クエリのうち、攻撃 SQL クエリに含まれる数値のみが異なるものや、重複するものを除いた攻撃 SQL クエリを 48,242 件収集した。

4.2 実環境データセット

実際の攻撃に対する識別精度を評価するために、実環境のトラフィックから SQL インジェクションを収集してデータセットを作成した。トラフィックを収集した期間は 2019 年 3 月 28 日から 2019 年 10 月 10 日である。SQL インジェクションの検知には、OWASP の Core Rule Set²に含まれる、正規表現で記述された SQL インジェクションのルールを用いた。検知した SQL インジェクションから攻撃 SQL クエリを抽出し、重複しない攻撃 SQL クエリを 1,578 件収集した。収集した攻撃 SQL クエリに対して、目視で判断を行い正解ラベルを付与した。

4.3 評価結果

人工データセットに対して提案手法を適用した結果と全体の正解率を表 6 に示す。各被害の種類について正し

¹ <http://sqlmap.org/>

² <https://coreruleset.org/>

表 6: 人工データセットに対する識別結果

識別結果	正解ラベル			合計
	脆弱性の偵察	情報漏洩（システム情報）	情報漏洩（DB 内容）	
脆弱性の偵察	8,877	3,857	8	12,742
情報漏洩（システム情報）	459	23,757	86	24,302
情報漏洩（DB 内容）	159	1,326	7,454	8,939
識別不可	2,009	215	35	2,259
合計	11,504	29,155	7,583	48,242
正解率（全体）				83.1%

表 7: 各被害に対する評価

被害の種類	適合率	再現率	F 値
脆弱性の偵察	69.7%	77.2%	73.2%
情報漏洩（システム情報）	97.8%	81.5%	88.9%
情報漏洩（DB 内容）	83.4%	98.3%	90.2%

く識別された数を下線で示した。識別不可となっているものについては、構文エラーによって識別ができなかったものである。全体の 48,242 件のうち 40,088 件を正しく識別でき、全体の正解率は 83.1%という結果となった、また、全体の 4.7%にあたる 2,259 件については、構文エラーにより識別ができなかった。

被害の種類毎の識別精度の確認のため、表 6 の結果から各被害の種類に対して算出した適合率、再現率、F 値を表 7 に示す。適合率とは、当該被害として識別したデータ数に対する正解データ数の割合である。再現率とは、当該被害として識別されるべきデータ数に対する正解データ数の割合である。表 7 から、情報漏洩（DB 内容）の再現率が 98.3%と高いことが分かる。情報漏洩（DB 内容）は、漏洩した内容によっては顧客の個人情報の漏洩などの重大な被害に繋がる。提案手法では、このような重大な被害の見逃しを減らすことができると考えられる。

次に、実環境データセットに対して提案手法を適用した結果を表 8 に示す。データセットの内訳としては、1,558 件のうち脆弱性の偵察が 1,551 件と、99.6%を占めていた。全体の 1,558 件のうち 1,120 件を正しく識別でき、全体の正解率は 71.9%という結果となった、また、全体の 7.8%にあたる 122 件については、構文エラーにより識別ができなかった。

4.4 考察

表 6, 表 8 に示したように、人工データセットにおいては全体の 4.7%にあたる 2,259 件、実環境データセットにおいては全体の 7.8%にあたる 122 件が構文エラーによって被害が識別できなかった。特に、脆弱性の偵察を行う攻撃について識別不可となった件数が多くなっている。これは、脆弱性の偵察を行う攻撃の中には、今回対象とした MySQL 以外の DBMS を対象とした攻撃が多く含まれていたためである。今回対象としなかった他の

DBMS のエミュレータを追加し、他の DBMS の構文に含まれる単語を表 2 の SQL-WORD に追加することで、被害が識別できるようになると考えられる。実環境データセットにおける正解率が 71.9%と、人工データセットの正解率より低くなっているのも同様の理由であるため、他の DBMS への対応を行うことで全体的に精度を向上させることが可能である。

また、表 7 によると脆弱性の偵察についての適合率が 69.7%と特に低かった。この理由は、システム情報の漏洩となる攻撃を正しく識別できず、脆弱性の偵察として誤って識別してしまっているためである。例えば、“@@HOSTNAME”というシステム変数からホスト名を取得するような攻撃はシステム情報の漏洩となるが、提案手法では“@@HOSTNAME”の値がシステム情報であると識別できない。意味解析ルールにシステム変数を表すラベルを付与するルールを追加し、合わせて被害タイプルールも追加することで正しく被害を識別できると考えられる。

5 関連研究

SQL インジェクションの対策のために、脆弱性の検知や攻撃の検知といった多くの研究が行われている。脆弱性を検知する研究としては、例えば、Web アプリケーションのソースコードを解析することで SQL インジェクションの脆弱性を検出する研究が存在する [4, 5]。

SQL インジェクションを検知する手法としては、WAF で行っているような正規表現のルールセットを用いたものの他にも、機械学習を用いた検知手法 [1, 2] や、DBMS 上で実行される SQL クエリを解析することで SQL インジェクションを検知する研究が行われている [6, 7]。例えば、AMNESIA [6] は、Web アプリケーションのソースコードをもとに作成した正常な SQL クエリのモデルを用いて、DBMS 上で実行される SQL クエリを検査することで SQL インジェクションの検知を行う。他にも、検知率の向上に向けて、WAF による SQL インジェクションの検知を回避する攻撃を自動生成する研究 [8] や、正規表現で記述された WAF のルールセットを自動で修正する研究 [9] も行われている。

SQL インジェクションを検知した後の分析に関わる研

表 8: 実環境データセットに対する識別結果

識別結果	正解ラベル				合計
	脆弱性の偵察	情報漏洩 (システム情報)	漏洩 (DB 内容)	改竄	
脆弱性の偵察	1,119	0	0	0	1,119
情報漏洩 (システム情報)	69	0	0	0	69
情報漏洩 (DB 内容)	246	1	1	0	248
改竄	0	0	0	0	0
識別不可	117	0	0	5	122
合計	1,551	1	1	5	1,558
正解率 (全体)					71.9%

究としては、トラフィックから攻撃の成否判定を行う研究 [10] などが存在する。

上記のいずれの手法でも、SQL インジェクションによって発生する被害を識別することはできないのに対して、情報漏洩の被害に着目した研究として、Gu らは、SQL インジェクションの検知に加えて、漏洩したデータの解析を行う手法である DIAVA [3] を提案している。DIAVA は、正規表現を用いて SQL インジェクションの検知と成否判定を行い、さらにデータが漏洩していた場合には漏洩したデータの抽出と、漏洩したデータがハッシュ化されていた場合には解読の容易さを分析する。DIAVA によって、SQL インジェクションによる情報漏洩を検出することができるが、漏洩したデータが HTTP レスポンスのペイロードに出力される一部の場合しか対応しておらず、Blind SQL injection などでは検出ができない。それに対して、本研究では、レスポンスへの出力に関わらず被害の種類を識別することができる。

6 おわりに

本稿では、SQL インジェクションに対する被害を識別する手法を提案した。提案手法では、攻撃 SQL クエリを構文解析した上で部分的に実行した際の挙動から、どのような被害を引き起こすかを識別する。提案手法では、HTTP リクエストに含まれる SQL クエリの断片のみを入力とするため、攻撃対象の DB で実行される SQL クエリ全体や、その他 DB や Web アプリケーションの情報を必要とせずに、その攻撃が成功した場合の被害を識別することが可能である。評価によって、人工データセットに対して 83.1%、実環境データセットに対して 71.9% の精度で正しく被害を識別できることを示した。

参考文献

- [1] N.M. Sheykhkanloo, “A Learning-based Neural Network Model for the Detection and Classification of SQL Injection Attacks,” International Journal of Cyber Warfare and Terrorism, vol.7, no.2, pp.16–41, 2017.
- [2] D. Kar, S. Panigrahi, and S. Sundararajan, “SQLiGoT: Detecting SQL injection attacks using

graph of tokens and SVM,” Computers Security, vol.60, pp.206–225, 2016.

- [3] H. Gu, J. Zhang, T. Liu, M. Hu, J. Zhou, T. Wei, and M. Chen, “DIAVA: A Traffic-Based Framework for Detection of SQL Injection Attacks and Vulnerability Analysis of Leaked Data,” IEEE Transactions on Reliability, vol.PP, pp.1–15, 2019.
- [4] Y. Xie and A. Aiken, “Static Detection of Security Vulnerabilities in Scripting Languages,” Proceedings of the 14th Conference on USENIX Security Symposium, pp.179–192, 2006.
- [5] V.B. Livshits and M.S. Lam, “Finding Security Vulnerabilities in Java Applications with Static Analysis,” Proceedings of the 14th Conference on USENIX Security Symposium, pp.18–18, 2005.
- [6] W.G.J. Halfond and A. Orso, “AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks,” Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp.174–183, ASE ’05, ACM, 2005.
- [7] G. Buehrer, B.W. Weide, and P.A.G. Sivilotti, “Using Parse Tree Validation to Prevent SQL Injection Attacks,” Proceedings of the 5th International Workshop on Software Engineering and Middleware, pp.106–113, SEM ’05, 2005.
- [8] D. Appelt, C.D. Nguyen, and L. Briand, “Behind an Application Firewall, Are We Safe from SQL Injection Attacks?,” 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp.1–10, 2015.
- [9] D. Appelt, A. Panichella, and L. Briand, “Automatically Repairing Web Application Firewalls Based on Successful SQL Injection Attacks,” 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pp.339–350, 2017.
- [10] 鐘本楊, 青木一史, 三好潤, 嶋田創, 高倉弘喜, “攻撃コードのエミュレーションに基づく Web アプリケーションに対する攻撃の成否判定手法,” 情報処理学会論文誌, vol.60, no.3, pp.945–955, 2019.