Vol.2019-IOT-46 No.8
2019/6/14

Webアプリケーションテストを用いたSQLクエリのホワイトリスト自動作成手法

野村 孔命[1,a)]　力武 健次[1,2]　松本 亮介[3]

概要：Webアプリケーションの脆弱性を利用してデータベースから機密情報を窃取する攻撃が問題になっている．対策として，アプリケーションが発行するクエリを正常なクエリとしてホワイトリストに定義し，リストにないクエリを検知する方法がある．従来のホワイトリスト自動作成手法では，Webサービスのユーザがサービスを利用する過程で発行されるクエリからリストを自動で作成するため，クエリを収集している間は検知が行えない．また，複数の実装の異なるアプリケーションを運用しており，これらに従来手法を適用する場合，それぞれに対して手法の実装が必要となり実装の工数が多い．これらの課題を解決するためには，アプリケーションが稼動する前の段階でアプリケーションの実装に依存せず，ホワイトリストを作成する必要がある．そこで，本稿では，アプリケーション稼動前の動作テスト実行時に発行されるクエリからホワイトリストを作成する手法を提案する．提案手法の評価のため，SQLインジェクションの脆弱性のあるアプリケーションに提案手法を適用し，SQLインジェクション攻撃を検知できることを確認した．しかし，提案手法はテスト時に発行されたクエリを利用するため，正常なクエリを異常とみなす，もしくは異常なクエリを正常とみなす誤検知が発生することが懸念される．この誤検知への対策を検討するために，実環境のアプリケーションに提案手法を適用し，誤検知されたクエリの発行元となったアプリケーションの処理を特定することで，誤検知の原因を調査し考察する．

Automatic Whitelist Generation for SQL Queries Using Web Application Tests

Komei Nomura[1,a)]　Kenji Rikitake[1,2]　Ryosuke Matsumoto[3]

1. はじめに

Webアプリケーションの脆弱性を利用した攻撃は後を絶たない．攻撃によって，Webサービスの個人情報などサービスの機密情報の漏洩が発生している [5,7]．このような攻撃は，Webアプリケーションの脆弱性を利用して開発者の想定と異なる不正クエリをデータベースで実行することで行われる．この際，1つの不正クエリの実行が大規模な情報漏洩を招くことにより，サービスの信頼性の低下が起こり得る．これを防ぐには，不正クエリはデータベースで実行される前に検知する必要がある．

不正クエリの検知には，ネットワーク攻撃検知に用いられる不正検知の手法 [13] が応用される [3]．不正クエリの検知には，ブラックリスト方式とホワイトリスト方式がある [16]．ブラックリスト方式は，既知の不正なパターンを定義して，パターンマッチングを行い，合致したクエリを検知する．不正なパターンには，攻撃に利用される特定のSQLの文字列を定義できるため，ブラックリストは，共通の不正なパターンとして，別のWebアプリケーションでも利用できる．しかし，不正なパターンの定義には，攻撃に関する膨大な知識が必要であり，開発者の知識の不足による定義漏れが発生することがある．また，全ての不正なパターンを定義できたとしても，リストに定義されていない

[1] GMOペパボ株式会社 ペパボ研究所, Pepabo R&D Institute, GMO Pepabo, Inc.
[2] 力武健次技術士事務所, Kenji Rikitake Professional Engineer's Office.
[3] さくらインターネット株式会社 さくらインターネット研究所, SAKURA Research Center, SAKURA Internet Inc.
[a)] komei.nomura@pepabo.com

ⓒ 2019 Information Processing Society of Japan

未知のパターンを持つ不正クエリを検知できない [11]．一方，ホワイトリスト方式は，Web アプリケーションが発行するクエリをホワイトリストに定義し，パターンマッチングを行い，定義にないクエリを検知する．この方法では，未知のパターンを持つ不正クエリが発行されたとしても検知できる．しかし，それぞれの Web アプリケーション毎に発行されるクエリは異なるため，ホワイトリストはそれぞれの Web アプリケーションに対して定義する必要がある．ブラックリストのみの利用では，未知のパターンを持つ不正クエリを検知できないため，ホワイトリストを併用することで未知のパターンにも対応することができる．

ホワイトリストの作成には，Web アプリケーションが発行するクエリを開発者が手動でリストに登録する方法がある [8]．しかし，大規模で複雑な Web アプリケーションでは発行されるクエリ数が膨大となり，開発者が全ての発行されるクエリを把握するのは困難である．さらに，Web アプリケーションが更新されることによって，発行されるクエリは変化する．この発行クエリの変化に対応するため，開発者はホワイトリストを更新しなければならない．これらのことから，この方法は開発者への負担が大きい．開発者への負担を軽減するために，Web アプリケーションが発行するクエリのホワイトリストを自動で作成する手法が提案されている [4,18]．しかし，これらの手法は，ユーザの入力によって生じるクエリをホワイトリストの作成に利用するため，(1)Web アプリケーションが稼働した後即時に不正クエリを検知できないことや，Web アプリケーションの実装に依存するため，(2)Web アプリケーションごとに対策の実装が必要となることが課題となる．

不正クエリ対策の実施を行う際，サービス運営者は，サービスの開発や運用に支障をきたすことを最小限にしたい．また，Web サービスの開発には，PHP や Ruby のような様々な言語や Web アプリケーションフレームワークが利用されるため，Web アプリケーションの実装に依存した対策は，適用する Web アプリケーションそれぞれに対して実装が必要となり，工数が多く作業効率が低い．そのため，不正クエリ対策は，Web アプリケーションの実装に依存しない方法である必要がある．また，不正クエリの検知漏れや，Web アプリケーションが発行するクエリの誤検知を軽減するために，不正クエリ対策には，Web アプリケーション稼働後即時に不正クエリを検知でき，かつ稼動直後からの高い検知精度が求められる．

本研究では，不正クエリ対策の実施による開発者への負担を軽減するために，開発者が Web アプリケーションの変更に追従してテストコードを整備する開発プロセスにおいて，テスト時に発行されたクエリを用いたホワイトリスト自動作成手法を提案する．提案手法では，Web アプリケーションが稼働する前のテストの段階にホワイトリスト作成を組み込むことで，Web アプリケーション稼働後，即座に不正クエリを検知可能な状態にできる．ホワイトリストの作成に必要なクエリは，データベースの前段にデータベースプロキシを配置して収集する．これらの収集したクエリのリテラルをプレースホルダーに置き換えたクエリ構造をホワイトリストに登録する．このように，データベースプロキシでクエリを収集することで，Web アプリケーションの実装に依存しないホワイトリスト作成を実現する．また，提案手法では，テスト時のクエリを用いてホワイトリストを作成するため，検知されるクエリは，ホワイトリストに載っていないテスト時に発行されなかったクエリであり，これには不正クエリも含まれる．

本稿の構成を述べる．2 章では，Web アプリケーションが発行するクエリのホワイトリスト作成の課題を整理する．3 章では，ホワイトリスト作成を組み込んだ開発プロセスと提案手法の検知特性について述べた後，提案手法の設計について述べる．4 章で，実験により提案手法の検知精度を評価し，5 章で，実環境のクエリログを使った実験により実運用上の課題と考察を述べ，6 章でまとめる．

2. ホワイトリスト作成の課題

不正クエリを検知するために，Web アプリケーションが発行するクエリのホワイトリストを開発者が手動で作成して検知する方法がある [8]．Web アプリケーションが発行するクエリのリテラル値は，ユーザ入力によって変化することがあるので，ホワイトリストには，クエリのリテラル値をプレースホルダーに置き換えたクエリ構造が登録される [1]．そのため，開発者は Web アプリケーションのソースコードから全てのクエリを発行する処理を特定し，発行されるクエリのクエリ構造をホワイトリストに登録する．

開発者による手動でのホワイトリストの作成は開発者への負担が大きい方法となっている．Web アプリケーションは開発が進むに伴って大規模化・複雑化するため，Web アプリケーションが発行するクエリ数は増加する．そのため，開発者が把握しなければならないクエリ数が膨大となり，ホワイトリストの作成が困難になる．また，開発者は，ホワイトリストの検知精度を保つために，Web アプリケーションが発行するクエリの変化に追従してホワイトリストを更新しなければならない．しかし，Web アプリケーションの改修頻度は高く [6]，発行クエリは頻繁に変化するため，開発者がホワイトリストを更新しなければならない頻度も高い．さらに，Web アプリケーションの実装にはオブジェクトリレーショナルマッピング（ORM）[15] が利用される場合があり，開発者は Web アプリケーションが発行するクエリを意識することが少なくなっている．ORM はオブジェクト指向言語におけるオブジェクトとデータベースのレコードを関連づける機能を提供している．これにより，開発者はデータベースのレコードをオブジェクトとして扱えるため，直接 SQL 文を記述することが少なくな

表 1　Ruby のコードによって発行される SQL クエリの例

Ruby code	User.find(1)
SQL query	SELECT * FROM users WHERE (users.id = 1) LIMIT 1

る．例えば，Web アプリケーションの実装に Web アプリケーションフレームワークである Ruby on Rails [9] を用いた場合，ORM として ActiveRecord [10] が利用される．ActiveRecord によってデータベースのレコードと Ruby のクラスオブジェクトが関連づけられ，開発者は Ruby のコードで SQL クエリの発行処理を記述できる．Ruby のコードによって発行される SQL クエリの例を表 1 に示す．表 1 は，クラスオブジェクト User がデータベースの users テーブルのレコードに関連づけられている場合の，Ruby のコードと SQL クエリの対応関係を示している．このように，ORM を用いた場合，開発者はオブジェクトを用いてデータベースからデータを取り出せるので，Web アプリケーションが発行するクエリを把握しづらくなる．しかし，手動でのホワイトリストを作成は，ORM の処理を理解し発行されるクエリを把握しなければならず，作業量は増えるため，ORM を用いた Web アプリケーション開発に適していない．これらのことから，手動でのホワイトリスト作成は開発者への負担が大きくなるので，負担を軽減するための方法が必要となる．

2.1　発行クエリを用いたホワイトリスト自動作成

Web アプリケーションの稼動時に発行されたクエリを収集し，クエリの構文解析を行いクエリ構造に変換することで，ホワイトリストを作成する手法が提案されている [2, 4]．この手法を用いることで，Web アプリケーションの稼動時に自動でホワイトリストを作成することができる．しかし，この手法には，クエリを収集しホワイトリストを作る学習フェーズと，作成したホワイトリストを用いて検知を行う検知フェーズがあり，学習フェーズ中は不正クエリの検知を行うことができない．そのため，Web アプリケーションが稼動した後，即座に不正クエリを検知することができない．更新頻度が高い Web アプリケーションにおいては，頻繁にホワイトリストの再学習を行わなければならず，その都度検知を行えない期間が発生してしまう．Web アプリケーション稼働中にホワイトリストを作成する方法は，不正クエリを検知できない期間を生じさせてしまうため，ホワイトリストの作成は，稼働前に行う必要がある．また，この手法では，Web アプリケーション稼働前に，過去に収集しておいたクエリログからホワイトリストを作成できる．しかし，クエリログによるホワイトリスト作成は，Web アプリケーションが改修されたときの発行クエリの変化に対応できない．これは，Web アプリケーションの改修前には発行されていたが，改修後に発行されなくなったクエリや，改修後に新たに発行されるようになったクエリが，クエリログに含まれるからである．

2.2　静的解析を用いたホワイトリスト自動作成

Web アプリケーションのソースコード内のクエリを発行する処理を解析することで，ホワイトリストを自動で作成する手法が提案されている [18]．この手法は Web アプリケーションが稼動する前にホワイトリストを作成できるため，Web アプリケーションが稼動した後，即時に不正クエリを検知できる．一方，この手法では，ソースコードの解析処理が，実装に利用されている言語や Web アプリケーションフレームワークなどに依存するため，適用する Web アプリケーションそれぞれに対して実装が必要となる．Web サービスの開発には，PHP や Ruby などの様々な実装言語や Web アプリケーションフレームワークが利用されるため，実装方法が異なる複数の Web アプリケーションが開発される場合がある．このような状況において，Web アプリケーションの実装に依存する方法は，適用するそれぞれの Web アプリケーションに対して実装が必要となるため，適していない．このことから，実装が異なる複数の Web アプリケーションに対してホワイトリストを作成する場合は，Web アプリケーションの実装に依存しない方法をとるべきである．

3.　提案手法

本研究では，ホワイトリスト作成が開発者に与える負荷を軽減するために，Web アプリケーションテストを用いたホワイトリストの自動作成手法を提案する．提案手法では，Web アプリケーションの動作テストをテストコードとして管理しており，開発者が Web アプリケーションの変更に応じてテストコードを整備する開発プロセスにおいて，テスト実行時のクエリを収集し，収集されたクエリからホワイトリストを作成する．提案手法は，開発プロセス内のテストの段階でホワイトリストを作成するため，開発プロセスへの影響を抑えつつ，Web アプリケーションが稼働した後，即時に不正クエリを検知できる．また，データベースの前段にデータベースプロキシを配置し，クエリの収集を行うことで，Web アプリケーションの実装に依存しないホワイトリストの作成を実現する．

3.1　ホワイトリスト作成を組み込んだ開発プロセス

提案手法は，自動テストを用いた開発プロセスのテストの段階にホワイトリストを作成する処理を組み込む．開発プロセスにホワイトリストの作成を組み込む方法を示すために，自動テストを用いた開発プロセスとホワイトリスト作成を組み込んだ開発プロセスを図 1 を用いて説明する．

図 1 の自動テストを用いた開発プロセスについて説明する．(1) で，開発者は，Web アプリケーションの新機能の

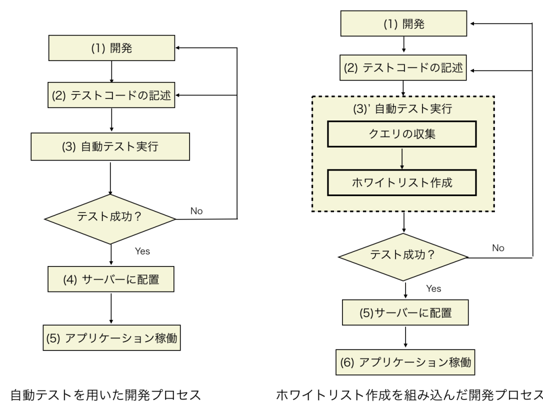

図 1　自動テストを用いた開発プロセスとホワイトリスト作成の組み込み

開発や既存機能の修正を行う．(2) で，開発者は，開発もしくは修正した機能に対して，Web アプリケーションをテストするときの動作手順であるテストケースと，期待される動作結果をテストコードに記述する．(3) の自動テストの段階で，開発者は，テストコードを用いて全てのテストを実行し，Web アプリケーションが仕様通りに動作しているかを確認する．このとき，テストが失敗した場合は，開発した機能の動作が仕様通りでない，もしくはテストコードの記述，すなわち仕様の定義に誤りがあることを意味する．この場合，開発者は，テストが失敗した原因を特定し，Web アプリケーションのソースコードもしくはテストコードを修正する．テストが成功した場合，開発者は，開発した機能が仕様通りに動作しているとみなし，(4) 新しい Web アプリケーションのソースコードがサーバに配置し，(5) 運用を開始する．

次に，図 1 のホワイトリストの作成を組み込んだ開発プロセスについて説明する．図 1 の (3)' のように，自動テストを実行する工程にホワイトリストを作成する処理を追加する．ホワイトリストは，自動テスト実行中に Web アプリケーションから発行されたクエリを用いて作成する．テスト成功後に，新しい Web アプリケーションのソースコードと，それに対応したホワイトリストをサーバに配置する．

Web アプリケーションが発行するクエリは，機能追加や修正によって変化するため，それに対応してホワイトリストを作成する必要がある．自動テストを用いた開発プロセスにおいて，テストコードは Web アプリケーションの変更に応じて整備されるので，テスト時に発行されるクエリも，それに応じて変化する．提案手法では，このクエリをホワイトリスト作成に利用しているので，Web アプリケーションが稼働中に発行するクエリとホワイトリストの整合性を保つことができる．また，テストの段階でホワイトリストを完全に作成できるので，新しい Web アプリケーションが稼働したときには，不正クエリを即座に検知できる状態にすることができる．この特性は，Web アプリケーションが稼動した後でホワイトリストの作成を始める方法では

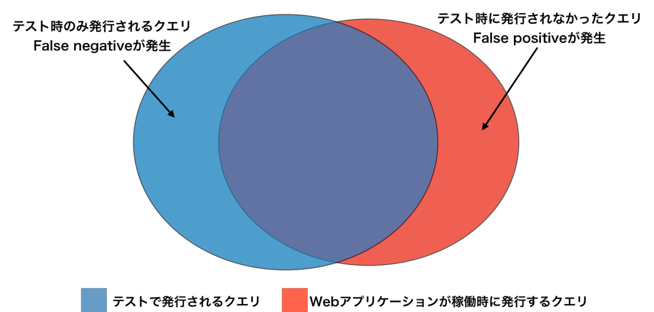

図 2　Web アプリケーションがテスト時と稼働時に発行するクエリの集合の関係

実現できない．さらに，開発者は既存の開発プロセスを変更せずにホワイトリストを作成できるため，提案手法を導入した時の既存の開発プロセスへの影響は小さい．

開発者がテストコードを記述することと，ホワイトリストを手動で作成することの負担の差について述べる．テストコードには，想定される Web アプリケーションの動作であるテストケースを記述するのに対して，ホワイトリストには，Web アプリケーションが動作の過程で発行するクエリ構造を登録する．テストコードを記述する場合，テストケースを導出するために，開発者は Web アプリケーションの動作を理解しなければならない．一方，ホワイトリストを作成する場合，クエリ構造を導出するために，開発者は Web アプリケーションの動作を理解した上で，その動作の過程で行われるクエリ発行の動作を理解しなければならない．また，Web アプリケーションの開発に ORM が利用されている場合，ソースコードから発行されるクエリを導出するためには，ORM の動作の理解も必要になる．そのため，この場合は，開発者がクエリを発行する動作を詳細に理解している可能性は低い．これらのことから，テストコードを記述することは，ホワイトリストを手動で作成することに比べ，要求される Web アプリケーションの動作に関する知識が少ないため，開発者への負担が小さいと考えられる．

3.2　提案手法の検知特性

ホワイトリストによる検知では，Web アプリケーションが稼働時に発行するクエリを正常なクエリとして定義するが，提案手法はテスト時に発行されたクエリからホワイトリストを作成する．そのため，Web アプリケーションがテスト時と稼働時に発行するクエリの集合の関係が提案手法の検知特性に影響するので，この関係を図 2 に示す．

図 2 を用いて提案手法の検知特性について説明する．テスト時の Web アプリケーションの実行パターンであるテストケースは，Web アプリケーション稼働時の実行パターンの一部でしかないため，テスト時に発行されるクエリは，Web アプリケーションが発行するクエリの一部しか含んでいない．そのため，開発者が想定するユーザ入力によっ

表 2　提案手法で検知できないクエリの例

正常なクエリ	SELECT * FROM users LIMIT 30
異常なクエリ	SELECT * FROM users LIMIT 1000

てWebアプリケーションが発行する正常なクエリを不正と判断する，False positiveが発生する．提案手法では，テストケースを追加することによって，False positiveを低減することができるが，テストケース増加による管理の複雑化が懸念されるため，テスト時に発行されなかったクエリを補う方法が必要となる．また，テスト時に発行されるクエリには，Webアプリケーション稼働時に発行されないがテスト時のみ発行されるクエリがある．テスト時のみ発行されるクエリの例として，テストデータの登録や削除を行うクエリが挙げられる．提案手法では，テスト時のみ発行されるクエリが作成されるホワイトリストに含まれるため，Webアプリケーションが発行しない不正なクエリを不正と判断できない，False negativeが発生する．

ホワイトリストには，クエリのリテラル部分をプレースホルダーに置き換えたクエリ構造を登録する．例えば,「SELECT * FROM users LIMIT 10」は，リテラルである「10」をプレースホルダー「?」に置き換え,「SELECT * FROM users LIMIT ?」としてホワイトリストに登録する．提案手法は，ホワイトリストにないクエリ構造を持つクエリを検知する．そのため，SQLインジェクション攻撃 [17] のように，SQL文を含む文字列をリテラルにインジェクションし，クエリ構造を変化させる攻撃は，提案手法によって検知できる．しかし，提案手法では，数値リテラルの閾値の設定や，文字列リテラルの正規表現による不正なパターンの定義を行っていないため，クエリ構造が同一でリテラル値が不正であるようなクエリを検知することができない．提案手法が検知できないクエリの違いの例を表2に示す．

表2に示したクエリは，クエリ構造は同一であるが，LIMIT句で指定されている数値リテラルが異なる．このような場合に対応するために，リテラル値の異常を検知する方法を検討する必要がある．

3.3　提案手法の設計

提案手法では，データベースの前段にデータベースプロキシを配置し，テスト時に発行されたクエリの収集とWebアプリケーション稼働時の不正クエリの検知を行う．テスト時のホワイトリスト作成フローとWebアプリケーション稼動時の検知フローを図3を用いて説明する．

テスト時のホワイトリスト作成フローを説明する．図3において，まず，テストが実行され，Webアプリケーションからデータベースプロキシにクエリが発行される．データベースプロキシは受け取ったクエリをそのままデータベースに渡し，この間に中継したクエリを記録する．全てのテストが終了した後，データベースプロキシは，記録されたクエリのリテラル値をプレースホルダーに置き換え，クエリ構造に変換し，ホワイトリストに登録する．このように，データベースプロキシを用いてクエリの収集を行うことで，Webアプリケーションの実装に依存せず，ホワイトリストを作成できる．

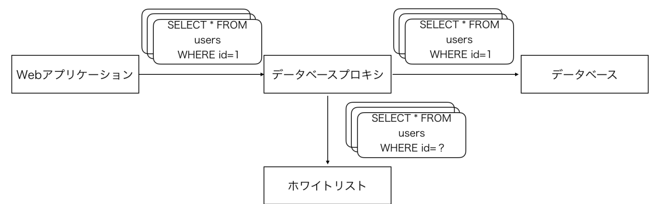

図 3　提案手法の設計

Webアプリケーション稼動時の検知フローについて説明する．図3において，まず，Webアプリケーションがユーザからの入力を受けクエリを発行する．次に，Webアプリケーションから発行されたクエリをデータベースプロキシが受け取り，受け取ったクエリをクエリ構造に変換し，ホワイトリストと照合する．照合した結果，ホワイトリストに登録されていた場合は，クエリをデータベースに渡し実行する．ホワイトリストに登録されていなかった場合は，開発者に検知したクエリを通知する．

不正クエリの検知には，発行されたクエリとホワイトリストを照合する処理がある．ホワイトリストの照合処理が低速だった場合，データベースプロキシで発行されたクエリが停滞し，ユーザへのレスポンス時間が増加することが懸念される．そのため，ホワイトリストの照合処理は高速である必要があり，照合処理の計算量が重要となる．Webアプリケーションから発行されたクエリに対してホワイトリストを全探索した場合，ホワイトリストの照合処理の計算量は，ホワイトリストのエントリ数nに対して$O(n)$となる．これを避けるために，クエリ構造をキーとしたハッシュテーブルとしてホワイトリストを作成し，ハッシュテーブルを用いて照合することで，探索の計算量を$O(1)$に抑えることができる．このことから，提案手法は，ホワイトリストのエントリ数に依存せず一定の計算量で，高速に照合処理を行えると考える．

4.　実験

提案手法の有効性を確認するために，提案手法によって作成したホワイトリストの検知精度の評価を行った．ホワイトリストの検知精度の指標として，False positiveとFalse negativeを利用する．False positiveは，開発者が想定するユーザ入力によってWebアプリケーションが発行する正常なクエリを不正と判断することとし，False negativeは，Webアプリケーションが発行し得ない不正なクエリを不正と判断できなかったこととする．

ⓒ 2019 Information Processing Society of Japan

表 3 実装メソッドの一覧

HTTP	URL	操作
GET	articles	全ての article を表示
GET	articles?title=""	title で検索して article を表示
POST	articles	article を作成
GET	articles/:id	id の article を表示
PATCH	articles/:id	article を更新
DELETE	articles/:id	article を削除

4.1 ホワイトリストの検知精度

提案手法によって作成したホワイトリストの検知精度を評価するために，実験環境を構築し，False positive と False negative を測定する実験を行った．

実験環境では，データベースプロキシとして ProxySQL [12] を採用した．ProxySQL には，受け取ったクエリを収集しクエリ構造に変換して保存する機能があり，この機能をホワイトリストの作成に利用した．また，データベースには ProxySQL がサポートしている MySQL を採用し，article テーブルに title カラムと content カラムを作成した．Web アプリケーションには，Ruby on Rails を用いて，article テーブルの操作を行うメソッドを実装し，全てのメソッドに対してテストを記述した．実装したメソッドを表 3 に示す．表 3 に示すメソッドの内，title で検索して article を表示するメソッドには，SQL インジェクションの脆弱性を持たせた．

ホワイトリストに登録するクエリ構造には，Web アプリケーションのテストを実行した後に，ProxySQL に記録されたクエリ構造を利用し，エントリ数が 23 個のホワイトリストを作成した．正常なクエリデータとして，ブラウザから手動で，表 3 に示すメソッドを持つ Web アプリケーションに HTTP リクエストを行い，その間に発行された 17 個のクエリを取得した．この正常なクエリデータとホワイトリストを照合して，誤検知されたクエリをカウントし，False positive を算出した．不正なクエリデータとして，Web アプリケーションの article を title で検索するメソッドに，sqlmap [14] を用いて SQL インジェクション攻撃を含んだ HTTP リクエストを行い，その間に発行された 265 個のクエリを取得した．この不正なクエリデータとホワイトリストを照合して，検知できなかったクエリをカウントし，False negative を算出した．

実験結果は False positive が 17.65%，False negative が 0%となった．実験結果より，False positive は 17.65%発生したため，正常なクエリの一部を誤検知することがわかった．また，False negative は発生しておらず，SQL インジェクション攻撃による不正クエリを全て検知できることがわかった．

False positive について考察する．表 3 に示す全てのメソッドに対してテストは記述されており，テストカバレッジの算出にも含まれていたが，False positive が発生した．

- UPDATE ‘articles‘ SET ‘content‘ = ?, ‘updated_at‘ = ? WHERE ‘articles‘.‘id‘ = ?
- UPDATE ‘articles‘ SET ‘title‘ = ?, ‘updated_at‘ = ? WHERE ‘articles‘.‘id‘ = ?
- UPDATE ‘articles‘ SET ‘title‘ = ?, ‘content‘ = ?, ‘updated_at‘ = ? WHERE ‘articles‘.‘id‘ = ?

図 4 False positive として検知されたクエリ

False positive として誤検知されたクエリは，17 個の正常なクエリデータの 17.65%である 3 個のクエリであった．誤検知されたクエリを図 4 に示す．

図 4 に示すようなクエリがテスト時に発行されなかった原因は，データベース上にある更新前の article データと，UPDATE のクエリを実行した後に生じる更新後の article データが同一になっていたことである．このように，更新前と更新後でデータが同一となる時，Ruby on Rails で実装した Web アプリケーションは UPDATE のクエリを発行しない．しかし，UPDATE のクエリを発行するメソッドは，テスト時に実行されるため，テストカバレッジの算出に含まれる．このことから，テストカバレッジが 100%であっても，テストケースによっては発行されないクエリがあり，False positive が発生することがわかった．False positive を低減するために，テストケースの欠如によって，テスト時に発行されなかったクエリをホワイトリストに補う方法が必要となる．

False negative について考察する．実験では，False negative は発生しなかったが，3.2 節で述べたように，ホワイトリストにはテスト時のみ発行されるクエリが含まれており，False negative を発生させる可能性がある．そのため，ホワイトリストに含まれるテスト時のみ発行されるクエリの割合とそのクエリ構造を調査した．調査の結果，ホワイトリストには，テスト時のみ発行されるクエリが 39.13%含まれていた．これらのクエリ構造の一覧を図 5 に示す．

提案手法では，図 5 に示すようなクエリ構造を持つクエリを検知できないことがわかった．SQL インジェクション攻撃では，Web アプリケーションが発行するクエリに任意の SQL 文字列をインジェクションし，クエリの実行結果を変更する．そのため，SQL インジェクション攻撃によって発行されるクエリは，Web アプリケーションが発行するクエリに攻撃に利用される特定の SQL 文字列を加えたものである．このようなクエリが上記したクエリ構造と一致する可能性は否定できないが，実験では，False negative が発生しなかったことから，このような場合が発生する可能性は低いと考えられる．

5. 実環境での評価

4.1 節の実験により，提案手法では False positive と False

- ROLLBACK
- SELECT COUNT(*) FROM ‘articles‘
- RELEASE SAVEPOINT active_record_1
- SAVEPOINT active_record_1
- SET FOREIGN_KEY_CHECKS = ?
- SELECT ‘articles‘.* FROM ‘articles‘ ORDER BY ‘articles‘.‘id‘ DESC LIMIT ?
- DELETE FROM ‘articles‘; INSERT INTO ‘articles‘ (‘id‘, ‘title‘, ‘content‘, ‘created_at‘, ‘updated_at‘) VALUES (?, ?, ?, ?, ?), (?, ?, ?, ?, ?), (?, ?, ?, ?, ?);
- SELECT @@FOREIGN_KEY_CHECKS
- SELECT @@max_allowed_packet

図 5 テスト時のみ発行されるクエリ

negative の発生が懸念されることが確認された．実環境において，これらの誤検知の発生頻度を確認し，実運用上での課題を考察するために，筆者が所属する GMO ペパボ株式会社が運営する Web サービスのクエリログを用いた実験を行なった．実験では，Web アプリケーションが実環境で稼働している間に発行するクエリと，テスト時に発行するクエリの集合の関係を調査した．

提案手法は Web アプリケーションの更新に追従してホワイトリストの更新を行うので，実験では，Web アプリケーションが発行するクエリとホワイトリストの整合性を取る必要がある．実験対象の Web アプリケーションは 1 営業日に複数回の更新が行われ発行クエリに変化が生じるため，休日で Web アプリケーションの更新が行われていない 2019 年 1 月 12 日から 2019 年 1 月 14 日の 3 日分のクエリログを取得した．クエリログには 172086 個のクエリが含まれ，クエリ数は膨大であるため，実験では 66 個のクエリ構造に変換して利用した．また，テスト時に発行されるクエリの収集には，実環境で稼働している Web アプリケーションとの対応を取るために，クエリログの取得開始日の前日である 2019 年 1 月 11 日時点の Web アプリケーションを利用した．テスト時に発行されるクエリ数は 11514 個であり，これを 99 個のクエリ構造数に変換して実験に利用した．実環境で発行されたクエリ構造とテスト時に発行されたクエリ構造を比較し，図 6 に示すクエリ構造の集合の関係を求めた．

図 6 より，実環境で発行されたクエリ構造とテスト時に発行されたクエリ構造の総数は 119 個であった．この内，46 個のクエリ構造が実環境とテストの両方で発行されたクエリとして観測され，提案手法はこれらのクエリを正常なクエリと判断する．

図 6 において，実環境でのみ発行された 20 個のクエリ構造について考察する．提案手法はこれらのクエリ構造に一致するクエリを不正クエリとして検知する．しかし，これ

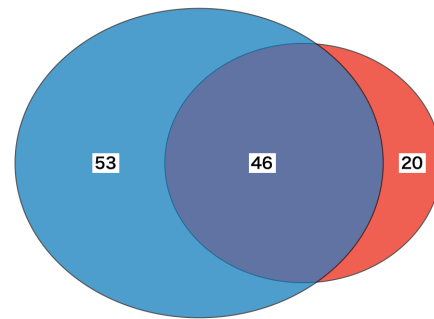

図 6 実環境ので発行されたクエリ構造とテスト時に発行されたクエリ構造の関係

らのクエリは，テーブルのデータを全削除するなどのサービスに損害を与えるような操作ではなかったため，False positive として誤検知された可能性がある．そのため，これらのクエリの発行元となった Web アプリケーションの処理を特定することで，検知されたクエリが正常なクエリであるかの調査を行った．調査の結果，全てのクエリの発行元の Web アプリケーションの処理を特定できたため，これらのクエリは全て False positive として誤検知されたクエリである．また，False positive として誤検知されたクエリがテスト時に発行されなかった理由は，テストケースが実際の利用パターンを網羅できていないことや，クエリの発行処理の実行がテスト時のみ省略されていたことである．また，実験に利用したクエリログは休日に発行されたクエリを記録したものであったため，サービスの運用者が手動で実行するクエリは含まれなかった．

本実験から，提案手法ではテストケースの不足などにより False positive が発生することが確認された．False positive への対処として，ホワイトリストによる検知をテーブル単位で限定的に適用することが挙げられる．本研究の目的は，データベース上の機密情報の窃取や改竄を防ぐことなので，全てのテーブルに対してホワイトリストによる検知を適用する必要はないと考える．そのため，機密情報が保管されたテーブルに対してはホワイトリストによる検知を適用し，その他のテーブルには適用しないなど，テーブル単位でホワイトリストの利用の有無を選択することで，検知対象となるクエリを削減できる．その結果，False positive となるクエリの総数を削減できると考える．しかし，この方法だけでは，False positive を完全に排除することはできないので，ホワイトリストにテストケースの不足によって登録できなかったクエリを補う方法も別途必要と考える．

図 6 において，テストでは発行されたが実環境では発行されなかった 53 個のクエリ構造について考察する．提案手法において，これらのクエリ構造はホワイトリストに登録されているが検知に利用されない．これらのクエリには

2 種類のクエリが含まれていた．1 種類目は，実験に利用したクエリログの期間に利用されていない Web アプリケーションの機能があったことによって，実環境では発行されなかった正常なクエリである．2 種類目は，テスト時のみ発行され，Web アプリケーション稼働時には発行され得ないクエリである．このようなクエリは False negative を引き起こす可能性があり，これらのクエリには，テーブルのデータを全て削除するなどのサービスに損害を与えるものが含まれていた．

本実験から，提案手法ではテスト時のみ発行されるクエリによる False negative が確認された．この問題に対しては，ホワイトリストのみではなく，ブラックリストを併用し多層的に対処する必要があると考える．ブラックリストには，機密情報が保管されているテーブルの全件取得や全件削除など，予め想定される不正クエリを定義する．このブラックリストと提案手法で作成したホワイトリストを併用することで，ホワイトリストで検知できない不正クエリをブラックリストによって検知できると考える．また，ブラックリストが検知できなかった不正クエリをホワイトリストで検知できる場合も考えられる．そのため，ブラックリストとホワイトリストを併用し多層的に対処することで，相補的に不正クエリの検知漏れを低減できると考える．

6. まとめ

本稿では，ホワイトリスト作成が与える開発者への負荷を軽減するために，開発プロセスのテストに着目し，テスト時に発行されるクエリを用いてホワイトリストを自動で作成する手法を提案した．提案手法は，開発プロセスにホワイトリストの作成を組み込むことで，開発者への負荷を軽減しながら，Web アプリケーションが稼働した直後からの不正クエリの検知を実現する．また，提案手法は，データベースプロキシでホワイトリスト作成に必要なクエリを収集するため，Web アプリケーションの実装に依存しない．このことから，実装が異なる複数の Web アプリケーションそれぞれに対して，ホワイトリストを作成する場合に適している．

実環境のクエリログを使った実験から，提案手法では，テストケースの不足により False positive が発生することや，テスト時のみ発行されるクエリによる False negative の発生が確認された．False positive に関しては，ホワイトリストによる検知を適用するテーブルを機密情報が保管されたテーブルに限定することで，検知対象となるクエリを削減し，結果として誤検知されるクエリ数を削減できると考察した．また，False negative に関しては，ブラックリストに予め想定される不正クエリを定義し，ブラックリストとホワイトリストを用いて多層的に検知を行うことで，不正クエリの検知漏れを削減できると考察した．

今後の課題としては，False positive の根本解決のために，ホワイトリストに不足しているクエリを補う方法の検討が挙げられる．また，今後は，ホワイトリストによる検知を機密情報が保管されたテーブルに限定した場合の検知精度の評価や，ブラックリストとホワイトリストを併用した場合の検知精度の評価を行う．

参考文献

[1] D. Kar and S. Panigrahi, "Prevention of SQL Injection attack using query transformation and hashing", *Advance Computing Conference (IACC), 2013 IEEE Third International*, pp.1317-1323, May, 2013.
[2] F. José, V. Marco and M. Henrique, "Detecting malicious SQL", *International Conference on Trust, Privacy and Security in Digital Business*, Vol.4657, pp.259-268, 2007.
[3] F. S. Rietta, "Application layer intrusion detection for SQL injection", *Proceedings of the 44th Annual Southeast Regional Conference*, pp.531-536, Mar, 2006.
[4] F. Valeur, D. Mutz and G. Vigna, "A learning-based approach to the detection of SQL attacks", *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp.123-140, 2005.
[5] IT Security Center, Information-technology promotion agency (IPA), *How to Secure your Website Fifth edition*, Apr, 2011.
[6] J. Mehdi, "Some Trends in Web Application Development", *Future of Software Engineering, 2007, FOSE '07*, pp.199-213, Jun, 2007.
[7] J. Rahul and S. Pankaj, "A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection" *2012 International Conference on Communication Systems and Network Technologies*, pp.453-458, May, 2012.
[8] K. Kemalis and T. Tzouramanis, "SQL-IDS: a specification-based approach for SQL-injection detection", *Proceedings of the 2008 ACM symposium on Applied computing*, pp.2153-2158, Mar, 2008.
[9] M. Bächle and P. Kirchberg, "Ruby on rails", *IEEE Software*, pp.105-108, Nov, 2007.
[10] M. Fowler, *Patterns of enterprise application architecture*, pp.147-150, 2002.
[11] M. Ofer, and S. Amichai, "SQL injection signatures evasion", *Imperva, Inc. White paper*, Apr, 2004.
[12] ProxySQL, http://www.proxysql.com/.
[13] S. Axelsson, "Intrusion detection systems: A survey and taxonomy", *Technical Report 99-15*, Mar, 2000.
[14] sqlmap, http://sqlmap.org/.
[15] S. W. Ambler, "Mapping objects to relational databases", *An AmbySoft Inc. White Paper*, 2000.
[16] V. Luong, "Intrusion detection and prevention system: SQL-injection attacks", *Master's Projects. 16*, 2010.
[17] W. G. Halfond and A. Orso, "A Classification of SQL Injection Attacks and Countermeasures", *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Vol.1, pp.13-15, 2006.
[18] W. G. 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, Nov, 2005.

ⓒ 2019 Information Processing Society of Japan