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)]　阿部 博[3,5]　菅野 哲[3,5]　力武 健次[1,2]　松本 亮介[1,4]

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，クエリのホワイトリストによる検知があるが，大規模なアプリケーションにおいてはクエリパターンが膨大であり，開発者によるリストの手動作成が困難なため，リストの自動作成手法が利用される．しかし，従来のホワイトリスト自動作成手法には，ユーザが Web サービスを利用することによって，発行されるクエリを収集するのに時間がかかり検知の即時性が低い課題や，アプリケーション毎に異なる実装を必要とし汎用性が低い課題がある．本稿では，アプリケーションの動作テスト実行時の発行クエリからホワイトリストを作成する手法を提案する．提案手法はテスト時の発行クエリを使いアプリケーションの実装に依存せず作成できるため，検知の即時性や汎用性は向上する．

Automatic Whitelist Generation for SQL Queries Using Web Application Tests

Komei Nomura[1,a)]　Hiroshi Abe[3,5]　Satoru Kanno[3,5]　Kenji Rikitake[1,2]　Ryosuke Matsumoto[1,4]

Abstract:
Stealing confidential information of the database has became a serious vulnerability issue of Web applications. Defining the whitelist of SQL queries issued by the Web application is a countermeasure to detect the attack. For large-scale Web applications, automated generation of the whitelist is conducted since manually defining a large number of the query patterns is impractical for the developers. Conventional methods for the automated generation are unable to detect the attacks immediately due to the long requiring time of collecting legitmate queries and require the application-specific implementations which reduce the versatility of the methods. In this paper, we propose a method to automatically generate a whitelist using issued queries during Web application tests. Our proposed method uses the generated queries during the application tests and does not depend on the specific application, which results in improved timeliness against the attacks and versatility for the multiple applications.

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ず [12]，Web サービスが保有する個人情報やサービス特有の機密情報を窃取するセキュリティインシデントが発生している [13]．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によって行われる．また，1つの不正クエリの実行が大規模な情報漏洩につながることもあり，サービスの信頼性低下を招いてしまう．そのため，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

[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.
[3] ココン株式会社, Cocon, Inc.
[4] さくらインターネット株式会社, SAKURA Internet, Inc.
[5] 株式会社レピダム, Lepidum Co. Ltd.
[a)] komei.nomura@pepabo.com

れる不正検知 [15] が応用可能である [8]．不正検知には，ブラックリスト方式とホワイトリスト方式がある．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．ブラックリストは，別の Web アプリケーションでも利用できる．しかし，パターンの定義に必要な事前知識が膨大であり，全ての既知の不正クエリの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想定外の不正クエリを検知できない．一方で，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．この方法は，想定外の不正クエリが発行されたとしても検知できる．しかし，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に対して定義する必要がある．本研究では，機密情報の保護を目的としており，不正クエリの検知漏れを許容できないため，ホワイトリスト方式に着目した．

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する方法がある [6]．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，作成したホワイトリストは，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が変化するため，その都度更新しなければならない．そのため，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 [11][4]．しかし，これらの手法には，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，Web アプリケーションが稼働した後即時に検知できない課題や，Web アプリケーションの実装に依存するため，Web アプリケーション毎に対策の実装が必要となる課題がある．

Web サービスの運営において，不正クエリ対策の実施は重要であるが，サービス運営者は，対策導入によって既存のシステム構成や開発プロセスが変更され，サービスの開発や運用に支障を出すことは避けたい．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用いられるため，Web アプリケーションの実装に依存した対策は複数の実装が必要となる．そのため，不正クエリ対策は，導入時のシステム構成や開発プロセスへの影響を抑えつつ，Web アプリケーションの実装に依存せず利用できる必要がある．これらの要件を満たしながら，高い不正クエリの検知精度が求められる．

本研究では，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，テスト実行時に発行される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．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．提案手法は，Web アプリケーションが稼働する前のテストの段階においてホワイトリストを自動作成することで，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可能な状態にできる．また，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することで収集し，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．提案手法で検知されるクエリは，テスト時に発行されなかったクエリ，もしくは不正クエリである．

本論文では，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クエリを不正と判断した割合とする．また，False negative は，Web アプリケーションの脆弱性攻撃によって発行される，開発者の想定外のクエリを不正と判断できなかった割合とする．実験において，Web アプリケーションにブラウザから HTTP リクエストを与えた時に発行された正常なクエリデータと，SQL インジェクション攻撃によって発行された不正なクエリデータを用いて，ホワイトリストの False positive と False negative を計測し提案手法を評価する．

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，提案手法の設計について述べる．4 章では，実験により，提案手法によって作成したホワイトリストの False positive と False negative を評価し，5 章で考察を述べ，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．ホワイトリストの登録内容としては，Web アプリケーションが発行するクエリはユーザ入力により変化することから，ユーザ入力が含まれる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が用いられる [5]．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，Web アプリケーションの改修頻度は高いため [14]，Web アプリケーションが発行するクエリは頻繁に変化する．開発者は，ホ

ワイトリストの False positive の増大を防ぐために，発行クエリに変化がある度にホワイトリストを更新しなければならないので，開発者への負担は大きい．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）[1] 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[2] を用いた場合，ORM として ActiveRecord[3] 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，開発者はクラスオブジェクトを用いてクエリの発行処理を記述できる．例えば，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，「User.find(1)」と Ruby のコードを記述すると，「SELECT * FROM users WHERE (users.id = 1) LIMIT 1」というクエリが発行される．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読み出せるので，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，作業量が膨大になり 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，ホワイトリストを手動で作成することは開発者への負担が大きい方法となるため，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，ホワイトリストを定義する手法が提案されている [11]．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，即時に不正クエリを検知することができない．また，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．これは，Web アプリケーション稼動中にホワイトリストを作成していることが原因で発生し，Web アプリケーション稼動後即時に不正クエリの検知を行うためには，稼動前の段階でホワイトリストを作成しておく必要がある．学習フェーズとして，過去に収集しておいたクエリのログからホワイトリストを作成することもできる．しかし，収集しておいたクエリログには，Web アプリケーション改修によって新しく発行されるようになったクエリが含まれない場合や，発行されなくなったクエリが含まれている場合があるため，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保つことができない．

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からクエリを発行する処理を特定し解析することで，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 [4]．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することができ，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．一方で，この手法はソースコード内のクエリを発行する処理の解析を行うため，Web アプリケーションの実装に依存する．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，実装方法の異なる複数の Web アプリケーションで利用する場合，実装しなければならない解析器は増大してしまう．そのため，実装方法の異なる複数の Web アプリケーションで利用するためには，Web アプリケーションの実装に依存しないような方法を取るべきである．

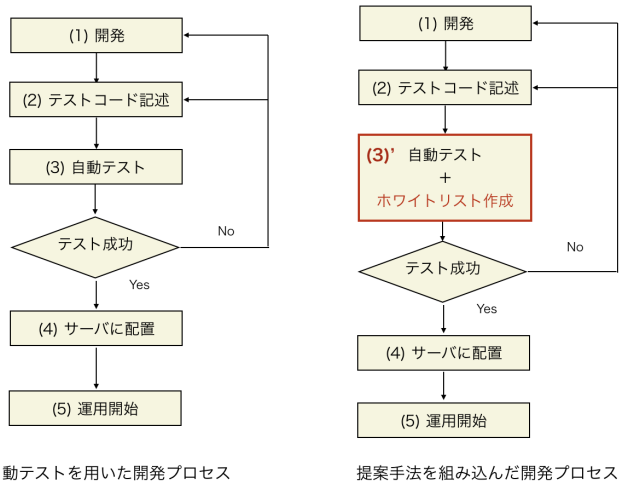
3. 提案手法

提案手法は，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自動作成する．提案手法によって，テストの段階でホワイトリストを作成できるため，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，システム構成への影響を抑えつつ，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自動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．開発プロセスに提案手法を適用する方法を示すために，自動テストを採用した開発プロセスと提案手法の位置付けを図 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



自動テストを用いた開発プロセス　　提案手法を組み込んだ開発プロセス

図 1　自動テストを用いた Web アプリケーションの開発プロセス
Fig. 1　Web application development process using automatic testing

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，開発した機能は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 の提案手法を組み込んだ開発プロセスについて説明する．提案手法では，図 1 の (3)' のように，自動テストの実行中に Web アプリケーションから発行されたクエリを用いて，ホワイトリストを作成する処理を追加する．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，開発による機能追加や修正などによって変化するため，ホワイトリストはクエリの変化に対応する必要がある．自動テストを用いた開発プロセスでは，テストコードは Web アプリケーションの機能変更に応じて開発者によって整備され，提案手法は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作成するため，発行クエリに変化があった場合も 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作成できるので，新しい Web アプリケーションの運用を開始するときには，不正クエリを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にホワイトリストを作成する方法では実現できない．また，提案手法は，開発プロセスを変更せずにホワイトリストを作成でき，導入時の既存の開発プロセスへの影響を低減できる．

自動テストを用いた開発プロセスにおいて，テストコードを整備することと，手動でクエリ構造を登録しホワイトリストを整備することの開発者へ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．テストコードを書く場合，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．一方で，ホワイトリストを作成する場合，開発者は，Web アプリケーションの動作を理解した上で，その過程で発行されるクエリ発行の動作を理解しなければならない．そのため，テストコードを整備することは，ホワイトリストを整備することに比べ，要求される Web アプリケーションの動作の知識が少ないため，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

提案手法は，テストコードを元に Web アプリケーションを動作させ，その過程で発行されたクエリを用いてホワイトリストを作成する．そのため，提案手法によって作成されたホワイトリストで検知されるクエリは，テスト時に発行されなかったクエリである．このようなクエリには，テストされていないクエリと不正クエリが含まれる．テストされていないクエリは，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である．不正クエリは，Web アプリケーションの脆弱性を利用した攻撃によって生じる想定外のクエリである．また，ホワイトリストには，テスト時のみ発行されるクエリも登録される．テスト時のみ発行されるクエリの例として，動作テストに用いるテストデータを登録するクエリや，登録したテストデータを削除するクエリが挙げられる．図 2 に前述したクエリの内包関係を示す．

図 2 より，提案手法は，テストケースの追加や，テストカバレッジの向上よって，テストされているクエリ領域を拡大し，Web アプリケーションが発行するクエリ領域に近づけることで，ホワイトリストの False positive を低減できる．このことから，開発者がテストケースを追加することで，テストされていないクエリをホワイトリストに登録できるが，テストケース増加による管理の複雑化が懸念される．そのため，テストケースの追加を行わず，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必要である．また，ホワイトリストに登録されたテスト時のみ発行

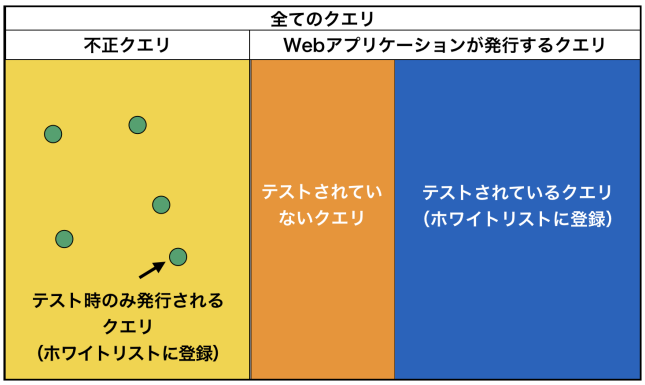


図 2 クエリの内包関係
Fig. 2 Relationship of queries

されるクエリは，Web アプリケーションが発行するクエリではないため，False negative につながる可能性がある．しかし，Web アプリケーションの脆弱性を利用した攻撃として SQL インジェクション攻撃を想定した場合，SQL 文を含む入力を与えることで Web アプリケーションが発行するクエリに任意の SQL 文をインジェクションするため，テスト時のみ発行されるクエリと発行されたクエリが一致する可能性は低いと考えられる．そのため，提案手法は，SQL インジェクションによる不正クエリを検知するのに有効であると考えられる．

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の発行を検知できる．そのため，SQL インジェクション攻撃のように，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以下に示す．

- 正常：SELECT * FROM users LIMIT 30
- 異常：SELECT * FROM users LIMIT 1000

上記したクエリは，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が受け取ったクエリは，テストの実行を妨げないために，そのままデータベースに渡される．このとき，データベースプロキシは通過したクエリを記録しておく．全てのテストが終了した後，データベースプロキシに記録されたクエリを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．

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，データベースプロキシは，クエリ構造に変換し，クエリ構造とホワイトリストの照合を行う．このとき，ホワイトリストに登録されていた場合は，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．

不正クエリを検知する処理には，発行されたクエリとホワイトリストを照合する処理が含まれ，Web アプリケーションとデータベース間のレイテンシーの増大が考えられる．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，結果的に，Web アプリケーションのユーザへのレスポンス時間が増大することが懸念される．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，ホワイトリストを作成する時に，クエリ構造をキーとしたハッシュテーブルを作成しておき，ハッシュテーブルを用いてホワイトリストの照合処理を行うことで，探索の計算量は $O(1)$ に抑えられ，Web アプリケーションとデータベース間のレイテンシーの増大を抑制できると考えられる．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，リストの照合処理の計算量は一定となる．

4. 実験

提案手法の有効性を確認するために，図 4 に示す実験環境を構築し，テストカバレッジに対する提案手法によって作成したホワイトリストの False positive と，False negative とホワイトリストの照合に要する時間の評価を行った．ただし，False positive は，開発者が想定するユーザ入力に

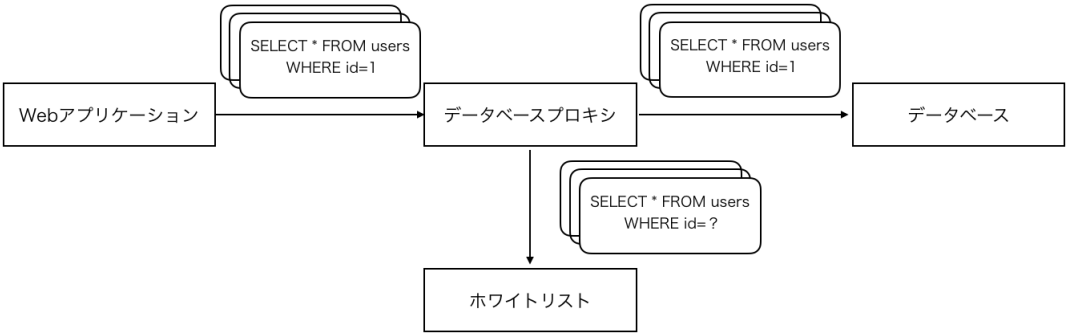


図 3 提案手法のアーキテクチャ
Fig. 3 Propose method architecture

図 4 実験環境の構成
Fig. 4 Experimental environment

表 1 実装メソッド一覧
Table 1 List of Implementation method

HTTP	URL	操作内容
GET	articles	全ての article を表示
GET	articles?title=""	title で検索して article 表示
POST	articles	article を一つ作成
GET	articles/:id	特定の article を表示
PATCH	articles/:id	特定の article を更新
DELETE	articles/:id	特定の article を削除

よって，Web アプリケーションが発行するクエリを不正と判断した割合とする．False negative は，Web アプリケーションの脆弱性攻撃によって発行される，開発者の想定外のクエリを不正と判断できなかった割合とする．

実験環境では，データベースプロキシとして ProxySQL[7] 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 アプリケーションの実装には Ruby on Rails を用いた．表 1 のように，article テーブルに対して，新規作成，データ一覧・個別取得，データの更新，データの削除の操作を行うメソッドを実装した．また，SQL インジェクションのインジェクションの脆弱性がある article を title で検索するメソッドを実装した．

4.1 ホワイトリストの False positive と False negative

3.2 節で述べた提案手法の検知特性から，提案手法は，SQL インジェクション攻撃を漏れなく検知できると考えられる．また，提案手法単体では，全ての 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登録することはできないため，False positive が発生すると考えられる．これらの特性を確認するために，実験では，テストカバレッジに対する False positive と False negative を計測した．提案手法はテストカバレッジによってホワイトリストに登録されるクエリ構造が変化するため，実験では Web アプリケーションのテストカバレッジの計測を行った．Web アプリケーションのテストカバレッジの算出には，テスト時に実行されたソースコードの行を計測して，行単位でのカバレッジを算出する SimpleCov[9] を利用した．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 1 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 個のクエリを用意した．用意した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[10] 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 Web アプリケーションから発行された 265 個のクエリを用意した．用意した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．

実験結果はテストカバレッジ 68.97%に対して，False positive が 17.65%，False negative が 0%となった．実験結果より，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たことがわかる．また，False positive は 17.65%発生したため，正常な入力によって Web アプリケーションが発行するクエリの一部を誤検知することがわかった．

ホワイトリストに False positive が発生した原因について考察する．表 1 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．False positive として誤

検知されたクエリを以下に示す．

- 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‘ = ?

誤検知されたクエリは全て表 1 における PATCH リクエストを送った時に動作する UPDATE メソッドが発行するクエリであった．これらのクエリが発行されなかった原因は，テスト時に UPDATE メソッドが更新しようとしている article データと，データベース上にある更新前の article データが同一になっていたことである．更新前の article データと更新しようとしている article データが同一の時，Ruby on Rails で実装した Web アプリケーションは UPDATE のクエリを発行せずテストは成功する．また，UPDATE メソッドの処理はテストカバレッジの算出にも含まれる．このことから，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 を発生させる原因となることが確認できた．このようなケースに対応し，False positive を低減するために，提案手法によってホワイトリストに登録されなかったクエリを補う方法が必要となる．

ホワイトリストに False negative が発生しなかった原因について考察する．実験では，False negative は観測されなかったが，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の一覧を調査した．調査の結果，ホワイトリスト全体に対してテスト時のみ発行されるクエリは 39.13%含まれており，含まれていたクエリの一覧を以下に示す．

- 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

この結果から，提案手法では，上記した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．しかし，SQL インジェクション攻撃は 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することによって引き起こすため，上記したクエリ構造と一致する可能性は低いと考えられる．この理由から，ホワイトリストに False negative が発生しなかったと考えられる．そのため，提案手法は，SQL インジェクション攻撃によって発行される不正クエリの検知に有効であると考えられる．

4.2 ホワイトリストの照合時間

3.3 節で述べたホワイトリストのエントリ数に依存せずにリストの照合を行えることを示すために，不正クエリの判定を行うソフトウェアを実装し，実験を行なった．実装したソフトウェアでは，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キーとするハッシュテーブルとしてホワイトリストを作成している．そして，このソフトウェアは，ハッシュテーブルを用いて標準入力から与えられたクエリの照合を行い，検知したものを標準出力に出力する．ホワイトリストのエントリ数が増加した場合も高速に照合できることを示すために，エントリ数 23 個とその 100 倍である 2300 個のホワイトリストを用意した．また，ハッシュテーブルを用いない場合はエントリ数に対して線形に照合の計算量が増える，ホワイトリストに登録されていないクエリを用意した．これらを実装したソフトウェアに与え，CPU の使用状況による実行時間の違いを抑えるために，照合に要した時間を 1000 回計測し，平均値を算出した．実験は，CPU が Intel Core i7 3.3 GHz 2core，メモリが 16 GB の環境で行った．

実験の結果，照合に要した平均時間は，エントリ数 23 個の時に 140.8μs，エントリ数 2300 個の時に 151.9μs であった．この結果から，照合に要した平均時間の差は 11.1μs と小さく，エントリ数がリスト照合に要する時間に与える影響は小さいことが確認できた．

5. 考察

提案手法は，自動テストを用いた開発プロセス内でホワイトリストを作成する．これによって，開発者は，既存の開発プロセスを変更することなく開発を行え，開発の過程で不正クエリの検知に利用できるホワイトリストを得ることができる．テストコードは開発プロセス内で Web アプリケーションの変更に応じて整備されるため，開発の過程で作成されるホワイトリストは発行されるクエリに変化があった場合にも対応できる．また，提案手法はデータベースプロキシでクエリを収集しホワイトリストを作成するため，Web アプリケーションの実装に依存せず，複数の Web アプリケーションで共通して利用できる．これらの特性から，提案手法は単一の実装で複数の Web アプリケーションに適用でき，かつ提案手法を導入することによる Web サービスの開発や運用に与える影響は小さいと考えられる．

4.1 節での実験結果から，提案手法は，SQL インジェクション攻撃によって発行される不正クエリを検知できるこ

とが確認された．しかし，全ての実装されたメソッドに対してテストを記述したとしても，メソッドが発行するクエリはテストケースによって変化する場合があるため，ホワイトリストに登録できないクエリが存在し，正常なクエリの一部を誤検知してしまうことが確認された．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．このことから，より大規模な Web アプリケーションに提案手法を適用する場合，発行クエリ数の増加により，誤検知されるクエリ数は増大すると考えられる．この問題を解決するために，Web アプリケーションが発行するクエリの内，ホワイトリストに登録できなかったクエリを補うための方法の検討が必要である．また，ホワイトリストには，テストデータを登録するクエリのようなテスト時のみ発行されるクエリが含まれる．これらのクエリは，ユーザ入力によって Web アプリケーションから発行されるクエリではなく，テーブルのデータを全て削除するようなクエリが含まれた場合には，攻撃が通り抜けてしまうため，ホワイトリストに含めるべきではない．そのため，テスト時のみ発行されるクエリへの対処方法の検討が必要である．

4.2 節の実験結果から，クエリ構造をキーとしたハッシュテーブルを用いて，ホワイトリストを表現することで，リストのエントリ数がリストを照合する時間に与える影響は小さいことが確認できた．このことから，大規模な Web アプリケーションに提案手法を適用した場合も，高速にリストを照合できる．

提案手法は，ホワイトリストの作成方法の特性と，実験で確認した不正クエリの検知特性やレイテンシーの抑制効果から，Web サービスの開発や運用への影響を小さく抑え実現できる SQL インジェクション対策として，有効に利用できると考えられる．

6. まとめ

本稿では，Web アプリケーションが利用するデータベースに発行される不正クエリを検知するために，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 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を提案し，実験により，提案手法の検知の特性を評価した．提案手法は，開発プロセスにホワイトリストの自動作成を組み込むことによって，Web サービスの開発者と運用者の負荷を低減しつつ，不正クエリの検知を行うことができると考えられる．

実験から，提案手法は SQL インジェクション攻撃によって発行された不正クエリを False negative を発生させず検知できることを確認し，SQL インジェクション攻撃に対して有効に利用できることがわかった．しかし，対象とするメソッドに対してテストを記述していたとしても，テストケースによって発行クエリに変化が生じる場合があるため，ホワイトリストに登録されないクエリが存在し，False positive が発生することが確認された．

今後の課題としては，False positive を低減するために，Web アプリケーションが発行するクエリの内，ホワイトリストに登録できなかったクエリを補うための方法や，データベースにテストデータを登録するようなテスト時のみに発行されるクエリへの対処方法の検討が挙げられる．また，今後は，提案手法のホワイトリストを用いた検知の実装を進め，提案手法の導入による Web アプリケーションとデータベース間のレイテンシーへの影響の検証や大規模な Web アプリケーションにおける False positive と False negative の検証を行う．

参考文献

[1] Ambler Scott W, Mapping objects to relational databases, White Paper, Ronin International, 2000.
[2] Bächle Michael and Paul Kirchberg, Ruby on rails, IEEE software, pp.105-108, November, 2007.
[3] Fowler Martin, Patterns of enterprise application architecture, pp.147-150, 2002.
[4] Halfond William GJ and Alessandro 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, November, 2005.
[5] Kar Debabrata and Suvasini Panigrahi, Prevention of SQL Injection attack using query transformation and hashing, Advance Computing Conference (IACC), 2013 IEEE 3rd International, pp.1317-1323, May, 2013.
[6] Kemalis Konstantinos and Theodores Tzouramanis, SQL-IDS: a specification-based approach for SQL-injection detection, Proceedings of the 2008 ACM symposium on Applied computing, pp.2153-2158, March, 2008.
[7] ProxySQL, http://www.proxysql.com/.
[8] Rietta Frank S, Application layer intrusion detection for SQL injection, Proceedings of the 44th annual Southeast regional conference, pp.531-536, March, 2006.
[9] SimpleCov, https://github.com/colszowka/simplecov.
[10] sqlmap, http://sqlmap.org/.
[11] Valeur Fredrik, Darren Mutz and Giovanni 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.
[12] IPA 独立行政法人 情報処理推進機構, 安全なウェブサイトの作り方 改訂第 7 版, 3 月 2015.
[13] NPO 日本セキュリティ協会セキュリティ被害調査ワーキンググループ, 長崎県立大学 情報システム学部情報セキュリティ学科, 2016 年情報セキュリティインシデントに関する調査報告書 個人情報漏洩編 第 1.2 版, 6 月, 2017.
[14] 青井 翔平, 坂本 一憲, 鷲崎 弘宜, 深澤 良彰, DePoT:Web アプリケーションテストにおけるテストコード自動生成テスティングフレームワーク, 情報処理学会論文誌 Vol.56 No.3 pp.835–846, 3 月, 2015.
[15] 藤田 直行, 侵入検知に関する誤検知低減の研究動向, 電子情報通信学会論文誌 B Vol.J89–B No.4 pp.402–411, 4 月, 2006.