修士論文

クロスサイトスクリプティング攻撃に対する脆弱性の静的解析と評価

Static Analysis and Evaluation Metrics of Cross-site Scripting Attacks

指導教員　　松浦幹太　教授

東京大学大学院 情報理工学系研究科 電子情報学専攻

48-156430　　林 昌吾

平成 29 年 2 月 2 日提出

内容梗概

クロスサイトスクリプティング (XSS) 攻撃は数多くの研究がなされてきているにもかかわらず常に首位を争う脅威となっている攻撃であり, 攻撃手法は多様化している. 防御策について, Open Web Application Project(OWASP) のようにチェックシートを公開している組織も存在するが, 開発者全員がこれらを基に安全な実装をするのは困難である. それ故, 脆弱性を自動で検知するツールがあれば大いに役に立つと言え, そのような研究も存在する. しかし, 既存研究では精度が低い. 特に, スクリプト言語に適用可能なオブジェクト指向に対する研究はあまりなされていない. また, 静的解析や動的解析など様々な解析手法が考案されている. しかし, ソースコード中の脆弱性箇所周辺の文脈や Web アプリケーションの利用環境などによる脆弱性判定の基準が解析ツールの利用者には不明瞭であることが多い.

本研究では 2 方向後方脆弱性分析とクラスキャッシュという概念を導入して, 解析手法の提案と実装を行う. また, 解析における脆弱性の危険度の評価, 脆弱性箇所周辺の文脈に応じて的確に判定できる陽性の判定基準の明確化, 対策の優先度判定に貢献するための脆弱性のメトリクスの提案と実装を行う.

目次

Chapter 1　序論

1.1　クロスサイトスクリプティング攻撃

Web アプリケーションは日常生活の様々な場面で利用される. しかし, インターネットに出回っている Web アプリケーションには多くの脆弱性が存在する. 中でもクロスサイトスクリプティング (Cross site scripting: XSS) 攻撃は特に深刻なセキュリティ攻撃の内の 1 つである. XSS は 1990 年代から存在が確認されており [1], これまでに 10 年以上に渡って数多くの研究がなされてきている. Open Web Application Security Project(OWASP) [33] という,Web アプリケーションで最も深刻な脆弱性の上位 10 位を発表を行っている団体がある. 世の中に XSS の被害が浸透していっているにも関わらず,OWASP によると,2001 年の設立以来常に対象となっているものであり, インターネットの利用者や Web サイトの運営者が被害を受けている. このような現状に対して Hydara 氏らは 2015 年に Web アプリ開発者は各フェーズでもっとセキュリティを意識し, 研究者は XSS を排除することに意識をより傾けるべきだと主張している [3].

OWASP のように XSS 攻撃の防御のためにチェックシートを公開している組織もある. しかし, これらは複雑で量も多いため Web アプリケーション開発者全員が表を基に適切に実装することは容易ではない. また, 対策に対して漏れが生じる可能性が極めて高い. さらには, チェックシートに記述されている対策もまた, 完全なものではない. そのため, 開発者が実装したコードに存在する脆弱性を自動検知できるツールがあれば大いに役に立つ. 実際, そのような研究もなされてきている.

Web アプリケーションの中でも, PHP のプログラミング言語によって実装されたサーバサイドの言語は全体の内, 80 %以上を占める [4]. PHP で書かれたコードに対する脆弱性の自動検知に対する研究はされてきている [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. しかしながら, 既存研究では精度は低い.

オブジェクト指向言語によって書かれたコードの解析に対する研究もなされてきている [18]. しかし, これらは Java 言語などの型付き言語を対象としたものであり, 型の概念の弱い PHP のようなスクリプト言語には直接適用することはできない. そのような現状に対して,Dahse 氏らによって 2014 年に一部の攻撃を対象にオブジェクト指向言語に対する脆弱性の検知に対する研究を発表している [14, 15]. これは PHP Object Injection(POI) 攻撃[1]を対象としたものである. しかし, PHP によって実装されたものに限定され, Web アプリケーションの脆弱性全体の観点では十分な対策とは言い難いものである.

[1]Code Reuse 攻撃の一つで, PHP におけるオブジェクト指向に対する攻撃

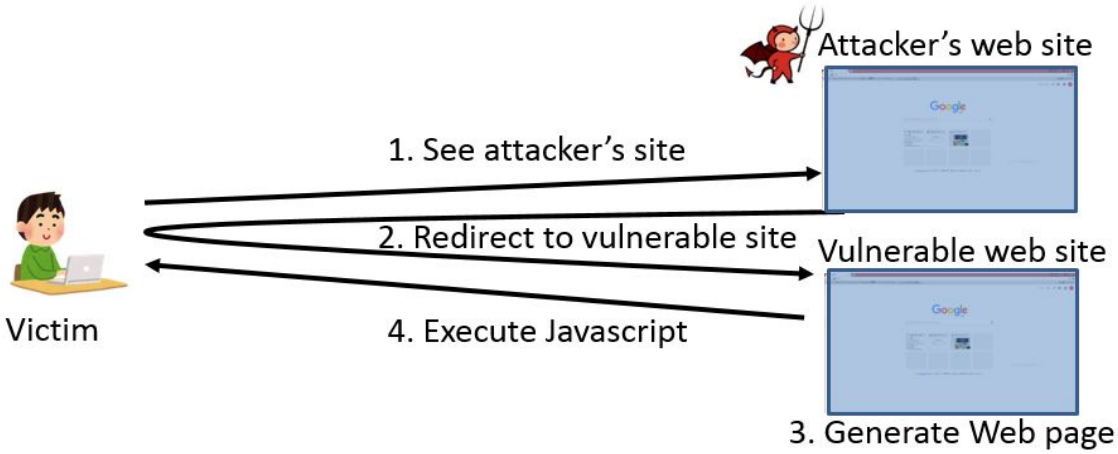

図 1.1: 反射型 XSS

一方で, 既存研究の解析では, ある脆弱性の可能性のある箇所に対して脆弱かどうかを 2 値で判定しており, ソースコード中の脆弱性箇所周辺の文脈や Web アプリケーションの利用環境などによる脆弱性判定の基準が解析ツールの利用者には不明瞭であることが多い.

本稿では, 型の概念の無いスクリプト言語を対象に, 脆弱性の自動検知の静的解析を行う. セキュリティ攻撃の中でも最も問題の大きいとされる XSS 攻撃について, クラスキャッシュと 2 方向後方脆弱性分析という概念を導入して, オブジェクト指向言語の静的解析を行う. 実装は PHP を対象に行った. また, 解析における脆弱性の危険度の評価, 脆弱性箇所周辺の文脈に応じて的確に判定できる陽性の判定基準の明確化, 対策の優先度判定に貢献するための脆弱性のメトリクスの提案と実装を行う.

1.2　関連知識

1.2.1　クロスサイトスクリプティング攻撃の種類

Reflected XSS

Reflected XSS は, 図 1.1 の通りとなる. Web サイトの利用者が自身によって PHP コード実装された web サイトに入力を与えた時に生じるものである. これは, 攻撃者がクエリー等に Javascript による攻撃コードを埋め込んでおいた URL を用意しておき, これを用いて利用者にアクセスさせること等によって生じるという間接的な攻撃である. 埋め込まれたコードはサーバサイド側で実行される.

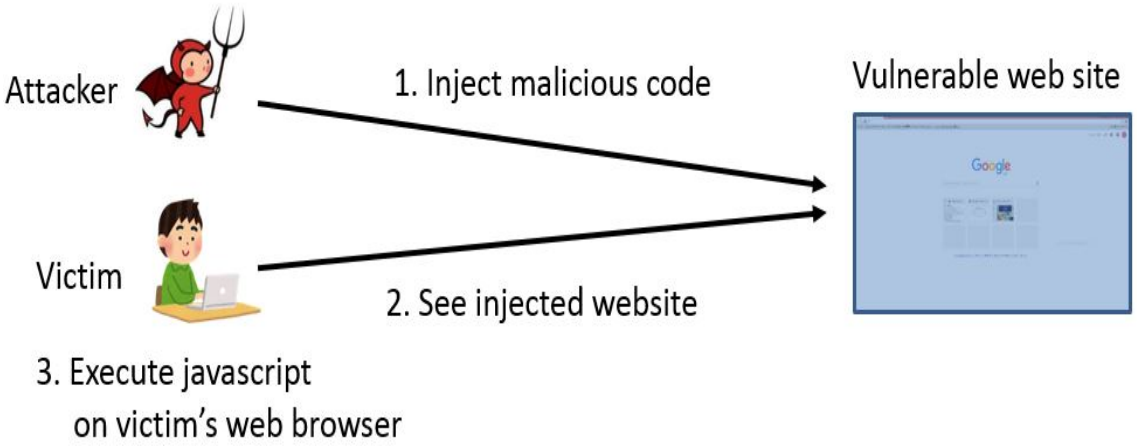

図 1.2: 蓄積型 XSS

Stored XSS

Stored XSS は, 図 1.2 の通りとなる. 攻撃者が予め Web サイトの入力に Javascript による攻撃コードを与えておきそれがデータベース等に保存される. 利用者がこの攻撃コードが与えられた Web サイトを訪れる時にこのコードが実行される. という直接的な攻撃であり, 埋め込まれたコードはサーバサイド側で実行される. 例えば, 掲示板のサイトなどでの被害がある.

DOM based XSS

DOM based XSS は, 図 1.3 の通りとなる. 2005 年に Klein らによって最初に言葉が用いられるようになった XSS で, 他の種類の XSS と比較して新しい概念である [37, 38]. DOM based XSS も Reflected XSS のように攻撃者が用意しておいた URL を利用者にアクセスされて間接的に攻撃されるものである. しかし,Reflected XSS との大きな違いは DOM based XSS はサーバサイド側ではなくクライアントサイド側で実行されるというものである. DOM based XSS の顕著な特徴としては, サーバサイドには攻撃コードを必ずしも送る必要がなく, サーバサイドでは検知できない状況も考えられる.

1.2.2　脆弱性の評価手法

CVSS

CVSS とは Common Vulnerability Scoring System の略称で共通脆弱性指標である. これは, 情報システムの脆弱性に対するオープンで包括的, 汎用的な評価手法の確立と普及を目指して, 米国家インフラストラクチャ諮問委員会 (NIAC: National Infrastructure Advisory Council) によって作成されたものである.

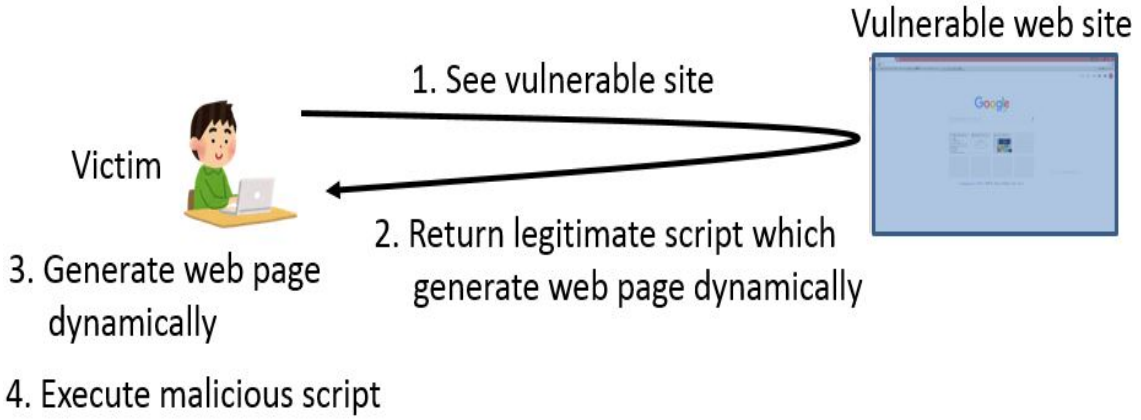

図 1.3: DOM based XSS

CVSS は以下の 3 つの要素で構成される.

1. 基本評価基準
2. 現状評価基準
3. 環境評価基準

これらの要素は順番に算出されるものであり, 基本評価基準を計算し, これを元に現状評価基準を計算し, そこから環境評価基準を計算する.

基本評価基準　基本評価基準とは, 脆弱性そのものに対する深刻度を 0 から 10 の間で数値化したものである. 基本評価基準は, 情報セキュリティの三大要件である機密性, 完全性, 可用性に影響を与える度合いに対する基準を攻撃元区分や攻撃条件の複雑さ, 攻撃前の認証要否を基準として, 以下の計算式で値を算出する.

$$\text{影響度} = 10.41 * (1 - (1 - C) * (1 - I) * (1 - A)) \tag{1.1}$$

$$\text{攻撃容易性} = 20 * AV * AC * Au \tag{1.2}$$

$$\text{f(影響度)} = \begin{cases} 0(\text{影響度} = 0) \\ 1.176(\text{影響度} \neq 0) \end{cases} \tag{1.3}$$

この 3 式を基に以下のようにして基本値を算出する.

$$\text{基本値} = ((0.6 * \text{影響度}) + (0.4 * \text{攻撃容易性}) - 1.5) * \text{f(影響度)} \tag{1.4}$$

表 1.1: 基本値のパラメータ

基本評価基準	評価結果	値
AV	ローカル	0.395
	隣接	0.646
	ネットワーク	1.0
AC	高	0.35
	中	0.61
	低	0.71
Au	複数	0.45
	単一	0.56
	不要	0.704
C	なし	0.0
	部分的	0.0.275
	全面的	0.660
I	なし	0.0
	部分的	0.0.275
	全面的	0.660
A	なし	0.0
	部分的	0.0.275
	全面的	0.660

ただし, 基本値の小数点第 2 位は四捨五入する. 上記の表におけるパラメータの詳細は図 1.1 の通りとなる.

AV は Access Vector の略で, 攻撃元区分を表し, 攻撃がどこから可能であるかの指標となる. この基準は, ローカル, 隣接, ネットワークの 3 つに分類される. ローカルとは対象システムが物理的もしくはローカル環境から攻撃する必要があるかを表す. 隣接とは隣接ネットワークから攻撃できるかを表す. ネットワークとは, ネットワーク経由でリモートから攻撃可能であるかを表す.

AC は Access Complexity の略で, 攻撃条件の複雑さを表し, 攻撃に必要な条件の複雑さの指標となる. この基準は, 高, 中, 低の 3 つに分類され, 高は, 権限昇格や偽装などで情報収集しておき, 対象システムが特定の場合に攻撃可能である場合である. 中は, いくつか情報を収集しておき, 大将システムが標準以外の設定になっている場合である. 低は, 対象システムが常に攻撃可能である場合である.

Au は Authentication の頭文字で, 攻撃前の認証の要否を表す. 攻撃するにあたって認証が必要かを表す. この基準は, 複数, 単一, 不要の 3 つに分類される. 複数は, 攻

撃のために認証が 2 つ以上必要であることを表す. 単一は, 攻撃のために認証が 1 つ必要であることを表す. 不要は, 攻撃に認証が必要ないことを表す.

C は Contidentiality Impact の 1 文字目であり, 機密性への影響を表す. この基準は, なし, 部分的, 全面的の 3 つに分類される. なしは, システムの機密性に影響がないことを表す. 部分的は, 一部の機密情報が参照可能であることを表す. 全面的は, すべての機密情報が参照可能であることを表す.

I は Integrity Impact の 1 文字目であり, 完全性への影響を表す. この基準は, なし, 部分的, 全面的の 3 つに分類される. なしは, システムの完全性に影響がないことを表す. 部分的は, システムの一部の情報またはシステムファイルが改竄可能であることを表す. 全面的は, システム全体の情報またはシステムファイルが改竄可能であることを表す.

A は Availability Impact の 1 文字目であり, 可用性への影響を表す. この基準は, なし, 部分的, 全面的の 3 つに分類される. なしは, システムの可用性に影響がないことを表す. 部分的は, リソースの一部枯渇, または業務の一時中断可能であることを表す. 全面的は, リソースの全体の枯渇, またはシステムの完全な停止が可能であることを表す. この基準は時間の経過や利用環境に依存しない.

現状評価基準　現状評価基準とは, 現在における脆弱性の深刻度を 0 から 10 の間で数値化したものである. 現状評価基準は, 攻撃される可能性や利用可能な対策のレベル, 脆弱性情報の信頼性を基準として, 以下の計算式で値を算出する.

$$\text{現状値} = \text{基本値} * \text{E} * \text{RL} * \text{RC} \tag{1.5}$$

ただし, 現状値の小数点第 2 位は四捨五入する.

上記の表におけるパラメータの詳細は図 1.2 の通りとなる.

E は Exploitability の 1 文字目を表し, 攻撃される可能性を表す. この基準は, 未実証, 実証可能, 攻撃可能, 容易に攻撃可能, 未評価の 5 つに分類される. 未実証とは, エクスプロイトが存在しない, または攻撃手法が理論上のみで存在することを表す. 実証可能とは, 完成度の低いエクスプロイトが存在することを表す. 攻撃可能とは, ほとんどの状況で使用可能なエクスプロイトが存在することを表す. 容易に攻撃可能とは, あらゆる状況で使用可能なエクスプロイトが存在する, またはエクスプロイトが無くても攻撃可能であることを表す. 未評価とは, この項目を評価しない場合に使用する.

RL は Remediation Level の頭文字を表し, 利用可能な対策のレベルを表す. この基準は, 正式, 暫定, 非公式, なし, 未評価の 5 つに分類される. 正式とは, 製品開発者からの正式な対策が利用できる場合を表す. 暫定とは, 製品開発者からの暫定的な対策が利用できる場合を表す. 非公式とは, 製品開発者以外からの対策が利用できる場合を表す. なしとは, 攻撃に対する対策がない場合を表す. 未評価とは, この項目を評価しない場合に使用する.

表 1.2: 現状値のパラメータ

基本評価基準	評価結果	値
E	未実証	0.85
	実証可能	0.90
	攻撃可能	0.95
	容易に攻撃可能	1.00
	未評価	1.00
RL	正式	0.87
	暫定	0.90
	非公式	0.95
	なし	1.00
	未評価	1.00
RC	未確認	0.45
	未確証	0.56
	確認済	1.00
	未評価	1.00

RC は Report Confidence の頭文字を表し, 脆弱性情報の信頼性を表す. この基準は, 未確認, 未確証, 確認済み, 未評価の 4 つに分類される. 未確認とは, 脆弱性の未確認情報が 1 件存在する, または相反する情報が存在する場合を表す. 未確証とは, セキュリティベンダーや調査団体から, 複数の非公式な情報が存在する場合を表す. 確認済みとは, 製品開発者が脆弱性情報を確認し, 脆弱性情報がエクスプロイトなので確認されている場合を表す. 未評価とは, この項目を評価しない場合に使用する.

この基準は脆弱性の対応状況により, 時間の経過と共に値が変化する可能性がある.

環境評価基準　環境評価基準とは, 製品利用者の利用環境も含め, 最終的な脆弱性の深刻度を 0 から 10 の間で数値化したものである. 環境評価基準は, 二次的被害の可能性や影響を受ける対象システムの範囲, または情報セキュリティの三大要件である機密性, 完全性, 可用性の重要視するものを基準として, 以下の計算式で値を算出する.

$$\begin{aligned} \text{調整後影響度} = min(10.0, 10.41 * (1 - (1 - C * CR) * \\ (1 - I * IR) * (1 - A * AR))) \end{aligned} \tag{1.6}$$

調整後現状値 = (式 1.3 と式 1.4 の影響度に先ほど計算した
調整後影響度の計算結果を代入し, 基本値を再計算する.　(1.7)
その基本値で式 1.5 で再計算した現状値)

この 2 式を基に以下のようにして環境値を算出する.

環境値 = (調整後現状値 + (10 - 調整後現状値) * CD) * TD　(1.8)

ただし, 環境値の小数点第 2 位は四捨五入する.

上記の表におけるパラメータの詳細は表 1.3 の通りとなる.

CD は Collateral Damage の頭文字を表し, 二次的被害の可能性を表す. この基準は, なし, 軽微, 中程度, 重大, 壊滅的, 未評価の 6 つに分類される. なしとは, 攻撃による二次的被害が発生しないことを表す. 軽微とは, 攻撃が成功すると軽微な被害が発生することを表す. 中程度とは, 攻撃が発生すると中程度な被害が発生することを表す. 重大とは, 攻撃が発生すると重大な被害が発生することを表す. 壊滅的とは, 攻撃が発生すると壊滅的な被害が発生することを表す. 未評価とは, この項目を評価しない場合に使用する.

TD は Target Distribution の頭文字を表し, 影響を受ける対象システムの範囲を表す. この基準は, なし, 小規模, 中規模, 大規模, 未評価の 5 つに分類される. なしとは, 対象システムがない, または物理的に隔離されているために対象システムへのリスクがないことを表す. 小規模とは, 対象システムが, 利用環境の 1 から 25%にリスクがあることを表す. 中規模とは, 対象システムが, 利用環境の 26 から 75%にリスクがあることを表す. 大規模とは, 対象システムが, 利用環境の 76 から 100%にリスクがあることを表す. 未評価とは, この項目を評価しない場合に使用する.

CR は Confidentiality Requirement の略で, 機密性の要求度を表す. これは機密性を特に重視する場合, 比重をつける. この基準は, 低, 中, 高, 未評価の 4 つに分類される. なしとは, 対象システムがない, または物理的に隔離されているために対象システムへのリスクがないことを表す. 低とは, 対象システムの損失時に, 影響が一部であることを表す. 中とは, 対象システムの損失時に, 深刻な影響があるを表す. 高とは, 対象システムの損失時に, 壊滅的な影響があるを表す. 未評価とは, この項目を評価しない場合に使用する.

IR は Integrity Requirement の略で, 完全性の要求度を表す. これは完全性を特に重視する場合, 比重をつける. この基準は, 低, 中, 高, 未評価の 4 つに分類される. なしとは, 対象システムがない, または物理的に隔離されているために対象システムへのリスクがないことを表す. 低とは, 対象システムの損失時に, 影響が一部であることを表す. 中とは, 対象システムの損失時に, 深刻な影響があるを表す. 高とは, 対象シス

テムの損失時に, 壊滅的な影響があるを表す. 未評価とは, この項目を評価しない場合に使用する.

AR は Availability Requirement の略で, 可用性の要求度を表す. これは可用性を特に重視する場合, 比重をつける. この基準は, 低, 中, 高, 未評価の 4 つに分類される. なしとは, 対象システムがない, または物理的に隔離されているために対象システムへのリスクがないことを表す. 低とは, 対象システムの損失時に, 影響が一部であることを表す. 中とは, 対象システムの損失時に, 深刻な影響があるを表す. 高とは, 対象システムの損失時に, 壊滅的な影響があるを表す. 未評価とは, この項目を評価しない場合に使用する.

この基準は時間と共に値が変化する可能性があり, また値を利用する利用者の環境毎に値が変化する.

OWASP Risk Rating Methodology

OWASP Risk Rating Methodology とは, OWASP が作成した脆弱性における評価指標である. これは脆弱性の発見における重要性だけでなく, ビジネスに与える影響度を考えることも同様に重要であるという考えのもとに作成されたものである.

基本的な算出の流れとしては, 脅威を与えるものを起点として, 攻撃手法を分類し, これを元にセキュリティ上の弱点を洗い出し, 制御方法を考え, 技術的な影響を計算し, 最終的にビジネスへの影響度を求めるというものである.

この手法の利用方法を以下の順を追って説明する.

1. リスクの特定
2. リスクの生じやすさの推定
3. リスクの影響の推定
4. リスクの深刻度の決定
5. リスクの修正方法の決定
6. リスク評価モデルの構築

リスクの特定　最初に評価するリスクを特定するために以下のようなことに関する情報を収集する.

- 脅威
- 攻撃
- 脆弱性

- ビジネスに悪用された場合の影響

リスクの生じやすさの推定

ここでは大雑把にリスクの生じやすさを決定する.

リスクの生じやすさを決定する要素として大きく以下の 2 つに分ける.

1. 脅威
2. 脆弱性

1 つ目の脅威については, 以下の 4 つに細分化する.

- 技術レベル
- 動機
- 機会
- 大きさ

技術レベル　脅威の技術レベルを表す. ペネトレーションスキル, ネットワークやセキュリティスキル, 発展的なコンピュータスキル, ある程度のスキル, 全くスキルを持たないの 5 段階に分類し, 点数はそれぞれ, 9, 6, 5, 3, 1 となっている.

動機　脅威が攻撃により得られる報酬を表す. 低いもしくは全く無い, 可能性としてあり得る, 高い, の 3 つの段階に分類し, 点数はそれぞれ, 1, 4, 9 となっている.

機会　脅威が脆弱性を悪用するにはどのようなものと機会が必要を表す. フルアクセスもしくは高価なもの, 特別なアクセスもしくはもの, あるアクセスもしくはもの, アクセスまたはものを必要としない, の 4 つに分類し, 点数はそれぞれ, 0, 4, 7, 9 となっている.

大きさ　脅威のグループの対象の大きさを表す. 開発者, システム管理者, イントラネットユーザ, パートナー, 認証ユーザ, 匿名インターネット, の 6 つに分類し, 点数はそれぞれ, 2, 2, 4, 5, 6, 9 となっている.

2 つ目の脆弱性については, 以下の 4 つに細分化する.

- 発見の容易性
- 悪用の容易性
- 認知性
- 侵入検知

発見の容易性　脅威による脆弱性の発見の容易性を表す. 現実的に不可能, 困難, 容易, 自動化ツールで可能, の 4 つの段階に分類し, 点数はそれぞれ, 1, 3, 7, 9 となっている.

悪用の容易性　脅威による脆弱性の悪用の容易性を表す. 理論上, 困難, 容易, 自動化ツールで可能, の 4 つの段階に分類し, 点数はそれぞれ, 1, 3, 5, 9 となっている.

認知性　脅威がどの程度脆弱性についてよく知っているかを表す. 未知, 密かに, 明白, 公知, の 4 つの段階に分類し, 点数はそれぞれ, 1, 4, 6, 9 となっている.

侵入検知　どの程度悪用が検知されるかを表す. 検知が, アプリケーション中, ログとレビュー, レビューなしのログ, ログもない, の 4 つの段階に分類し, 点数はそれぞれ, 1, 3, 8, 9 となっている.

リスクの影響の推定　技術的影響とビジネス的影響の 2 つの観点からの影響どを表す.
技術的影響については以下の通りとなる.

- 機密性の損失
- 完全性の損失
- 可用性の損失
- 責任追跡性の損失

機密性の損失　どの程度機密性に影響を与えるかを表し, 点数はそれぞれ, 2, 6, 6, 7, 9 となっている.

完全性の損失　どの程度完全性に影響を与えるかを表し, 点数はそれぞれ, 1, 3, 5, 7, 9 となっている.

可用性の損失　どの程度可用性に影響を与えるかを表し, 点数はそれぞれ, 1, 5, 5, 7, 9 となっている.

責任追跡性の損失　どの程度責任追跡性に影響を与えるかを表し, 点数はそれぞれ, 1, 7, 9 となっている.
ビジネス的影響とは以下の通りとなる.

- 財政的被害

- 評判被害
- 非コンプライアンス
- プライバシー違反

財政的被害　財政的被害がどの程度悪用されるかを表す. 点数はそれぞれ, 1, 3, 7, 9 となっている.

評判被害　悪用がビジネスを害する評判を引き起こすかを表す. 点数はそれぞれ, 1, 4, 5, 9 となっている.

非コンプライアンス　どの程度コンプライアンス違反につながるかを表す. 点数はそれぞれ, 2, 5, 7 となっている.

プライバシー違反　どの程度個人情報を漏らすかを表す. 点数はそれぞれ, 3, 5, 7, 9 となっている.

リスクの深刻度の決定　リスクの深刻度の決定方法は非公式手法と繰り返し手法 2 通りある.

非公式手法　非公式手法は調査者が結果を大きく制御するような要素をレビューして答えを出すものである. しかし, 初見では明らかでなかった側面も発見し得るので, 繰り返し評価するのであれば公式な方法を取る必要がある.

繰り返し手法　リスクの影響の推定で脅威要因と脆弱性要因のそれぞれで各要素の点数を求めたが, 2 つの要因の全体の要素の点数の平均を算出する. この値が起こりやすさの点数となる. リスクの生じやすさの推定で求めた技術的影響とビジネス的影響のそれぞれについて, 各要素で点数を求めたがそれらの平均を算出する. これらの値がそれぞれ, 技術的影響の点数, ビジネス的影響の点数となる.

深刻度の決定　上記の手法で求めた点数を表 1.4 より深刻度を決定する.

ただし, ビジネスに影響を与える良質な情報があれば技術的影響の点数ではなく, ビジネス的影響の点数を使用する.

リスクの修正方法の決定　アプリケーションのリスクの分類後, 修理すべきものの優先度付けをする.

リスク評価モデルの構築　リスク評価モデルのカスタマイズをする.

- 要因の追加
- オプションのカスタマイズ
- 要因の重み付け

要因の追加　特定の企業にとって重要な要因を追加することができる.

オプションのカスタマイズ　各要因に関連するオプションをカスタマイズする. カスタムした値は, 専門家によるモデルのスコアと比較することで, 正当性を確認すると良い.

要因の重み付け　各要素について均等に重み付けをしていたが, 特定のビジネスによって重要な要因には重みをつけて計算することができる.

表 1.3: 環境値のパラメータ

環境評価基準	評価結果	値
CD	なし	0.0
	軽微	0.1
	中程度	0.3
	重大	0.4
	壊滅的	0.5
	未評価	0
TD	なし	0.00
	小規模	0.25
	中規模	0.75
	大規模	1.00
	未評価	1.00
CR	低	0.5
	中	1.0
	高	1.51
	未評価	1.00
IR	低	0.5
	中	1.0
	高	1.51
	未評価	1.00
AR	低	0.5
	中	1.0
	高	1.51
	未評価	1.00

表 1.4: 全体のリスク深刻度

	可能性			
		低	中	高
影響	高	中	高	壊滅的
	中	低	中	高
	低	確認	低	中

Chapter 2　関連研究

2.1　XSS 攻撃の静的解析

PHP 言語による Web アプリケーションにおける XSS 攻撃の静的解析の研究の特徴に関する比較は以下の表 2.1 の通りとなる. この他の既存研究は Pixy [16, 17] に基づいて安全なバリデーション方法を提案するものである, もしくは, RIPS [14, 15] の中で採用されているものであるものであるため, 表から割愛している. ただし, Pixy と RIPS に関しては論文の内容と合わせて実際に検証した結果であるが, XSSDM [19] に関しては論文の内容を基に作成したものである.

	Not OOP	OOP	Dynamic OOP	Refered	Stored	DOM-based
Pixy	Able	Sometimes	Disable	Able	Disable	Disable
RIPS	Able	Sometimes	Disable	Able	Sometimes	Disable
XSSDM	Able	Disable	Able	Able	Able	Disable

表 2.1: PHP 言語における XSS の自動検知の既存研究の比較

Pixy については, XSS 攻撃については Reflected XSS のみ, オブジェクト指向については静的な関数の呼び出しの時, かつ静的な呼び出しの階層が浅い時のみ検出可能であった. また, Pixy は PHP4.x までしか対応しておらず, PHP5.x 以降の記述では一部 syntax error と判定されるものがある. 特に PHP4.x から PHP5.x ではオブジェクト指向に対して大幅なアップグレードを行っているため実際に解析するためには PHP4.x の文法に書き換える必要がある. また, Pixy では解析するファイル毎にファイル名を 1 つ 1 つ指定しなければならず, 複数の PHP ファイルにまたがる大規模な Web アプリケーションの静的解析には手間のかかるものである. 現在は公開されていたツールは使用することができない.

RIPS については, XSS 攻撃については実際に検証したところ Reflected XSS のみの検出であったが, 論文中で参照している function and file summaries という手法を提案している論文中では Stored XSS も検出したという結果を報告している. オブジェクト指向については PHP Object Injection(POI) 攻撃と呼ばれる攻撃に対する検出は可能であった. ただし, これは静的な呼び出しによるものだけであり, 動的な呼び出しについては検出することができない

2.1.1 Pixy

Pixy とは XSS 攻撃を静的に検知するための最初のオープンソースである. これは脆弱性検知のために flow-sensitive で interprocedural かつ context-sensitive なデータフロー分析を用いたものである. より正確性の高い分析を行うためにフロントエンドとバックエンドでの大きく 2 段階の処理を行う. 以下これらについて詳細に述べる.

フロントエンド

静的分析を行うために最初にソースコードを解析し, 静的解析が行いやすいように linearization を行う必要がある. そのために, 最初に Parse Tree を作成し, 次にそれらを基に P-Tac という TAC [20] を基にした表現で表わし複雑な表現を単純化する. P-Tac における制御フローグラフ (CFG) の種類についての表は表 2.2 の通りとなる.

Simple Assignment	{ var } = { place }
Unary Assignment	{ var } = { op } { place }
Binary Assignment	{ var } = { place } { op } { place }
Array Assignment	{ var } = array()
Reference Assignment	{ var } &= { var }
Unset	unset({ var })
Global	global { var }
Call Preparation	A call node's predecessor.
Call	Represents a function call.
Call Return	A call node's successor.

表 2.2: P-Tac における CFG ノードの主な種類

P-Tac で Parse Tree を表現するに当たり, 以下の 3 つの操作を行う.

- 様々なループの種類を if と goes だけで表現する
- グローバルな範囲にあるコードはメイン関数に移動する
- 定数は define 関数で表現する

上記の図中の place は変数, 定数, 文字列のいれずかを表わし, function call は call preparation ノード, actual call ノード, call return ノードの 3 つの CFG ノードで表わされる. 定義が見つからない関数呼び出しは unkown function に置き換えられる.

バックエンド

バックエンドでは, LITERAL 分析, ALIAS 分析, TAINT 分析の大きく 3 つに分かれる. 以下, これらについてそれぞれ説明する.

LITERAL 分析　LITERAL 分析の特徴は, Carrier lattice と Transfer 関数を定義して, 各プログラム地点での変数や定数の値を分析することである. 以下, 詳細を述べる.

Carrier lattice の定義 Carrier lattice を定義するメリットととして, すべての変数や定数の対応付けを提供すること, また, あらゆる文字列への対応付けを表わせる点が挙げられる.

以下の図は LITERAL 分析における lattice の一部分の例である. プログラム分岐ごとに枝が分かれそれらに対応する値が変数に割り振られる. それ故, lattice 全体の頂点はすべての変数, 定数が空の状態であり Ω が割り当てられる.

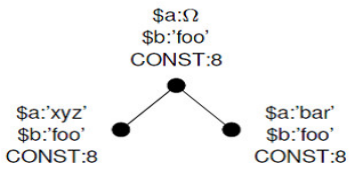

図 2.1: LITERAL 分析の lattice の一部 (引用)

遷移関数の定義 carrier lattice の定義後, 各 CFG ノードを結びつけるための関数を定義する. 対応する CFG ノードを制御フローが通過するときに分析対象の情報がどのように影響を与えるものかを定義するものである.

ALIAS 分析への依存 各プログラムの地点での変数や定数の値はエイリアスのよる依存がある. そのため ALIAS 分析後, それらの情報を基に LITERAL 分析を拡張する必要がある.

ALIAS 分析　静的分析では, if 文などの状況依存に応じた解析に対して, 以下の図のような手法を取る.

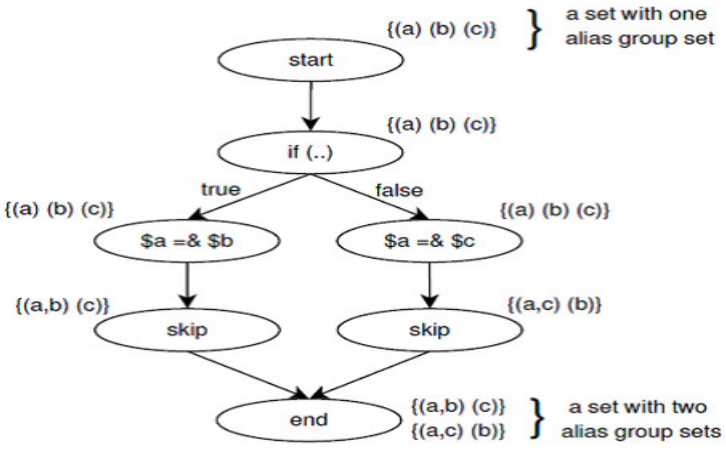

図 2.2: ALIAS 分析の lattice の要素 (引用)

始めは各変数は独立した集合を表わすが, 条件分岐ごとに参照されると集合を合わせる. それらの条件ごとに和集合という形で表現するものとなる.

これらの情報を基に再び LITERAL 分析を拡張しエイリアスの情報を含めた分析を行う.

TAINT 分析　TAINT 分析における Carrir Lattice と遷移関数の定義の 2 つの観点から説明する.

Carrier Lattice の定義　TAINT 分析では保守的なアプローチを採用されている. 変数では脆弱であるかもしれないという情報を保持して起き, サニタイジングされていれば脆弱でないと判断する. 文字列でない配列の要素については, 脆弱であると対応づける. また, 変数が途中で配列の初期化宣言された場合はそれまでの変数の内容をすべて消去して Clear Array フラグ (CA フラグ) を立てて分析を行う.

遷移関数の定義　LITERAL 分析における遷移関数の定義における分析と同様であるが, 大きな違いは CA フラグを考慮しながら分析を行うという点で異なる.

2.1.2　関数とファイルのサマリー

Xie らにより提案された手法で, スクリプト言語における脆弱性の分析手法を提案した. [13] これは関数とファイルのサマリーという手法を用いている. 論文中では SQL インジェクションにおける対処法で紹介されているが, 同様の手法で XSS 攻撃にも対処可能だと主張している. 概要は以下の流れになる.

抽象構文木 (AST) を作成　PHP のソースコードを解析して AST を作成する. 解析の際はユーザ定義関数を保存しておき, メイン関数から解析を始める.

制御フローグラフ (CFG) の作成　作成した AST を基に CFG を作成する. CFG の各ノードは基本ブロックを表わし, 各エッジは基本ブロック間の関係を表わす.

ブロックサマリーの作成　各基本ブロックを symbolic 実行をすることにより, 各 basic block がグローバル環境に与える影響を集めてブロックサマリーを作成する.

関数サマリーの作成　作成したブロックサマリーをまとめて関数サマリーを作成する

この全体の中で重要となる基本ブロックのシミュレーション, Intraprocedural 分析, Interprocedural 分析について詳細に述べる.

Basic Blocks のシミュレーション

PHP 言語のモデル化　PHP のようなスクリプト言語は動的に型付けがなされる．このような挙動に対して Xie らは, 図 2.3 のような定義を行った．

$$
\begin{aligned}
\text{Type } (\tau) &::= \text{str} \mid \text{bool} \mid \text{int} \mid \top \\
\text{Const } (c) &::= \text{string} \mid \text{int} \mid \text{true} \mid \text{false} \mid \text{null} \\
\text{L-val } (lv) &::= x \mid \texttt{Arg\#i} \mid lv[e] \\
\text{Expr } (e) &::= c \mid lv \mid e \text{ binop } e \mid \text{unop } e \mid (\tau)e \\
\text{Stmt } (S) &::= lv \leftarrow e \mid lv \leftarrow f(e_1, \ldots, e_n) \\
&\quad \mid \text{return } e \mid \text{exit} \mid \text{include } e \\
\\
\text{binop} &\in \{+, -, \text{concat}, ==, !=, <, >, \ldots\} \\
\text{unop} &\in \{-, \neg\}
\end{aligned}
$$

図 2.3: PHP 言語のモデル化 (引用)

PHP の値の型を string, boolean, integer の 3 種類に加えて $\perp$ というパラメータの入力などで静的な型付けがなされていないものを表わすシンボルを定義している．

L 値とは, 変数や関数に加えて, 配列やハッシュテーブルに操作をする添え字による操作を含むものである．式とは定数や L 値, 単項演算子, 二項演算子, 型のキャストを表わすものである．宣言とは, 割り当て, 関数呼び出し, return, exit, include を表わすものである．

シミュレーション中の値と状態の表現　各基本ブロックにおいてシンボリック実行をするが, このときの値と状態の表現について, 以下の図のような定義を行った．

Value Representation

$$
\begin{aligned}
\text{Loc } (l) &::= x \mid l[\text{string}] \mid l[\top] \\
\text{Init-Values } (o) &::= l_0 \\
\text{Segment } (\beta) &::= \text{string} \mid \text{contains}(\sigma) \mid o \mid \perp \\
\text{String } (s) &::= \langle \beta_1, \ldots, \beta_n \rangle \\
\text{Boolean } (b) &::= \text{true} \mid \text{false} \mid \text{untaint}(\sigma_0, \sigma_1) \\
\text{Loc-set}(\sigma) &::= \{l_1, \ldots, l_n\} \\
\text{Integer } (i) &::= k \\
\text{Value } (v) &::= s \mid b \mid i \mid o \mid \top
\end{aligned}
$$

Simulation State

$$\text{State } (\Gamma) : \text{Loc} \rightarrow \text{Value}$$

図 2.4: シミュレーションにおける値と状態の表現の定義 (引用)

上記のような String の定義を取ることにより, string による自動的な定数の取得や,

ある変数へのユーザによる入力が別の変数に代入される様子, また関数サマリーに基づく返り値を追っていくことで後述する Internalprocedural 分析が可能となる.

シミュレーション中の L 値とメモリ配置の扱いについて　l 値は非定数キーによりハッシュにアクセスすることがあり, 同じ L 値でもハッシュ関数とそのキーによっては異なるメモリー配置となる可能性があり, L 値はシミュレーションでメモリ配置を表わすには適さないため以下のような特別な定義を与えた.

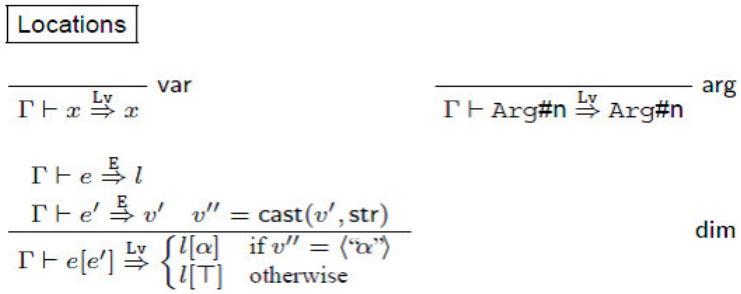

図 2.5: シミュレーションにおける値と状態の表現の定義 (引用)

var は具体的な変数名によって変数のメモリ配置を定義するもの, arg は具体的な関数名によって関数のメモリ配置を定義するもの, dim はハッシュ関数においてキーによって値を解決するために定義したものとなる.

式におけるシミュレーション法　上記による表現を基にして評価を行う. この時の式におけるシミュレーション法は以下の通りとなる.

上記の通り, キャストと評価のルールについて定義したものとなっている. キャストのルールについては, boolean における true が string の”1”への対応などの異なる型への変換の対応関係を定めたものとなっている.

評価のルールについては, L 値, 文字列連結, boolean の干渉の 3 つについてルールを定めている. L 値については, 最初に L 値をメモリ配置を行い評価中に配置された場所を調べるということを表わしている. 文字列連結については, 最初に string 型にキャストを行い, それらを連結することを表わしている. boolean の干渉については, もし変数がサニタイズされていれば, untaint について false を表わす boolean となることを表わす.

宣言におけるシミュレーション法　宣言についてシミュレーションをするために, 割り当て, 関数呼び出し, return, exit, include についてモデル化を行う.

割り当てのモデル化　割り当てのモデル化について, 以下の図のようになる.

上段の左側についてはメモリ配置, 右側については評価を表わし, 下段はそれらを基に新たに生成したものとなっている.

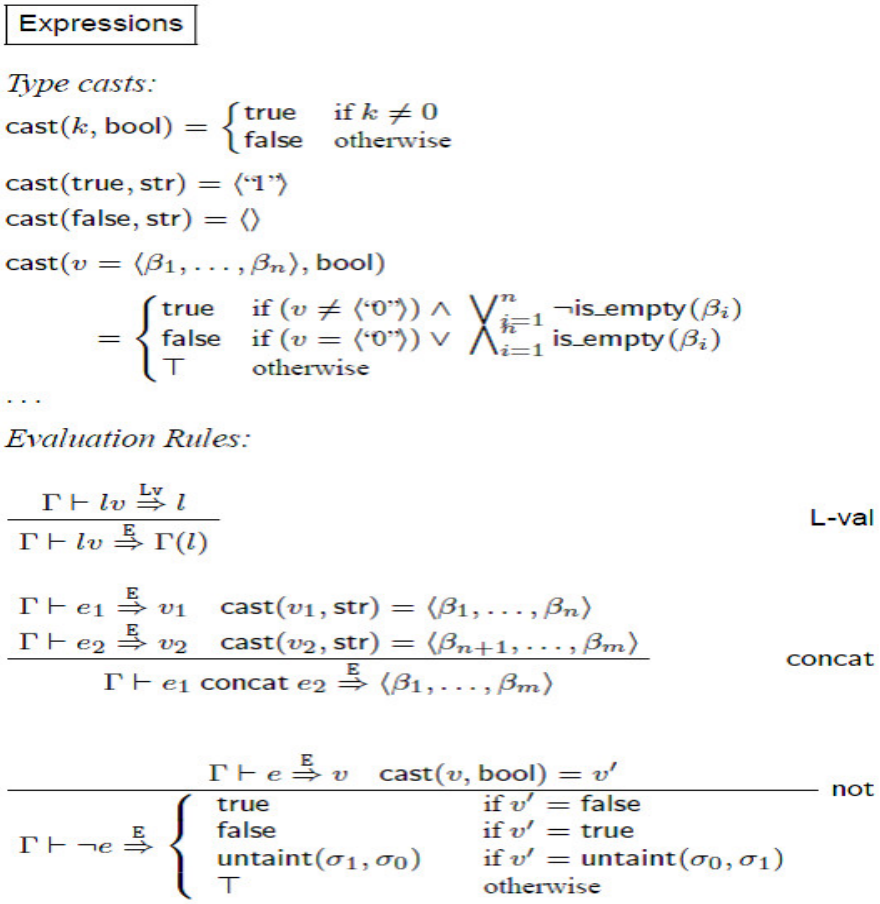

図 2.6: 式におけるシミュレーション法 (引用)

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

図 2.7: 割り当てのモデル化 (引用)

関数呼び出しのモデル化 関数呼び出しのモデル化について, 以下の通りとなる.

基本的な処理は割り当てと同様である.

ただし, 以前の宣言の出力環境が呼び出される時に, 割り当ての連鎖と関数呼び出しがシミュレーションされる. 割り当ての連鎖のモデル化については以下の通りとなる.

return のモデル化 return は制御フローを終了させるもので, この時の値を関数サマリーに使用する.

exit のモデル化 exit は制御フローを終了させるもので, 分析中の block を exit block とする.

$$\frac{\Gamma \vdash lv \overset{\mathrm{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\mathrm{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\mathrm{E}}{\Rightarrow} v_n}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\mathrm{S}}{\Rightarrow} \Gamma[l \mapsto f(v_1, \ldots, v_n)]}\ \mathsf{fun}$$

図 2.8: 関数呼び出しのモデル化 (引用)

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

図 2.9: 割り当ての連鎖のモデル化 (引用)

include のモデル化 実際のコードの実行時において include では動的にファイルを読み込み, 読み込み地点で関数の定義や変数が呼び出される. シミュレーションでは, include されたファイルを最初に解析しておき, 新たな関数の定義として追加する.

ブロックサマリー　CFG のブロックからブロックサマリーを作成する. ブロックサマリーは, Error set, Definitions, Value flow, Termination predicate, Return value, Untaint set の 6 つの要素からなるタプルを表わし, これらの計算を行う. 以下, これらについて説明する.

Error set 現在の Block に入る前にサニタイズの必要のある入力変数の集合

Definitions 現在の Block 中のメモリ配置の集合

Value flow ある変数の値が別の変数の値に代入されたときの流れを表わす, 変数の前後のペアの集合を表わす.

Termination predicate 現在の Block が exit かプログラムを終了させる関数が含まれていれば true

Return value 返り値の集合

Untaint set 次の Block が呼び出される前に作られるもので, サニタイズされているメモリ配置の集合を表わし, キャストや正規表現による一致等による方法で判断する.

Intraprocedural 分析

Basic Blocks を基に, 関数毎に関数サマリーを求める. 関数サマリーは Error set, Return set, Sanitized set, Program exit の 4 つの要素からなるもので, これらを計算する.

Error set データベースへ挿入する可能性のある変数やパラメータ, ハッシュ関数へのアクセス等の要素の集合. 後方到達性分析により計算する.

Return set 関数の返り値になりうるパラメータやグローバル変数の集合. 前方到達性分析により計算する.

Sanitized set 関数の終わりにサニタイズされているパラメータやグローバル変数の集合. 前方到達分析で計算する.

Program exit 全てのパスでプログラムを終了する関数が存在するかの真偽値. すべての継承するものがない CFG の Block について T や R があるかの判定により計算する.

Interprocedural 分析

分析中に他のユーザ定義関数が現れた時, Intraprocedural 分析の結果を基にして, Pre-conditions, Exit conditions, Post-conditions, Return value のの要素からなる集合を求め, これらをそれぞれ計算する.

Pre-conditions 関数が呼び出される前にサニタイズの必要のあるパラメータやグローバル変数の集合.

Exit condition 関数がプログラムを終了させる関数であれば, 継承する関数の呼び出しや現在の Block に続く Block は全て消去し, 現在の Block を Exit Block とする.

Post-conditions 無条件に入力パラメータやグローバル変数をサニタイズすれば安全とみなし, そうでないときは両方の可能性をセットで記録しておく.

Return value 関数の返り値が boolean 型のときで条件によってサニタイズされるとき, 以下のようにモデル化される.

$$\frac{\begin{array}{l}\Gamma \vdash lv \overset{\mathtt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\mathtt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\mathtt{E}}{\Rightarrow} v_n \\ \mathsf{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \\ \mathcal{S} = (\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1) \quad \sigma_* = \sigma_0 \cap \sigma_1 \\ \sigma_0' = \mathsf{subst}_{\bar{v}}(\sigma_0 - \sigma_*) \quad \sigma_1' = \mathsf{subst}_{\bar{v}}(\sigma_1 - \sigma_*)\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{untaint}(\sigma_0', \sigma_1')]}$$

図 2.10: 返り値が boolean 型におけるモデル化 (引用)

関数の返り値が string 型のときは, 関数サマリーを使用して, 以下のようにモデル化される.

$$\frac{\Gamma \vdash lv \overset{\mathtt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\mathtt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\mathtt{E}}{\Rightarrow} v_n \quad \mathsf{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \quad \sigma' = \mathsf{subst}_{\bar{v}}(\mathcal{R})}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{contains}(\sigma')]}$$

図 2.11: 返り値が string 型におけるモデル化 (引用)

2.1.3　スクリプト言語におけるオブジェクト指向の静的解析

スクリプト言語の脆弱性の静的解析でも一部のオブジェクト指向を対象に試みたものも存在する. Automated POP Chain Generation と呼ばれる PHP Object Injection (POI) 攻撃を対象に静的なオブジェクト指向による関数についての脆弱性に対して静的解析を可能とするものである. これは PHP 言語を対象とした静的解析手法である.

PHP Object Injection 攻撃

PHP Object Injection 攻撃とは Code Reuse 攻撃の 1 つである. この PHP 言語における Code Reuse とは 2009 年に指摘された攻撃で比較的新しいものとなっている. [21, 22]Code Reuse 攻撃はサーバサイド側で書かれた既存のコードを利用するので攻撃コードを攻撃者が Injection する必要がないのが特徴である. 攻撃の際には, 脆弱な gadget chain を見つけ, ユーザによる入力から脆弱性の箇所まで Chain を連鎖して辿ることで行う. PHP には magic method と呼ばれる以下の 15 種類の特定の条件化で呼び出される関数が存在する.

- __contruct()
- __destruct()
- __call()
- __callStatic()
- __get()
- __set()
- __isset()
- __unset()
- __sleep()
- __wakeup()

- __toString()
- __invoke()
- __set_state()
- __clone()
- __debuginfo()

これらの関数を呼び出す関数中で引数がユーザによる入力であった場合に脆弱性が発生する可能性があり, magic method を呼び出す関数の例として unserialize 関数が存在する.

PHP Object Injection 攻撃の静的解析

最初に各 PHP のソースコードのファイル毎に抽象構文木を作成する. そしてこれらの分析を行い, 関数やクラスの宣言は subtree に, 残りの tree には各ファイル名を割り振る. 次に作成した抽象構文木から制御フローグラフを作成する. この時に, 前セクションで説明したブロックと関数サマリーの手法を用いる. 抽象構文木から制御フローグラフを作成するに当たって, 関連付けられた基本ブロックに分けられる. この基本ブロックを作成するときに, 現在のブロックを新しいブロックが繋がる時に D データフロー分析が始まり, その後, 現在の基本ブロックがシミュレーションされる. データフロー分析についての詳細は後述する. シミュレーション中に sensitive sink に当たったら backwards-directed taint 分析が始まる. また, シミュレーション中にユーザによる定義関数に当たった場合は処理を中断して, その関数の抽象構文木を分析して制御フローフラフを作成する. この際, パラメータやグローバル変数に当たれば関数サマリーに格納する. その後, 中断した分析を再開する. 全てのファイルの変形が終わったら, 各ファイル毎にファイルサマリーを作る.

データフロー分析

基本ブロックの抽象構文木に基づいて, 全てのデータの割り当てを loc :=< $assigneddata$ > の形式でメモリに割り当てる. この assigned data はさらに以下の３つの形式に変換する.

- $VALUE$
- $VARIABLE$
- $ARRAYFETCH$

```
function getPost() {
    if (isset($_POST['hoge'])) {
        $fuga = $_POST['hoge'];
        $output = $fuga;
    }
    else {
        $fuga = null;
        $output = $fuga;
    }

    return $output;
}

$piyo = getPost();
```

図 2.12: 第一段階

```
function getPost() {

        $fuga = $_POST['hoge'];
        $output = $fuga;


        $fuga = null;
        $output = $fuga;


    return $output;
}

$piyo = getPost();
```

図 2.13: 第二段階

```
function getPost() {

        $output = $_POST['hoge'];


        $output = null;


    return $output;
}

$piyo = getPost();
```

図 2.14: 第三段階

```
function getPost() {




    return ($_POST['hoge'] | null);
}

$piyo = getPost();
```

図 2.15: 第四段階

図 2.16: ブロックと関数のサマリーによる手法解析

$VALUE$ は string または integer 型の値, $VARIABLE$ は変数, ARRAYFETCH は多次元配列を含む配列を表わす.

VARIABLE, ARRAYFETCH は共に loc に割り振られるが, ARRAYFETCH は扱いの際に処理が複雑になる. この問題に対して木構造である ARRAYWRITE というラッパーシンボルを導入する. この木のエッジは次元を表わし, 葉は割り当てられたデータを表わし, エッジと葉はデータシンボルに入れられる. これらの割り当て処理が終了したら, ブロックサマリーにインデックスされる.

後方脆弱性分析によるブロックと関数のサマリーの作成

後方脆弱性分析によるブロックと関数のサマリーの作成について, 以下の図 2.16 のような 4 段階により分析を行う.

最初にブロックサマリーを作成する. この処理を大きく 4 つの段階に分かれる. 第一段階について, ユーザ定義関数とそれ以外の部分について分ける. 第二段階では, 条

件分岐による処理について, 条件条件は考慮せず各分岐先内での処理に着目する. 第三段階では, それぞれの処理についてブロックサマリーを作成する. 第四段階では, それらのブロックサマリーの結果を基に関数サマリーを作成する. このとき, 関数の返り値に各分岐先での処理について, それぞれの結果をセットにして値を返すものとする.

前方向データ伝搬

通常の後方脆弱性分析に加えて, もし, unserialize 関数で呼び出される引数がユーザによる入力であった場合, unseriaze 関数で呼び出されたオブジェクトにはフラグが立てられ, 以降このフラグが立てられたオブジェクトに関わるプロパティは脆弱なものとして扱われる.

Inter-procedural 分析

オブジェクト指向の静的解析に対する問題を解決するために Dahse らは OBJECT シンボルを導入した. 静的な関数の呼び出しについてはクラスと対応付けるのは容易であり, self や parent による関数の呼び出しにおけるクラスの解決もクラスの階層から解決することで可能である. レシーバに対するクラス名との結びつけについて, 分析中の基本ブロックにおけるオブジェクトキャッシュを用いて解決を図り, クラス名をキャッシュされた OBJECT シンボルによって解決される. しかし, オブジェクトキャッシュは分析中の制御フローグラフにしか存在しないため, 動的なオブジェクト指向の呼び出し等には対応できない. 例えば, レシーバが関数の引数として渡されたとき, または関数内でレシーバが global により宣言された場合や$GLOBALS を用いて宣言された時について説明する. 前者の問題について, 全てのクラスの定義における処理を特定する関数を含むすべての関数について探索する. もし, 関数名が一意に特定できれば対応するレシーバと関数が結びつけられる. 一意に特定できない場合については引数の数に着目して一意に特定できれば対応するレシーバと関数が結びつけられる. これらの処理でも一意に特定できない場合についての図を以下に示す.

すべての可能性のあるクラスの関数の関数サマリーを作成中のサマリーに含める. もし, 候補の中に sensitive sink に結びつくものがあれば sensitive プロパティとして判定する.

レシーバが関数内でグローバル変数として宣言された場合について, 検査対象全体の関数内におけるグローバル変数をインデックスしておく. 動的なグローバル変数の呼び出しであった場合にはレシーバが関数の引数として呼ばれた場合の処理と同様の処理により処理をし直す.

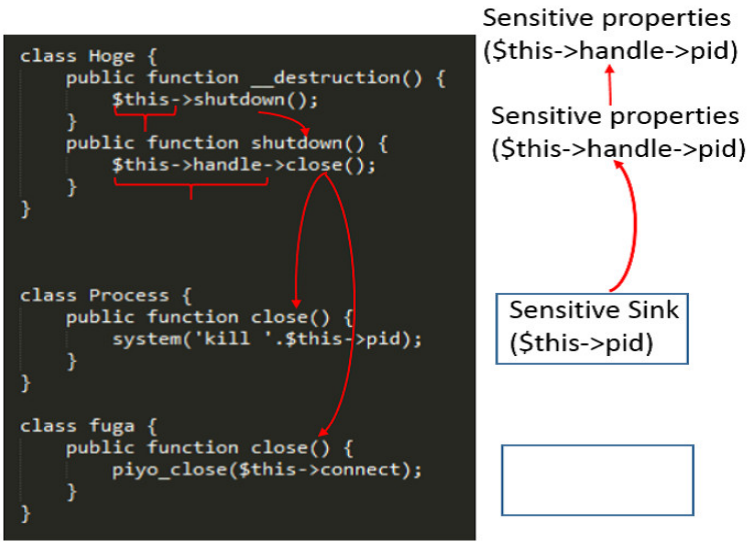

図 2.17: Inter-procedural 分析

2.1.4　XSSDM

XSSDM では XSS 攻撃の解析を HTML 文脈に基づいて脆弱性の判定を行い, これらは以下 6 つの規則に基づいて行われる.

規則 1 HTML タグ全体を含む, もしくは文字列中で HTML タグを持たないとき

規則 2 a, style, script 等の特殊タグで始まり, ”, ’, もしくは引用符を含まないもので終わるとき

規則 3 特殊タグ以外の HTML タグで始まり, ”, ’, もしくは引用符を含まないもので終わるとき

規則 4 HTML タグで始まり, イベントハンドラを含み, ”, ’, もしくは引用符を含まないもので終わるとき

規則 5 HTML タグで始まり, イベントハンドラを含み, ”, ’, もしくは引用符を含まないもの以外で終わるとき

規則 6 ”＜”の文字列だけを含むとき

XSSDM では最初に脆弱性のある可能性のある出力関数中で使われている変数宣言部分のリスト化を行う. そして上記の規則を基にそれらの宣言の HTML の文脈判定を行う. そして, 文脈に応じた適切なサニタイズ関数が使われているかの判定という段階を経て XSS 攻撃の脆弱性判定を行う手法を取る.

2.1.5　既存研究まとめと問題点

Huang らは, 静的解析と動的解析を組み合わせた手法を提案し, XSS 攻撃や SQL インジェクション, その他インジェクション攻撃を対象とした. [8]

Xie らは, 静的解析により, SQL インジェクション攻撃を対象としており, 論文中で同様の手法で XSS 攻撃やその他インジェクション攻撃の検出も可能だと主張する. [13]

Jovanovic らは, 静的解析により, XSS 攻撃と SQL インジェクション攻撃を検出する手法を提案した. 筆者は論文中で Pixy と名付けている. [16, 17]

Wassermann 氏は, 2007 年, 静的解析により SQL インジェクション攻撃の検出のための手法を提案している. 翌年, 2008 年には静的解析により XSS 攻撃の検出のための手法を提案している. [46] 同年, Balzalotti らは, 静的解析と動的解析を組み合わせて, XSS 攻撃と SQL インジェクション攻撃を対象とするものを発表した. これを論文中では Saner と名付けられ, Pixy を基にして, 適切なバリデーションをするための関数を提案するというものである. [5] 2010 年, Kneuss らによって Phantm という変数のタイプの不一致に着目した脆弱性検出の手法を提案した. [9] 翌年, Sooel らによって SAFEPHP というセマンティクスの観点から脆弱性の検出の手法を提案し, これにより DoS 攻撃などに利用されることを防止する手法を提案した. [10]

2014 年に Dahse らによって, 静的解析により PHP Object Injection(POI) 攻撃を対象として一部のオブジェクト指向により実装されたコードの解析を可能とする手法を提案している. これは, XSS 攻撃や SQL インジェクション攻撃, その他様々な脆弱性の検出も可能である. [14]

2015 年に Gupta らによって, 静的解析により XSS 攻撃の検出のための手法を提案している. これは HTML コンテキストに着目して, いくつかのルールを定めこれを基に分類を行って検出に活用するものである. [19]

XSS 攻撃の自動検知のおける問題点

XSS 攻撃に関する研究は数多くなされてきている. しかし, Hydara らによる調査により, 71.3 %の論文が Reflected XSS や Stored XSS, DOM based XSS のどの種類の XSS 攻撃に対処したかの明記がないという. [3] また, DOM based XSS について触れられているものはあまり存在しない. [23, 24, 25, 26, 38]

DOM based XSS は, DOM から悪意のあるデータを読み込むものであり, サーバサイドの処理には必ずしも悪意のあるデータが現れないことがあり, 文法の観点からの解析では対策が困難であり, 生成された Web ページのセマンティクスに着目する必要がある. [11]

脆弱性の自動検出におけるオブジェクト指向言語の問題点

オブジェクト指向言語における静的解析において, Smaragdakis らによって型付き言語を対象にオブジェクト指向を対象に静的解析を行う手法を提案された. [18] これはオブジェクトそのものではなく, コード中における型の文脈に着目して解析を行ったものであるが, 型の概念の弱い PHP 言語には適用することができず, 十分に研究がなされていない. オブジェクト指向における特徴のうちポリモーフィズムが問題を難しくしている原因だと考えられる. オブジェクト指向言語では, 同じメッセージ名でもそのメッセージを実行するレシーバによってメッセージに結びつくメソッドが変化し, 動的に決定する, そのため, 脆弱性の自動検出における静的な解析においては, 解析地点でメッセージと結びつくメソッドが決定していないため解析するのが困難である. [14]

2.2　評価方法

従来の既存研究では手法の評価手法について, 以下の 5 つの既存研究の評価方法を取り上げる.

- Jovanovic らによる Pixy
- Dahse らによる RIPS
- Wassermann らによる手法
- Xie らによる手法
- Gupta らによる XSSDM

2.2.1　Jovanovic らによる Pixy

Jovanovic らによる Pixy[42, 43] ではファイルの総行数, source となる関数中で引数として変数を用いている数, そのうち実際に脆弱性だった source の数, 誤検知の数の項目を既知の脆弱性と新たに見つかった脆弱性に分けて評価している. 検査の対象となる Web アプリケーションは, Jovanovic らが自ら選んだ有名だとする PHP のオープンソースとなる.

- PhpNuke 6.9
- PhpMyAdmin 2.6.0-pl2
- Gallery 1.3.3
- Simple PHP Blog 0.4.5

- Serendipity 0.8.4
- Yapig 0.95b

これらを図 2.3, 図 2.5 のように評価している.

表 2.3: Pixy で発見された既知の脆弱性

Program	File	LOC	Variables	Vulnerabilities	FP's
PhpNuke 6.9	Reviews Module	8409	3113	15	5
	YourAccount Module	9070	3452	9	25
PhpMyAdmin 2.6.0-pl2	select server.lib.php	89	23	9	0
Gallery 1.3.3	search.php	1810	530	2	1
	login.php	1719	488	1	0

表 2.4: Pixy で発見された未知の脆弱性

Program	File	LOC	Variables	Vulnerabilities	FP's
Simple PHP Blog 0.4.5	preview cgi.php	6938	2342	3	5
	preview static cgi.php	6883	2316	4	4
	colors.php	6971	2313	1	6
Serendipity 0.8.4	personal.inc.php	6588	2305	2	1
Yapig 0.95b	view.php	5128	1302	5	0

2.2.2　Dahse らによる RIPS

POP チェーンの自動生成により PHP Object Injection 攻撃の脆弱性を検知する拡張を行った RIPS[44, 45] による評価について, Dahse らは, 2013 年から 2014 年に報告された CVE(Common Vulnerabilities and Exposures: 共通脆弱性識別子) の中から以下の基準に基づいて検査対象のファイルを選定している.

- 対象の脆弱性を複製できるように脆弱のあるソフトウェアのバージョンが Dahse らの評価地点でダウンロード可能なもの

- アプリケーションが 40000 行以上の大規模で, かつ主にオブジェクト指向で実装されたもの
- サードパーティのプラグインや実装を要するコンポーネントを要するものを除いた, アプリケーションがそのままで攻撃可能であるもの

上記の基準を満たす Dahse らが最も有名なアプリケーションとして, 以下の 9 つのオープンソースと PHP Object Injection 攻撃の脆弱性があるとして最初に報告された Piwik を対象に評価を行った.

- Open Web Analytics
- Contao CMS
- CMS Made Simple
- LiveZilla
- Wordpress
- Vanilla Forums
- GLPI
- CubeCart
- Joomla

評価基準は, ファイルの数, 総行数, 測定時間, メモリ使用量, PHP Object Injection 攻撃の脆弱性の数, ガジェット (既存コードを利用して攻撃した時のチェーンを構成する要素のかたまり), ガジェットチェーンの数で評価を行っている.

2.2.3　Wassermann らによる手法

Wassermann ら [46] は有名な PHP の Web アプリケーションを幾つか選び, 3 つの観点から評価を行っている. 1 つ目の評価方法は表 2.6 となる. 文字列分析やポリシーチェックにかかる時間, 消費メモリ, 脆弱性が検出されたかどうかの 2 値で評価を出して, 以下のように計算している.

2 つ目の評価方法は表 2.7 となる. バグの内訳として, 直接的に攻撃者ができる場合を GET, POST, COOKE 変数を使用している場合, 未初期化の場合の 2 通りと間接的に攻撃できる場合でどれだけ数が見つかったかを評価している.

3 つ目の評価方法は, 表 2.8 となる. HTML で記述できるかの可否と XSS 攻撃の脆弱性があるか, またはその詳細についても記述している.

表 2.5: RIPS で発見された未知の脆弱性

Software	Version	Files	LOC	Times [s]	Mem [MB]	POI
Open Web Analytics	1.5.6	463	82013	155	475	0/1
Contao CMS	3.2.4	578	202993	298	1264	19/3
CMS Made Simple	1.11.9	692	135478	567	922	1/1
LiveZilla	5.1.2.0	103	42753	151	342	2/1
Wordpress	3.5.1	425	190800	1138	7640	0/1
Vanilla Forums	2.0.18.5	597	123465	951	6471	2/2
GLPI	0.83.9	1025	347682	676	1632	15/1
CubeCart	5.2.0	846	141404	447	1483	1/1
Joomla	3.0.2	1592	289207	338	1251	2/1
Piwik	0.4.5	750	174314	87	476	1/1
Total		7071	1730109	4808	21956	43/13

2.2.4　Xie らによる手法

Xie らによる評価方法 [47] は, 表 2.9 となる. Xie らが有名だとする 6 つの有名な PHP のオープンソースを対象に, オープンソースの総行数, SQL Injection の脆弱性の数をユーザ入力由来の値を使用してクエリーを組み立てている場合は Bugs として Bugs の数, それ以外の場合は Warn として Warn の数がいくつ見つかったかを評価している.

2.2.5　Gupta らによる XSSDM

Gupta らによる XSSDM[48] の評価手法は, 表 2.10 となる. これは PHP Vulnerability test suite[49] というデータセットを用いて既存研究を評価を行い, Pixy や RIPS, XSSDM の間で False Positive と False Negative を比較している.

2.2.6　既存研究のまとめと問題点

Jovanovic ら, Dahse ら, Wassermann ら, Xie ら, Gupra らによる静的解析の評価方法について取り上げた.

Pixy や RIPS, Wassemann ら, その他多くの既存研究による評価はファイルの規模に対する, 脆弱性の数や誤検知の数がどれだけ新規に, または既知のものが見つかったかを評価していることが多い. 一方, Gupta らによる XSSDM では, PHP Vulnerability test suit という PHP を対象とした脆弱性を含むデータセットを利用している. これは安全な場合の XSS 攻撃の脆弱性のデータセットと安全でない場合の XSS 攻撃の脆弱性のデータセットが含まれるためこれを利用して, False Positive や Flase Negative の測定を行い, 既存研究との比較などを行っている.

表 2.6: Wassermann らの評価

Project name	Time (h:mm:ss)		Memory (MB)	Vulnerability	
	String Analysis	Policy Check		Reported	Present
PHPlist 2.10.2	0:00:01	0:00:01	36	yes	yes
PHProjekt 5.2.0	0:00:36	0:00:39	167	yes	yes
Sendcard 3.2.2	0:00:15	1:01:11	2822	yes	yes
VLBook 1.21	0:00:01	0:00:27	232	yes	yes
Drupal 4.2.0	—	—	—	failed	yes
BASE 1.2.5	0:00:01	0:00:01	33	no	no
FishCart 3.1	0:00:01	0:00:01	39	no	no
SugarSuite 4.2.1	0:00:01	0:00:01	36	no	no
LinPHA 1.3.0	—	—	—	failed	no

表 2.7: Wassermann らの評価 (バグの内訳)

Subject	GPC		Uninit		Indirect
	t	f	t	f	
Claroline 1.5.3	32	43	38	25	42
FishCart 3.1	2	2	30	12	2
GecBBLite 0.1	1	1	0	0	7
PhPetition 0.3.1b	0	0	7	8	7
PhPoll 0.96 beta	5	6	0	0	0
Warp CMS 1.2.1	1	1	22	19	18
Yapig 0.95b	15	13	9	1	14

Pixy や RIPS などの前者の評価方法において, 脆弱性があるか, ないかの 2 値で評価しているが, 実際に脆弱性が現れるかは, Web アプリケーションが利用される環境や, サニタイジングの際にアプリケーションの構造をどの程度維持するかなど様々な要因に依存する. そのため, Web アプリケーションの脆弱性の扱い方に対する戦略に依存するため, 一概に脆弱性だと判定するのは困難である. そのため, 各手法の脆弱の数だと判定した脆弱の定義が不明瞭であり, 違いが生じうる.

また, 各評価方法は実験計測時における見つかった脆弱性の数を検知しているが, オープンソースは日々アップデートされており, 脆弱性も修正されている. そのため, ある研究による手法で新規に脆弱性を見つけることができても, 既存研究で見つけることができていたものがその手法で見つかるかなどの評価はなされていないことが多く, 相対的に既存手法の比較をすることが困難である. また, 脆弱性だと判定した根拠が利

表 2.8: Wassermann らの評価 (HTML Markup の可否と XSS 攻撃脆弱性の有無)

Project name	Allows some HTML mark-up	Has XSS Vulnerability
PHPlist 2.10.2 PHProjekt 5.2.0 Sendcard 3.2.2 VLBook 1.21 Drupal 4.2.0	yes	yes
BASE 1.2.5 FishCart 3.1 SugarSuite 4.2.1 LinPHA 1.3.0	no	no

表 2.9: Wassermann らの評価 (HTML Markup の可否と XSS 攻撃脆弱性の有無)

Application	LOC	Error Msgs	Bugs	FP	Warn
News Pro	6500	8	8	0	8
myBloggie	9200	16	16	0	16
PHP Webthings	38300	20	20	0	20
DCP Portal	121000	39	39	0	39
e107	126000	16	16	0	16
Total		99	99	0	115

用者には分からない, 分かりにくい, もしくは実際には根拠としては薄いことが多い.

評価対象の選定に対しても, 各既存研究がそれぞれ自分の提案手法で効果が現れる知名度の高いと思われるアプリケーションを選定しており, 提案手法間の相対的な比較が困難である.

Gutpta らによる XSSDM の後者の方法では, データセットを用いて, こちらに安全な場合と安全でない場合のデータ, それぞれが含まれている. そのため, 予め研究手法間で脆弱性の有無の判定の基準は統一される. また, 既存手法の間での相対的な比較が容易となる. ただし, 使用されるデータセットは, キャストや引用符で囲っているかなど, ソースコードそのものの観点のみで場合分けをなされたものであり, XSS 攻撃の脆弱性という観点の場合, データセットの脆弱性のパターンに偏りがある.

一方で, CVSS や OWASP Risk Rating Methodology は, ある脆弱性の概念に対して, 自分の環境に応じて, 一般的な評価ができる. 一方でソースコードの解析という観

表 2.10: XSSDM の評価方法

Results	Vulnerabilities Detection Tools		
	Pixy	RIPS	XSSDM
False Positive	34.45%	0%	0%
False Negative	0%	35.88%	0%

点の場合, 危険度の高いある脆弱性につながるパターンは自分のソースコード中での出現頻度は低い, その逆の危険度が低いがソースコード内での出現頻度が高いといった観点での戦略に応じた対策, ソースコード中のある箇所の脆弱性対策にかかる時間など, ソースコードにおける各脆弱性に関しての評価や対策の優先度判定などをすることはできない.

Chapter 3　オブジェクト指向で実装されたスクリプト言語における静的解析

3.1　提案手法

Web アプリケーション, 特にスクリプト言語におけるオブジェクト指向の解析を困難にする原因として, オブジェクト指向において関数の具体的な処理は実行時に動的に決定することが挙げられる. これは関数と結びつくレシーバが実行前には決定せず, 実行時にレシーバのクラスが決定し, その後関数名に一致するそのクラス中に存在する関数が実際の処理内容となるというものである. 本研究では, 主に Xie らによる関数とファイルのサマリー [13] と Dahse らによる RIPS [14] を拡張し, クラスキャッシュと 2 方向後方脆弱性分析という概念を導入して問題の解決に取り組む. このクラスキャッシュには 2 種類あり, それぞれ 2 種類のクラスサマリーを持つ. 最初にオブジェクト指向において XSS 攻撃が発生する状況の分類. また, それらをそれぞれ掘り下げて分類していき, クラスキャッシュと 2 方向後方脆弱性分析による分析方法を説明する. 最後に Dahse らによりオブジェクト指向の分析において課題だとする状況における分析手法ついて説明する.

3.1.1　攻撃発生状況の分類

オブジェクト指向において XSS 攻撃が発生する状況として大きく 2 つに分類される.

Case 1 ユーザ入力由来の変数を引数として, echo などの脆弱を引き起こす可能性のある組み込み関数を持つクラス関数が呼び出される場合.

Case 2 ユーザ入力由来のクラスプロパティが echo 関数などの脆弱な可能性を含む組み込み関数によって呼び出される場合.

3.1.2　クラス関数が呼び出される場合

ユーザ入力由来の変数を引数とした echo など脆弱を引き起こす可能性のある組み込み関数を持つクラス関数が呼び出される場合が存在する. 最初にソースコードをスキャンしてクラスメソッドの脆弱な情報をクラスキャッシュとしてキャッシュする. クラス関数の情報を含むクラスキャッシュは図 3.1 のようになる. クラスキャッシュは各クラスごとにそれぞれのクラスに含まれる関数の情報, それら関数についての脆弱性につい

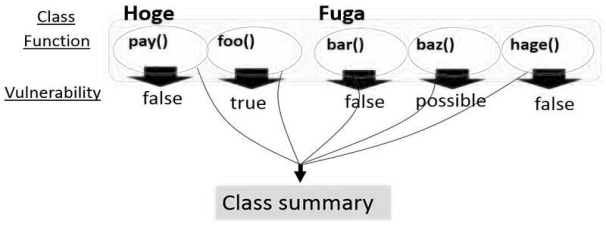

図 3.1: クラスメソッド用のクラスキャッシュ

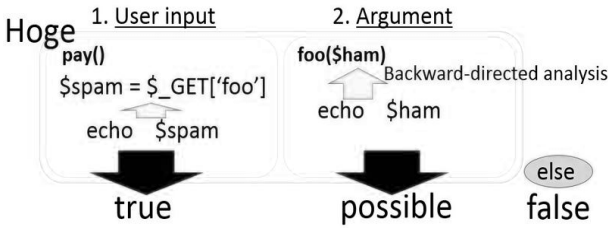

図 3.2: クラスメソッド用のクラスサマリー

て true, possible, false の 3 種類の情報を含む．true はこのクラス関数が呼び出された場合に無条件で脆弱となる場合．false はこのクラス関数が呼び出されても脆弱性に繋がる可能性のない場合．possible はクラス関数が呼び出された場合において, 関数の引数がユーザ入力由来である場合に脆弱性となる場合における情報である．クラスキャッシュのそれぞれの要素はクラスサマリーと呼び, 以下クラスサマリーの詳細を述べる．

関数におけるクラスサマリーについて説明したものについて図 3.2 のとおりとなる．この時のクラスサマリーには true, possible, false の 3 種類の情報が含まれる．true の場合について, echo 関数など脆弱性に繋がる組み込み関数が呼び出される際に, 呼ばれる対象が変数であり, かつその変数を後方脆弱性分析によりソースコードを遡って行った際に変数がユーザ入力由来であると分析された場合である．possible について, echo 関数など脆弱性に繋がる組み込み関数が呼び出される際に, 呼ばれる対象が変数であり, かつその変数を後方脆弱性分析によりソースコードを遡って行った際に変数が関数の引数由来であると分析された場合である．この場合, 関数が呼び出される時, 関数の引数がユーザ入力由来であれば脆弱性に繋がるが, それ以外の場合であれば脆弱性にはならないということを意味する．false について, 関数が true や possible の条件を満たさない場合すべての場合を意味する．

クラスキャッシュを作成後, このクラスキャッシュを基にして脆弱性分析を行う．本研究ではオブジェクト指向の分析のために 2 方向後方脆弱性分析という概念を導入して分析を行う．2 方向後方脆弱性分析を説明した図は 3.3 となる．$hoge→ baz($piyo) について分析を行う際に, 最初に引数に渡された$piyo 変数について一方の後方脆弱性分析を行う．この変数の代入などの繰り返しを遡って行った際に$_GET などによる

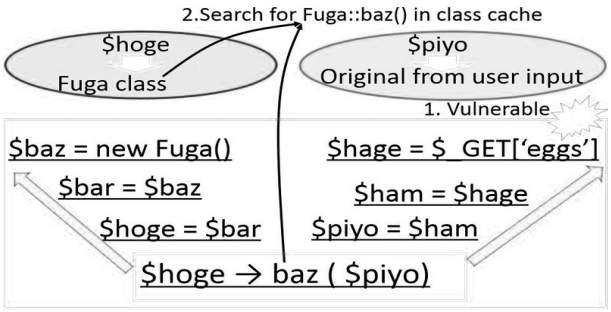

図 3.3: クラスメソッドにおける 2 方向後方脆弱性分析

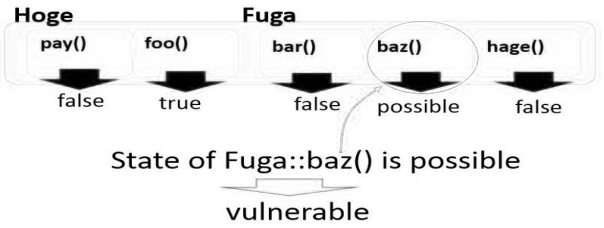

図 3.4: クラスメソッドの脆弱性情報の探索

ユーザ入力由来であると分析された場合, 脆弱な可能性があるとして, さらにもう一方の後方脆弱性分析を行う. こちらの後方脆弱性分析は$hoge を対象に後方向脆弱性分析を行い, 変数の代入などの繰り返しを遡って行った際にインスタンスの生成部分で宣言されたクラス名を特定する. 図 3.3 中においては Fuga クラスの baz 関数が引数としてユーザ入力由来の変数$poyo を渡しているために脆弱であるかの可否をクラスキャッシュに情報を探索しにいく. 探索について説明したものは図 3.4 となる. ここでは Fuga クラス中の baz 関数の情報を探索しにいったときに possible という情報を持っている. この possible は Fuga クラスの baz 関数が引数としてユーザ入力由来のものを受け取った場合に脆弱性となるという意味であるため, 探索の結果$hoge→ baz($piyo) は脆弱性であると判断する.

3.1.3　クラスプロパティが原因となる場合

ユーザ入力由来の変数を持つクラスプロパティが原因となる場合が存在する. ユーザ入力由来の変数を持つクラスプロパティが原因となる場合について, プロパティに値が代入される場合とプロパティの値が呼び出される場合の 2 つの場合について考える.

オブジェクト指向において, プロパティに値が代入される場合について大きく 4 つの状況に分類する.

Case 1 クラスプロパティにユーザ入力由来の値が直接代入される場合. 例)$hoge→

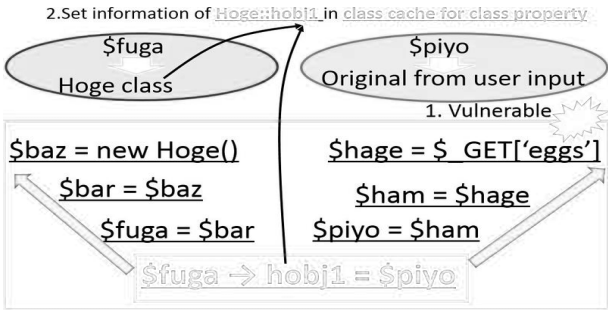

図 3.5: クラスプロパティにおける 2 方向後方脆弱性分析

obj=$_GET['input']

Case 2 クラスメソッドでセット関数を通してクラスプロパティに値が代入される場合 例) $hoge→ setParam($fuga)

Case3 コンストラクタ内でユーザー入力由来の変数がクラスプロパティに直接値が代入される場合.

Case4 コンストラクタ内で引数を通してクラスプロパティに値が代入される場合.

一つ目のクラスプロパティにユーザ入力由来の値が直接代入される場合について, 関数における分析と同様に 2 方向後方脆弱性分析を行う. これを説明したものは 3.5 となる. 最初に一方の後方脆弱性分析を行い$piyo の代入の繰り返しを遡って行く. もし, $piyo がユーザー入力由来であれば脆弱な可能性があると判断する. ここでは遡った先が$_GET['eggs'] 由来とユーザからの入力を受け取るものであるため, 脆弱な可能性があると判断する. 脆弱な可能性があると判断された場合, もう一方の後方脆弱性分析を $fuga を対象に行う. $fuga の代入の繰り返しを遡って行った先に Hoge クラスのインスタンスが生成されている箇所がある. そのため, $fuga は Hoge クラスのインスタンスを持つと分析する. 次に, Hoge クラスの hobj1 プロパティは脆弱であるという情報をプロパティ用のクラスキャッシュにキャッシュする. このクラスキャッシュの要素であるプロパティ用のクラスサマリーも関数のクラスサマリーの場合と同様のものであり, 3.6 の通りとなる.　関数用のクラスサマリーとプロパティ用のクラスサマリーの大きな違いについて説明する. 関数用のクラスサマリーは各クラスの情報, その各クラスにはそれぞれの関数の情報, それらの各関数には脆弱性の情報である true, possible, false の 3 種類の値をキャッシュしていた. 一方で, プロパティ用のクラスサマリーは各クラスの情報, その各クラスにはそれぞれ所有するプロパティの情報, それらの各プロパティには脆弱性の情報である true と false の 2 種類の情報を持ち, それぞれ true はユーザ入力由来であるため脆弱性を引き起こすもの, false はそうでない場合を表す.

図 3.6: クラスプロパティ用のクラスサマリー

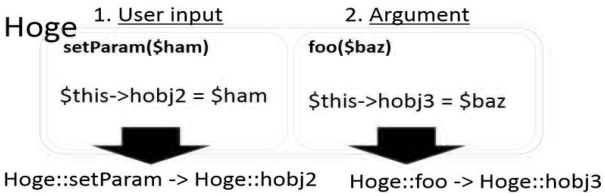

図 3.7: クラスメソッド用のクラスキャッシュ

二つ目のクラスメソッドでセット関数を通してクラスプロパティに値が代入される場合について，　一つ目のクラスプロパティに値が直接代入される場合と同様だが, その前に関数用のクラスサマリーの拡張を行い, この拡張した情報を基にクラスプロパティに値が直接代入される場合のような処理を行う. このときの状況について説明した図が 3.7 となる. 関数用のサマリーの拡張方法について, 関数用のクラスサマリーを作成する際にクラスメソッド内でクラスプロパティに値が直接代入されている場合, このクラスメソッドから対象のクラスプロパティに対してエイリアス情報を持たせる. この場合, Hoge クラスの setParam 関数中で$this→ hobj2 = $ham という代入がなされている. そのため, Hoge クラスの setParam 関数は Hoge クラスの obj2 プロパティに対するエイリアス情報を持つ. このようなキャッシュを作っておき, もし, Hoge クラスの setParam 関数が呼び出された場合はクラスキャッシュの探索を行い, Hoge クラスの setParam 関数のクラスサマリーは Hoge クラスの hobj2 プロパティへのエイリアス情報を持つため, このクラスサマリーの脆弱性の情報をみて脆弱性の判断を行う.

三つめのコンストラクタ内でユーザ入力由来の変数がクラスプロパティに直接代入される場合について, これは一つ目のクラスプロパティにユーザ入力由来の値が直接代入される場合と同様だが, 解析のタイミングについてインスタンス生成時に行われる. ただし, 解析対象となるソースコードが実装したクラスは必ずインスタンスが生成されるのであればクラスの解析時にも可能である.

四つ目のコンストラクタ内で引数を通してクラスプロパティに値が代入される場合について, これは二つ目のクラスメソッドでセット関数を通してクラスプロパティに値が代入される場合と同様だが, 解析のタイミングについてインスタンス生成時に行われる. ただし, 解析対象となるソースコードが実装したクラスは必ずインスタンスが生成されるのであればクラスの解析時にも可能である.

次に, オブジェクト指向において, プロパティの値が呼び出される場合について大きく 2 つの状況に分類する.

Case 1 クラスプロパティが組み込み関数によって直接呼ばれる場合.　例) echo $hoge→obj

Case 2 クラスプロパティがクラスメソッドを通して呼ばれる場合　例) echo $hoge→getParam($fuga).

一つ目のクラスプロパティが組み込み関数によって直接呼ばれる場合について, これはオブジェクト指向において, オブジェクトを所有するレシーバを後方脆弱性分析を行う. この分析によってインスタンス生成部を特定することによりクラス名を特定し, クラス名とオブジェクトの組み合わせを具体的に特定する. 次に, プロパティが代入される場合の一つ目の場合におけるクラスキャッシュで, 先ほどのクラス名とオブジェクトの組み合わせの情報を探索する. このクラスキャッシュは, クラスプロパティにユーザ入力由来の値が直接代入される場合において作成したものである. 探索した情報が true であれば脆弱性があると判断し, false であれば脆弱性がないと判断する.

二つ目のクラスプロパティがクラスメソッドを通して呼ばれる場合について, これは関数用のクラスサマリーの拡張を行い, この拡張した情報を基にクラスプロパティが組み込み関数によって直接呼ばれる場合と同様の処理を行う. このときの状況を説明したものが図 3.8 となる. 関数用のサマリーの拡張方法について, 関数用のクラスサマリーを拡張する際に, クラスメソッド内でクラスプロパティの値を返り値とする場合はこのクラスメソッドから対象のプロパティに対してエイリアス情報を持たせる. この場合, Hoge クラスの getParam 関数中で return $this→hobj1 のように hobj1 プロパティが返り値として値を返している. そのため, Hoge クラスの getParam 関数は Hoge クラスの hobj1 プロパティに対するエイリアス情報を持つ. 解析時に echo などの脆弱性を引き起こす可能性のある組み込み関数によってクラスメソッドが呼ばれた場合の解析方法を説明したものが図 3.9 となる. 最初に getParam 関数における引数の由来を分析するために後方脆弱性分析を行う. ここでは$piyo 変数が代入されるのを遡って行き$_GET['eggs'] 由来であり, これがユーザ入力由来となり脆弱な可能性があると判断する. 次にもう一方の後方脆弱性分析を行う. ここでは$hoge 変数の代入を遡って行きインスタンス生成部で Fuga クラスが生成されているため, Fuga クラスの getParam 関数をクラスキャッシュに探索しにいく. この時のクラスキャッシュの探索の様子が 3.10 となる. クラスキャッシュ中に Fuga クラスの getParam 関数の情報を探索し, 先ほど作成したクラスサマリーの情報を確認する. このとき, Fuga クラスの getParam 関数は Fuga クラスの fobj1 プロパティにエイリアスが張られているため, Fuga クラスの fobj1 のプロパティをクラスキャッシュ中からクラスサマリーを見つける. もしここで true という情報を持っていれば, echo $hoge→getparam($piyo) は脆弱性を持つと判定される. false であれば脆弱性がないと判定される.

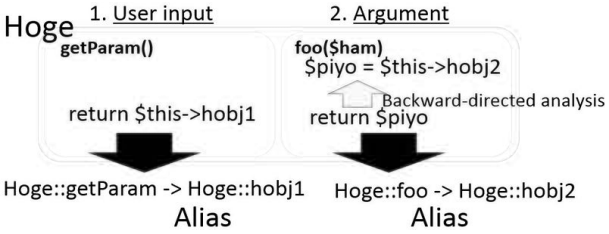

図 3.8: GET メソッドにより呼ばれる場合のクラスサマリー

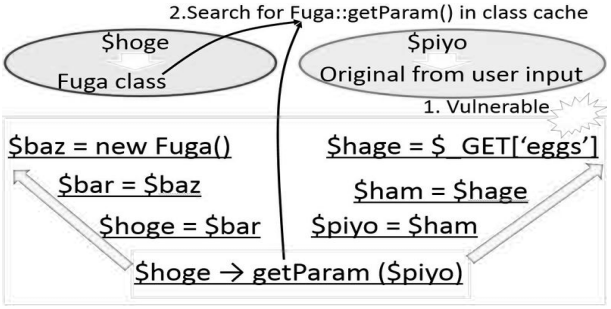

図 3.9: クラスメソッドによりクラスプロパティが呼ばれる場合における 2 方向後方脆弱性分析

3.2　実験と評価

3.2.1　データセット

データセットは PHP Vulnerabilities test suite を使用する．このデータセットは, NIST が公開しているテストケースであり, PHP の様々な種類の攻撃の脆弱性を安全な場合と安全でない場合においてファイルを用意したものとなる．各ファイルには 1 つの脆弱性を含むものとしている．以下のような脆弱性の種類が集められている．

- IDOR

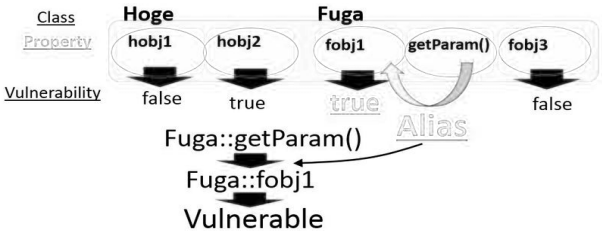

図 3.10: クラスメソッドによりクラスプロパティが呼ばれる場合における探索

- インジェクション
- SDE
- SM
- URF
- XSS

IDOR

IDOR とは, Insecure Direct Object Reference の頭文字を取ったもので, アプリケーションがユーザ入力に基づいたオブジェクトにアプリケーションが直接アクセスできる時に生じるものである. 結果として, 攻撃者は認証を通過し, データベース内のレコードやファイルといったシステム内のリソースに直接アクセスできるようになるというものである.

データセット中において IDOR の場合の脆弱性さらに以下の通りに分類している.

- Fopen
- SQL
- XPath

Fopen　ユーザ入力由来の値を第一引数として PHP の open 関数でファイルを開く操作を行ったファイルを集めたものである.

SQL　ユーザ入力由来の値を一部として含む SQL 文を実行するファイルを集めたものである.

XPath　XPath とは, PHP において xpath() 関数を使用して XML の解析した結果を出力するファイルを集めたものである.

インジェクション

インジェクションは, 以下のような様々な種類のインジェクション攻撃の種類の脆弱性を集めたものである.

- OS コマンドインジェクション
- SQL インジェクション

- LDAP インジェクション
- XML インジェクション
- ファイルインジェクション
- PHP リモートファイルインクルード

SDE

SDE とは, Sensitive Data Exposure の略で機密データ漏えいのことで, 以下の 2 つの種類の脆弱性のファイルを集めたものである.

- 機密データの暗号化忘れ
- 壊れた, もしくはリスクのある暗号化アルゴリズムの使用

SM

SM とは, Security Misconfiguration の略で, セキュリティの設定ミスのことで, ここではエラーメッセージを通した情報漏えいを含むファイルを集めたものである.

URF

URF とは, URL Redirects and Forwards のことで, ここでは信頼できないサイトへの URL のリダイレクションを含むファイルを集めたものである.

XSS

XSS 攻撃の脆弱性を含むファイルを集めたのである.

上記の種類の脆弱性において, 安全な場合と安全でない場合のファイルのデータセットがそれぞれ 用意されている. 各脆弱性における安全な場合と安全でない場合のファイルの数は表 3.1 の通りとなる. XSS 攻撃に脆弱性については, 安全な場合のファイルが 5728 個, 安全でない場合のファイルが 4352 個で合計 10080 個用意されている. 脆弱性全体では, 合計 42212 個のファイルがある.

本研究では, この内の XSS を使用している.

3.2.2　実験と評価

評価について, 代表的な脆弱性のパターンをテストコードとして用意したものによる評価と PHP Vulnerability test suite [49] というデータセットを用いた評価という 2 つの方法で行った.

代表的な脆弱性のパターンをテストコードについて,　以下のパターンを用意した.

表 3.1: データセットの内容

Vulnerability	Safe sample	Unsafe sample	Total
IDOR	400	80	480
Injection	20912	5920	26832
SDE	5	7	12
SM	5	3	8
URF	2208	2592	4800
XSS	5728	4352	10080
Total	29258	12954	42212

Case 1 ユーザ入力由来の変数を引数として動的なメソッドが呼び出され, このメソッド中で引数を echo などの組み込み関数によって呼ばれる場合

Case 2 コンストラクタ内でクラスプロパティに直接ユーザ入力由来の値が代入されこのクラスプロパティが直接 echo などの組み込み関数によって呼ばれる場合

Case 3 クラスメソッドを通してクラスプロパティを取り出して, echo などの組み込み関数によって直接呼ばれる場合

Case 4 クラスのプロパティに直接ユーザ入力由来の値を代入し, このプロパティを直接 echo などの組み込み関数によって直接呼ばれる場合

Case 5 ユーザ入力由来の変数を引数として静的なメソッドが呼び出され, このメソッド中で引数を echo などの組み込み関数によって呼ばれる場合

Case 6 インスタンスを生成して変数に代入してから, 実際にユーザ入力由来の変数を引数とした動的にメソッドが呼び出されるまでに変数の代入が繰り返される場合

Case 7 ユーザ入力由来の変数を引数として動的なメソッドが呼び出され, このメソッド中で別のクラスメソッドが呼び出され, このメソッド中で引数を echo などの組み込み関数によって呼ばれる場合

上記のパターンはいずれも検出可能と判定された.

PHP Vulnerability test suite のデータセットを対象に行った実験について, RIPS というツール名の既存研究を対象に行った. データセットのうちの XSS のものを使用した. ただし, 本研究の実装において, 実装において静的解析の対象外として, ユーザ入力由来の変数が引用符で囲われている場合や GET 値など直接ユーザ入力を受け取る

変数が echo 関数などで渡されるまでの間にキャストされている場合, または途中で配列のインデックスの一部を通して値のやり取りをするものなどの PHP の実装が含まれているファイルは除外した.

このデータセットを用いて, 既存研究として, RIPS と提案手法による実装のそれぞれで XSS 攻撃の脆弱性の静的解析を行った.

3.2.3　結果・考察

既存研究と提案手法で実際に静的解析して脆弱だと判定した出力数は表 3.2 となる. 脆弱な場合である 4349 個のファイルの内, RIPS では 276 個脆弱だと判定したのに対して, 提案手法では 2422 個が脆弱だと判定され, 2146 個多くの脆弱性を検知できる結果となった.

一方で, False Positive と False Negative の観点での結果は, 表 3.3 となる. RIPS では, False Negative が 93.65%となっており, 脆弱なデータセットの大部分を見逃している結果となっているのに対して, False Positive は 6.63%となっており, 誤検知率は比較的小さい結果となった. 提案手法では, False Negative は 44.31%となっており, 脆弱性を見逃す割合は先ほどの結果と比べて 49.34%減少する結果となった. しかし, False Positive は 49.74%となっており, 誤検知率は上昇するものとなった.

表 3.2: 脆弱だと判定した出力数

	Safe	Vulnerable
Sample number	5191	4349
RIPS	344	276
Suggestion	2582	2422

以下, 上記の結果について考察を行う. Dahse らは論文中でオブジェクト指向の分析が困難である状況として大きく以下の 3 つを挙げている.

Case 1 クラスメソッド中から別のクラスメソッドが呼ばれている場合

Case 2 引数にインスタンスが格納された変数が渡される場合

Case 3 関数内でインスタンスが格納された変数について global 変数が使用される場合

一つ目のクラスメソッド中から別のクラスメソッドが呼ばれている場合について, Dahse らは動的なクラスメソッドの呼び出しは実行時でないと決まらないためこのような状況に対応するのは困難であると述べている. しかし, 論文中で述べられている例

について, 別のクラスメソッドが現れた場合も本稿で提案したクラスキャッシュの作成時のクラスサマリーの情報を参照することで対応できる.

二つ目の場合である引数にインスタンスが格納された変数が渡される場合について, クラスメソッド用のクラスキャッシュを作る際に true, possible, false に加えて object という情報を追加し, object とセットでクラス名を格納することで, 分析時に引数由来のインスタンスからメソッドを呼び出す際にはこの情報を参照することで分析可能になる.

三つ目の関数内でインスタンスが格納された変数について global 変数が使用される場合について, これは変数名とクラス名の対応するキャッシュを新たに用意して作っておく. 関数内の分析中に global で宣言された変数があり, かつこれをインスタンスが格納されていると見なし, クラスメソッドが呼ばれている場合はこのキャッシュを参照することで分析可能になる.

また, 脆弱なデータセットの内, 1927 個のファイルが本提案手法で見つけることができなかった原因については, 以下のような原因が考えられる.

- オブジェクト指向でないファイルにおいて, 既存研究でも検知することができない脆弱性
- オブジェクト指向のファイルにおいて, クラス内の処理に着目し, ミクロな観点で既存研究でも検知することができない脆弱性
- source で出力される変数が sink でユーザ入力を直接受け取ってから処理されていく過程で考慮されていない要素の存在

オブジェクト指向でないファイルにおいて, 既存研究でも検知することができない脆弱性

オブジェクト指向でないファイルにおいて, 既存研究でも検知することができない脆弱性については, 従来研究でも見つけることができないファイルについては, 提案手法においても見ることはできない.

オブジェクト指向のファイルにおいて, クラス内の処理に着目し, ミクロな観点で既存研究でも検知することができない脆弱性

オブジェクト指向のファイルにおいて, クラス内の処理に着目し, ミクロな観点で既存研究でも検知することができない脆弱性について, クラス内に定義されている関数内の処理に着目した時に, その範囲内を対象に考えた時に, 既存研究でも検知することができない脆弱性は提案手法においても検知することはできない.

source で出力される変数が sink でユーザ入力を直接受け取ってから処理されていく過程で考慮されていない要素の存在

提案手法による実装では, source で出力される変数が sink でユーザ入力を直接受け取ってから処理されていく過程で変数に対して脆弱性の有無に関わる作用を働きかける要素で, 例えば以下の要素を考慮されていない.

- 暗黙的な型変換
- 組み込み関数による型変換
- 組み込み関数による文字列の置換

暗黙的な型変換　データセットの中に, 例えば, ユーザ入力由来の値を格納する変数を $\$tainted$ とした時に, 途中処理で $\$tainted + 0$ や $\$tainted + 0.0$ のような暗黙的な型変換がなされているファイルが存在する.

この場合, 元の $\$tainted$ が文字列の場合はそれぞれ int 型, float 型に変換される. そのため, 文字列としてスクリプトが含まされても int や float などの数値の型に変換されているため, 脆弱性が現れないと考えられる.

組み込み関数による型変換　データセットの中に, 例えば, ユーザ入力由来の値を格納する変数を $\$tainted$ とした時に, 途中処理で $intval(\$tainted)$ や $floatval(\$tainted)$ のような暗黙的な型変換がなされているファイルが存在する.

この場合, 元の $\$tainted$ が文字列の場合はそれぞれ int 型, float 型に変換される. そのため, 文字列としてスクリプトが含まされても int や float などの数値の型に変換されているため, 脆弱性が現れないと考えられる.

その他の場合として, データセットの中に, 例えば, ユーザ入力由来の値を格納する変数を $\$tainted$ とした時に, 途中処理でのような暗黙的な型変換がなされているファイルが存在する. この場合, 元の $\$tainted$ が文字列の場合は float 型に変換される. そのため, 文字列としてスクリプトが含まされても float などの数値の型に変換されているため, 脆弱性が現れないと考えられる.

組み込み関数による文字列の置換　データセットの中に, 例えば, ユーザ入力由来の値を格納する変数を $\$tainted$ とした時に, 途中処理でのような文字列の置換処理がなされているファイルが存在する. この場合, パターン修飾子として, si を指定しており, パターン中の文字列を大文字小文字をにもマッチさせ, かつ改行を含む文字にもマッチを指定し, アルファベット, 数字, アンダースコア以外の文字を除去する処理を行っている.

ここで script のタグでスクリプトを実行するような処理がある場合, このタグが preg_replace() 関数によって除去されるような処理があってスクリプトが実行できなくなるために脆弱性が現れない場合を考慮できない.

提案手法により, 誤検知率が上昇した理由について, 既存研究ではオブジェクト指向の解析がそもそも考慮されていなかった部分に対して, 提案手法により解析の対象となったが, クラス内に含まれる関数内の処理というミクロな観点で既存研究でも誤検知のあった部分が結果に表れてきたことが原因だと思われる.

表 3.3: 既存研究との比較

	False Positive	False Negative
RIPS	6.63%	93.65%
Suggestion	49.74%	44.31%

Chapter 4　ソースコード中におけるXSS攻撃の脆弱性の評価指標

4.1　提案手法

本研究では, 以下の 3 点を考慮したメトリクスの提案と実装を行う.

- 解析における脆弱性の危険度の評価,
- 脆弱性箇所周辺の文脈やアプリケーションが使用される環境に応じて的確に判定できる陽性の判定基準の明確化
- 対策の優先度判定

4.1.1　状況分類方法

最初に脆弱性箇所周辺の文脈やアプリケーションが使用される環境に応じた判定基準のために, 状況を分類する方法について述べる.

- ある source 単体
- 文脈に応じたある source
- その他の source

ある source 単体　ある source, 例えば PHP を例に取った時の典型的なものとして echo() 関数について考える.

echo 関数が GET の値を直接受け取った場合の単純な場合の状況分類を考える. これはアプリケーションが実際に使用される環境に応じて脆弱性がリスクになるかを考えるための分類である. この場合は, 動的解析で状況を分類する. echo $_GET['source']; のような PHP 中の文に対して, 例えば以下のような様々なパターンの値を投入する.

- <ScRiPt >alert(document.cookie) </sCrIpT >
- javascript:alert(document.cookie)//
- javascript:alert(document.cookie)

- <iMg srC=1 lAnGuAGE=VbS oNeRroR=mSgbOx (document.cookie) >
- <img src='1' onerror \x00=alert(0) / >
- <img src='1' onerror/=alert(0) / >
- <img src='1' onerror= \x00alert(0) / >
- <img src='1' onerror \x0b=alert(0) / >
- <img src='1' o \x00nerr \x00or=alert(0) / >
- <\x00img src='1' onerror=alert(0) / >
- <script \x00 >alert(1) </script >

この時にもし, どのパターンで実際にスクリプトが実行可能かを確認する. 環境に依存した状況を考えるために Google Chrome, Firefox, Internet Explore, Microsoft Edge などのブラウザで解析を行う.

文脈に応じたある source

ある source となる脆弱性が現れる箇所周辺のソースコードやソースコードが Web アプリケーションとして稼働するサーバサイドもしくはインフラの環境, または利用が想定されるクライアントサイド側の環境において, 脆弱性が現れるか現れないかを根拠とする基準となる.

- HTTP レスポンスのヘッダーにおいて文字コードが設定されているか
- HTML のタグ属性がダブルクオテーション, またはシングルクオテーションで囲われているか
- タグのメタ文字がエスケープされているか
- 想定される XSS 攻撃の種類の分類
- その他のコンテキスト

HTTTP レスポンスヘッダーにおいて文字コードが設定されているかについて, 例えば, UTF-7 において XSS 攻撃の脆弱性が現れるときに以下のような状況において現れる.

- HTTP レスポンスヘッダーもしくは meta タグにおいて文字コードを指定しなかった場合

- 文字コード宣言箇所より前に脆弱性箇所がある場合
- 指定した文字コードがブラウザが識別することができなかった場合

上記のような状況において, 文字コードの判定で UTF-7 だと認識させたときに現れる. これは Internet Explore7 以前の特定のブラウザで問題になっていたことがありCVE2007-1114 として識別子がつけられている. 現在これら Internet Explore 7 以前のバージョンの Internet Explore の利用者はほとんどいない.

HTML のタグ属性がダブルクオテーション, またはシングルクオテーションで囲われているかについて, これはタグ属性をクオテーションで囲わない場合において, 例えば, 以下のように脆弱性になり得る.

<input type=input name=”age” value= <?php echo $_GET[’age’]; ? >>

のような HTML の宣言において, URL 末尾に?age=1+onmouseover=alert(document.cookie) のようなクエリーを付加することで

<input type=input name=”age” value=1 onmouseover=alert(document.cookie) >

となり, JavaScript のスクリプトとして関数が実行される. これに対して,

<input type=input name=”age” value=” <?php echo $_GET[’age’]; ? >” >

のようにクオテーションをつけておいた場合,

<input type=input name=”age” value=”1 onmouseover=alert(document.cookie)” >

のように value の値が”1 onmouseover=alert(document.cookie)” という文字列として認識され JavaScript のスクリプトとして関数が実行されないというものである.

タグのメタ文字がエスケープされているかについて, これは source となる箇所が, HTML 中のどのような文脈において現れるかを考慮する. 例えば, 以下のような HTML 中でのコンテキストが想定される.

- タグ外のテキスト
- タグの属性値

タグ外のテキストについては, スクリプトが実行されるのを防ぐために以下の文字をエスケープすることが考えられる.

- <
- >
- &

<は < , >は >, & は & へエスケープ処理をする.

タグの属性値については幾つか分類が考えられる. 通常時, タグ外のテキストに加えて, ' や ' ' など引用符も以下のようにエスケープ処理をする必要がある.

' は ', ' ' は" へエスケープ処理をする. これをしない場合, ユーザ入力由来の変数中に引用符が含まれている場合, その属性の値を終端させて, 別の属性を含めることが可能なため脆弱性となり得る.

ただし, タグの属性値についても対象の属性に応じて適切に処理をしているかも確認事項であり, 例えば, 以下のものが挙がられる.

- a タグの href 属性
- img タグの src 属性
- style 属性
- イベントハンドラ
- スクリプトの文字列

上記の場合もそれぞれ適切に処理しているかを考慮してコンテキストを場合分けをする. href 属性や src 属性であれば, 先頭が http://, https://, または / が含まれるかなども考えられる. スクリプトの文字列であれば, エスケープ処理の方法も変わり, \ であれば \\, ' であれば \' , ' ' であれば \' ' , 改行であれば \n などのように適切な処理をしているかなどの確認する必要がある.

想定される XSS 攻撃の種類の分類について, 2 章において XSS 攻撃の分類で詳細を説明した種類があり, 大きく 3 つに分けたときに Reflected XSS, Stored XSS, DOM based XSS の 3 種類に分類される. これらは表 1 での説明のように XSS 攻撃実行時がサーバサイドやクライアントサイドのどちらで実行されるかや直接的に攻撃が実行されるか間接的に実行されるかなど様々な観点で異なり, これらに基づいて分類する必要がある.

アプリケーションの対策だけではなく, アプリケーションを稼働もしくは利用される環境の対策以外に, ゲートウェイでの対策を行い, 不審な通信パターンで XSS 攻撃に関する通信を遮断するなどの対策を実施しているかどうかの可否も挙げられる.

その他の source

先ほどの PHP における echo() 関数以外でも print, print_r など source となる元の関数のパターンにおいても上記のある source 単体や文脈に応じたある source を考える. もしくはそれ以外の XSS 攻撃の脆弱性となる原因を考慮する.

状況分類におけるコンテキストの追加　XSS 攻撃における攻撃の種類は多様化している．これは Web アプリケーションが日常生活で必要不可欠なものとしてなっており，アプリケーションに期待する機能の高度化や技術の発展に伴い，これまでに見つかってこなかった脆弱性も見つかり得る．

前章に挙げた，XSS 攻撃の脆弱性が現れる状況分類以外にも実際には様々に挙げられる．例えば，Content Seucrity Policy (CSP) の設定や，X-XSS-Protection，X-Content-Type-Options オプションの設定，Cookie やセッションに対する HttpOnly 属性の設定，ブラウザ間における XSS フィルターの差異，または設定の有無などがある．その他，ファイルアップロードに関して幾つかの場合が挙げられる．イメージファイルや git ファイルの拡張子を用いて，ファイルの中身はスクリプトとすることで攻撃する方法，もしくは，HTML 上でファイルをアップロードするとファイル名が表示されるものを利用して，ファイル名自体をスクリプトとすることで攻撃する方法などもあり，ファイルの中身の正当性やファイル名の確認処理がなされているかなども確認するなどのコンテキストも追加で考える必要が生じるなどが考えられる．

実際の XSS 攻撃が脆弱性となって現れるまでには細かいコンテキストも考慮すると数多く考えられる．それ故，解析ツールの開発者がツールを利用する実際のアプリケーションが特に重視しなければいけない項目，もしくは Web アプリケーションの開発者自身がコンテキストを適宜追加して使用することも可能である．

または，安全と利便性のトレードオフの戦略もあり，Web アプリケーションでの出力がある程度壊れても，アプリケーション自体が稼働し，かつ壊れた出力も許容できるのであれば，サニタイジングの方針も変わることも考えられる．例えば，<や >などタグのエンクロージャーが含まれていれば必ず取り除くなどの方針も考えられる．このようなアプリケーションが利用される環境や開発者の考える戦略も考慮してコンテキストを適宜変更，追加することも可能である．

4.1.2　評価方法

評価手法について，最初に脆弱性の拡散の度合いを以下ように分類する．

$$\begin{cases} \text{空間的広がり具合} \begin{cases} \text{世界中での広がり度合い} \\ \text{自分のソースコード内での広がり度合い} \end{cases} \\ \text{時間的広がり度合い} \end{cases}$$

最初に空間的広がりと時間的広がりの 2 つに分類する．さらに空間的広がりは世界中での広がりと自分のソースコード内での広がりの 2 つに分類する．

世界中での広がり度合いとは，世界中で利用されるシステムにおいてどの程度の危険度かを数値化するものである．自分のソースコード内での広がりとは，自分たちで開発した Web アプリケーションにおいて，脆弱性をパターン化した時にある種のパター

ンはソースコード内でどのくらい現れるかという出現頻度を表す. 空間的広がりとは, ある脆弱性に対して攻撃を受けた時にその脆弱性がどのくらい Web アプリケーションに残り続けるかを表す.

これら 3 つの指標は 0 から 10 の間の点数が付けられる. 最終的には 3 つの指標に対して重み付け平均を取って値を算出する. その結果の点数は 0 から 10 の間となる. 最終的な値が対策の優先順位も考慮した形で結果を出力する時の基準となる.

評価手法を構成する指標

上述の通り, 評価手法を構成する要素は, 世界中での広がり度合い, 自分のソースコード内での広がり度合い, 時間的広がり度合いの 3 つに分類される.

世界中での広がり度合い　世界中での広がり度合いとは, 検査対象の脆弱性がどの程度の危険かを表す. これは, 基本的には CVSS の概念を拡張して導入する. CVSS において, 基本評価基準を求める時に必要な, 機密性, 完全性, 可用性の評価値は, 先ほどの 6.1.3 章における状況分類方法に基づいて洗い出した脆弱性の種類のパターンを元にそれぞれ, アプリケーションが実際に使用される想定を元にスコアを算出する. 脆弱性の危険度に応じて結果を出力する時の基準となる.

自分のソースコード内での広がり度合い　自分のソースコード内での広がり度合いとは, ソースコード内であるパターンの脆弱性の出現頻度を表す. 6.1.3 章における状況分類方法に基づいて洗い出した脆弱性の種類のパターンがソースコード中において何回出現するかを計算する.

時間的広がり度合い　時間的広がり度合いとは, 検査対象の脆弱性がどの程度アプリケーションに残り続けるかの基準となる. 時間的にどれだけアプリケーションに残り続ければ被害が最大だとする時間を定義した後に, 4.2.1 章における状況分類方法に基づいて洗い出した脆弱性の種類のパターンがそれぞれどの程度脆弱性が残り続けるかを考え, その相対的な割合を 0 から 10 の間で評価する.

4.1.3　脆弱性陽性判定の出力基準

期待する脆弱性の陽性判定基準は実際に脆弱性の解析ツールを使用する利用者によって異なることが考えられる. 例えば, XSS 攻撃に詳しい人が利用する場合, 対象の source の脆弱性がリスクとなる可能性が低いものも含めて大量に出力しても, その中から実際に必要なものを選定できることが想定される. しかし, 必要なものを選定できない利用者によって使用される時, 例えば, 実際の Web アプリケーション開発の現場において, 選定要員や時間が制限された環境, もしくは XSS 攻撃に詳しくない人が脆弱性の

解析ツールを使用する場合においては, 確実に脆弱性がリスクになる場合のみを陽性として出力したい場合も考えられる.

つまり, 出力結果を適切に判定できる利用者が使用する場合は, 誤検知率となる False Positive Rate(FPR) が高くても見逃し率となる False Negative Rate(FNR) が低い可能性のある脆弱性を出力することが望まれる場合も考えれる. しかし, 結果を適切に判定できない利用者が使用する場合は, FNR が高くても FPR を低く, 可能性のある脆弱性を出力することが望まれる場合も考えられる.

閾値を設定することで期待する基準に基づいた結果を出力することが可能となる. 閾値の定義域は単回帰分析で決定する. 単回帰分析における直線の傾きの決定のために, 最小二乗法を使用する. 直線を

$$y = ax + b \tag{4.1}$$

ただし, x はスコア, y は出力割合, a, b は定数となる.

$$a = \frac{Cov(x, y)}{\sigma_x^2} \tag{4.2}$$

ただし, $0 \leq x \leq x_s$
x_s の決定方法について, 任意の a' において以下の式を満たす値である.

$$a' = \frac{a_{x=x_s}}{a_{x=x_s \ \ 1}} < 1 \tag{4.3}$$

ただし, 以下の式を満たすものとする.

$$x_s = 0.1 * s \tag{4.4}$$

任意の x_s において,

$$p \leq x_s < p + c \tag{4.5}$$

ただし, p は $p \geq d$ (d は定数) を満たす実数
閾値を $Threshold$ とすると, $Threshold$ は以下の式を満たすものとする.

$$Threshold_{max} = min(p) \tag{4.6}$$

以上の結果を用いて, 閾値 ($Threshould$) は以下の式を満たすものとする.

$$0 \leq Threshold \leq Threshould_{max} \tag{4.7}$$

4.2　実験と評価

4.2.1　データセット

データセットは, 3.2.1 章におけるオブジェクト指向で実装されたスクリプト言語における静的解析と同じ PHP Vulnerabilitu test suite を使用した.

4.2.2　実験と評価

実験は, PHP Vulnerability test suite を対象に RIPS というツール名の既存研究を対象に行った. データセットに含まれる安全である場合のファイル 5728 個と安全でない場合のファイル 4352 個のファイル, 合計 10080 個のファイルに対して, 提案手法を以下の観点から測定を行った.

- データセットを脆弱性をパターン別に分類したそれぞれの数
- データセットを脆弱性をパターン別に分類したそれぞれの割合
- 世界中での広がり度合いにおける, 提案手法によるスコアに対する実際に source に対して脆弱性だと判定した割合の分布
- 自分のソースコード内での広がり度合いにおける, 提案手法によるスコアに対する実際に source に対して脆弱性だと判定した割合の分布
- 時間的広がり度合いにおける, 提案手法によるスコアに対する実際に source に対して脆弱性だと判定した割合の分布
- 提案手法の評価における, 提案手法によるスコアに対する実際に source に対して脆弱性だと判定した割合の分布

4.2.3　結果・考察

パターン別に基づいたデータセットの脆弱性

データセットを脆弱性をパターン別に分類したそれぞれの数における実験結果は図 4.1 のようになる.

各分類ごとの右側における All とは source 全体を表しており, Output は source の内, 実際に解析の結果, 脆弱性を陽性だと判定した数を表す.

1 はユーザ入力由来の変数が source となる箇所で出力するまでの間にキャストをしているファイルの数である. 2 はユーザ入力由来の変数が source となる箇所で出力する時にシングルクオテーション, またはダブルクオテーションで変数の値を囲ってい

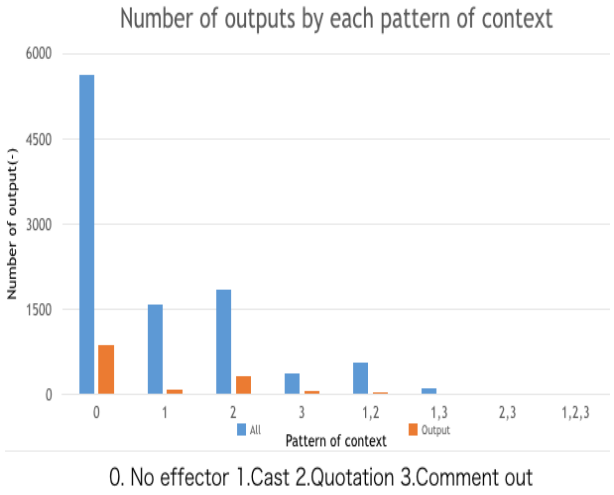

図 4.1: 脆弱性のパターン別分類とその出力の数

るファイルの数である. 3 はコメントアウト中でユーザ入力由来の変数が source となる箇所で出力しているファイルの数を表す. 0 は 1,2,3 やいずれの対策も施されていないファイルの数を表す. その他, 脆弱性がアプリケーションにどのくらい残り続けるかという指標や HTTP レスポンス中のヘッダーで文字コードの種類がセットされているかの判定も実装には含まれている. しかし, データセットに含まれる XSS 攻撃の脆弱性は Reflected XSS という特定の種類におけるもののみであったために結果には区別をつける要因とならなかった.

図 4.2 は図 4.1 を数ではなく割合で表したものとなっている. 割合で見た時に, キャストやダブルクオテーションもしくはシングルクオテーションにおける変数に対するクオテーションで囲う処理, またはコメントアウト中ではない文脈といういずれの XSS 攻撃の脆弱性の対策がなされていない場合である図中でいう 0 の場合が最も source の数に対して, 脆弱性判定を陽性だとする割合が多い結果となっている.

それ以外は source に対する脆弱性が陽性だと判定する割合が多い順に, クオテーションで囲った場合, キャストをしている場合, コメントアウト中の場合, ユーザ入力由来の変数が source で出力される前での間にキャストをしており, かつその変数が source でクオテーションで囲われて出力しているという複合の場合, ユーザ入力由来の変数が source で出力される前での間にキャストをしており, かつその変数が source がコメントアウト中で出力しているという複合の場合のような結果となっている. 以下, 定義域は式 4.5 において, $c = d = 0.5$ として定義域を求める.

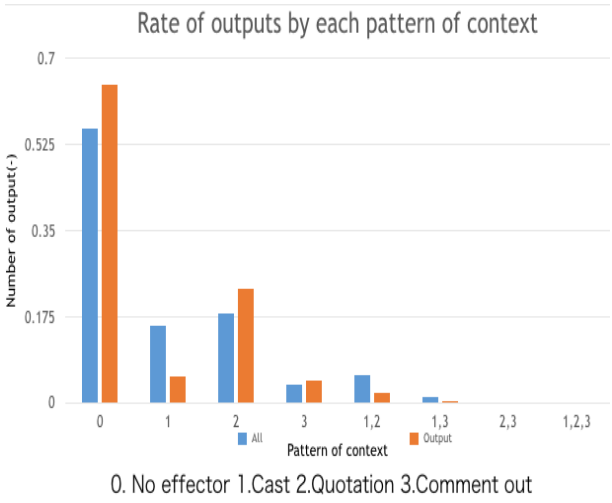

図 4.2: 脆弱性のパターン別分類とその出力の割合

世界中での広がり度合いにおける分布

世界中での広がり具合における, 提案手法によるスコアに対する実際に source に対して脆弱性だと判定した割合の分布における実験結果は図 4.3 のようになる. 図の縦軸は提案手法による各 source に対する脆弱性の危険度を表すスコアを 0.1 刻みで表したものであり, 横軸はあるスコアにおける source 全体のうち, 実際に脆弱性において陽性だと判定した数の割合を示す.

実験結果としては, 脆弱性の危険度が減少するにつれて, 脆弱性の陽性だと判定する割合も減少していく傾向を表すものとなった. これは, 脆弱性の危険度を表すスコアが大きければ, 実際に対象となる source が脆弱性がリスクとして判定し陽性だと判定する傾向が強いことを表している. 単調減少にならなかった理由としては, 使用した PHP Vulnerability test suite が実際に存在する XSS 攻撃の種類を網羅しているわけではなく, 特定の種類に偏っていることが原因の 1 つとして考えられる.

ソースコード内での広がり度合いにおける分布

ソースコード内での広がり具合における, 提案手法によるスコアに対する実際に source に対して脆弱性だと判定した割合の分布における実験結果は図 4.3 のようになる.

図の縦軸は提案手法による各 source に対する脆弱性の危険度を表すスコアを 0.1 刻みで表したものであり, 横軸はあるスコアにおける source 全体のうち, 実際に脆弱性において陽性だと判定した数の割合を示す.

この結果は, 使用したデータセットである PHP Vulnerabirity suite case 特有の結果となる.

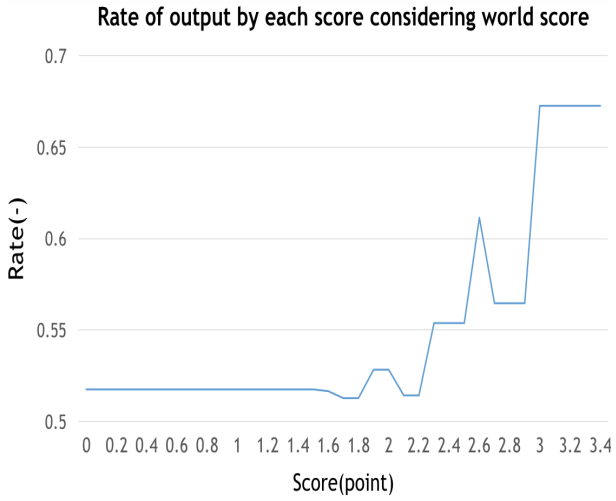

図 4.3: 世界中での広がり具合のスコア別出力割合

時間的広がり度合いにおける分布

使用したデータセットは全て Reflected XSS で脆弱性は瞬発的なものであるため, 時間的広がりぐらいにおけるスコアは 0 となる.

提案手法における分布

提案手法によるスコアに対する実際に source に対して脆弱性だと判定した割合の分布における実験結果は図 4.5 のようになる. ただし, 以下のそれぞれ 3 つの値の 0 から 10 の 3 つの要素において, 均等に比重を置いて, 加重平均を取った 0 から 10 の間のスコアとなる.

- 世界中での広がり具合のスコア別出力割合
- ソースコード内での広がり具合におけるスコア別出力割合
- 時間的広がり具合におけるスコア別出力割合

図の縦軸は提案手法による各 source に対する脆弱性の危険度を表すスコアを 0.1 刻みで表したものであり, 横軸はあるスコアにおける source 全体のうち, 実際に脆弱性において陽性だと判定した数の割合を示す. これにより, 実際の用途に応じて必要な FPR や FNR を閾値として, 基準を満たす結果のみを出力することが可能となる.

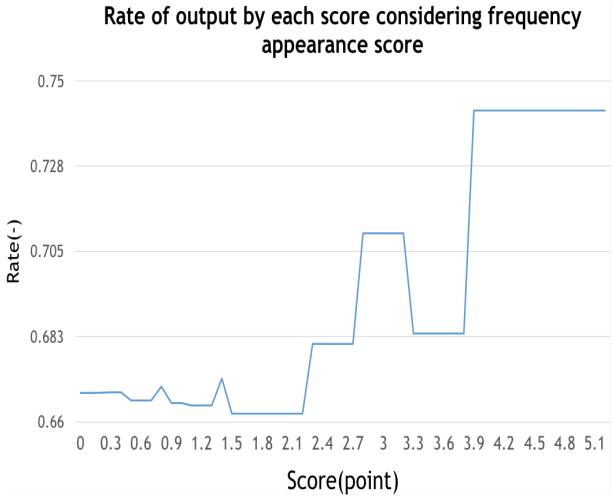

図 4.4: ソースコード中での広がり具合のスコア別出力割合

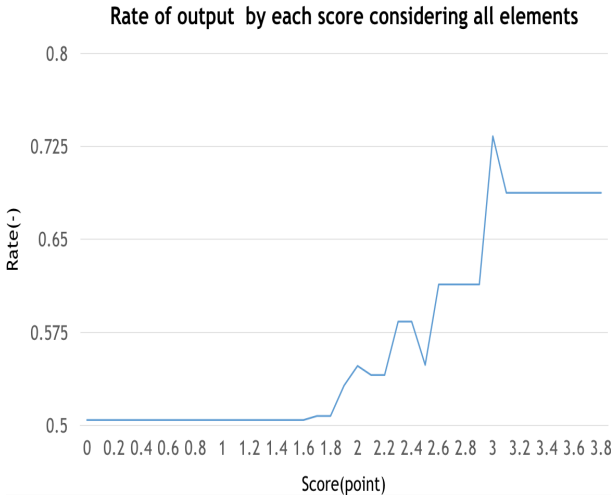

図 4.5: 提案手法のスコア別出力割合

Chapter 5　結論

本稿では Web アプリケーションの脆弱性における静的解析について, 特にスクリプト言語を対象にオブジェクト指向における XSS 攻撃の脆弱性の自動検知のための静的解析手法を提案し, サーバサイド言語で 80 %以上の割合で使用される PHP 言語を対象に実装を行った. XSS 攻撃はこれまで数多くの研究がなされてきているにも関わらず, 常に被害の最も大きい Web アプリケーションの脆弱性の攻撃の中で対象となっている. 原因はサーバ側だけでなく, クライアント側のブラウザの種類やバージョン, その他様々な要素が起因してくるため完璧な対策は不可能に近いものであろう. しかし, 多くの研究がなされてきているにも関わらず, 静的解析では精度は低い. また, XSS 攻撃の対策を防ぐための方法はある程度確立されてはいるが, それにも関わらずなくならない理由としては検査やレビューの技術が未発達であるといえる.

本研究において, オブジェクト指向に対して静的解析を可能にするための手法の提案と実装を行ったことにより検知可能な XSS 攻撃の脆弱性の種類が広がった. また, 脆弱性の原因となる箇所において危険度をスコア化し, 脆弱性箇所周辺の文脈に応じて的確に判定できる陽性の判定基準を脆弱性検査ツールの利用者にも明確化した. そして, 実際の Web アプリケーション開発の現場における限られた要員や時間において対策するべきものを判定するための, 対策の優先度判定することが可能となるようなメトリクスを提案し, 実装と評価を行った. 実験結果は, 既存研究において脆弱性を陽性だと判定する基準が何も脆弱性が対策されていない場合において最も脆弱性だと判定する割合が多いことを示した. さらに判定の根拠も明確化することを可能にした. また, 各脆弱性に対して, 提案手法の評価によって計算したスコアに基づいて脆弱性の原因となる箇所に対する脆弱性判定を陽性だとする割合の分布を示すことにより, 概ね単調減少の傾向となることを示した. これにより, 実際の用途に応じて必要な FPR や FNR を閾値として, 基準を満たす結果のみを出力することが可能となる. また, 既存研究において, 手法を比較することが困難であったが, 本研究におけるメトリクスを使用することにより, より有益な情報として対象の手法の比較できるようになるであろう.

謝辞

本研究を遂行するにあたり, 日頃からご指導くださった東京大学生産技術研究所の松浦幹太教授には大変感謝致します. また, 特任教授の Miodrag Mihaljevic さん, 研究室生活が円滑に過ごせるように尽力してくださった秘書の仲野小絵さん, 佐伯麻紀さん, 鶴山陽子さん, 技術職員の細井琢郎さん, 並びに研究室やミーティング等で助言をくださった松浦研メンバーである石坂理人さん, 竹之内玲さん, 孫達さん, 先崎佑弥さん, 今田丈雅さんに大変感謝致します.

参考文献

[1] Grossman et al. ：XSS Attacks: CROSS SITE SCRIPTING EXPLOITS AND DEFENSE，SYNGRESS, pages 480, 2007.

[2] OWASP, https://www.owasp.org/index.php/Main_Page, 2016-01-30

[3] Hydara et al., Current state of research on cross-site scripting(XSS) - A systematic literature review, Information and Software Technology 58(2015), pages 170–186, 2015.

[4] W3techs: Historical trends in the usage of server-side programming languages for websites, http://w3techs.com/technologies/history_overview/ programming_language, 2016-01-30

[5] Balzarotti, Davide, et al.,Saner: Composing static and dynamic analysis to validate sanitization in web applications., Security and Privacy, 2008. SP 2008. IEEE Symposium on. IEEE, 2008.

[6] Dahse, Johannes, and Thorsten Holz, Simulation of built-in PHP features for precise static code analysis., Symposium on Network and Distributed System Security (NDSS), 2014.

[7] Dahse, Johannes, and Thorsten Holz, Static detection of second-order vulnerabilities in web applications, USENIX Security Symposium, 2014.

[8] Huang, Yao-Wen, et al., Securing web application code by static analysis and runtime protection, Proceedings of the 13th international conference on World Wide Web. ACM, 2004.

[9] Kneuss, Etienne, Philippe Suter, and Viktor Kuncak, Phantm: PHP analyzer for type mismatch, Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, pages 373–374, 2010.

[10] Son, Sooel, and Vitaly Shmatikov, SAFERPHP: Finding semantic vulnerabilities in PHP applications, Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security. ACM, 2011.

[11] Wassermann, Gary, and Zhendong Su, Static detection of cross-site scripting vulnerabilities, Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. IEEE, pages 171–180, 2008.

[12] Wassermann, Gary, and Zhendong Su, Sound and precise analysis of web applications for injection vulnerabilities, ACM Sigplan Notices. Vol. 42. No. 6. ACM, pages 32–41, 2007.

[13] Xie, Yichen, and Alex Aiken, Static Detection of Security Vulnerabilities in Scripting Languages, USENIX Security. Vol. 6, pages 179–192, 2006.

[14] Dahse, Johannes, Nikolai Krein, and Thorsten Holz, Code Reuse Attacks in PHP: Automated POP Chain Generation, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security(2014), pages 42–53, 2014.

[15] Dahse, Johannes, and Jorg Schwenk, RIPS-A static source code analyser for vulnerabilities in PHP scripts, Retrieved: February 28 (2010), 2012.

[16] Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities, Security and Privacy, 2006 IEEE Symposium on. IEEE(2006), pages 6–263, 2006.

[17] Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda, Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report), Secure Systems Lab, Vienna University of Technology (2006), 2006.

[18] Smaragdakis, Yannis, Martin Bravenboer, and Ondrej Lhotak, Pick your contexts well: understanding object-sensitivity, ACM SIGPLAN Notices 46.1(2011), pages 17–30, 2011.

[19] Gupta, Mukesh Kumar, et al., XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications, Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on. IEEE, 2015.

[20] Alfred V. Aho et al., Compilers: Principles, Techniques, and Tools, Addison Wesley, ISBN: 0201100886, pages 466, 1986.

[21] Stefan Esser, Shocking News in PHP Exploitation, POC(2009), 2009.

[22] Stefan Esser, Utilizing Code Reuse Or Return Oriented Programming in PHP Applications, BlackHat USA(2010), 2010.

[23] Parameshwaran, Inian, et al., Auto-patching DOM-Based XSS at Scale, Foundations of Software Engineering (FSE) (2015), 2015.

[24] Weissbacher, Michael, et al., ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities, 24th USENIX Security Symposium (USENIX Security 15). USENIX Association, 2015.

[25] Stock, Ben, et al., Precise Client-side Protection against DOM-based Cross-Site Scripting, Proceedings of the 23rd USENIX security symposium(2014), pages 655–670, 2014.

[26] Lekies, Sebastian et al., 25 Million Flows Later - Large-scale Detection of DOM-based XSS, Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

[27] Stock, Ben, et al., From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting, Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.

[28] PHP Vulnerability test suite, https://github.com/stivalet/PHP-Vulnerability-test-suite, 2016-01-30

[29] Klein, Amit, DOM based cross site scripting or XSS of the third kind, Web Application Security Consortium, Articles 4 (2005), 2005.

[30] ソフトウェア等の脆弱性関連情報の取扱いに関する届出状況 [2016 年第 1 四半期（1 月～3 月）], https://www.ipa.go.jp/files/000052095.pdf

[31] OWASP Top Ten Project: 2013 Top 10 List, https://www.owasp.org/index.php/Top_10_2013Top_10

[32] Hydara, Isatou, et al. "Current state of research on cross-site scripting (XSS)-A systematic literature review." Information and Software Technology 58 (2015): 170-186.

[33] OWASP, https://www.owasp.org/index.php/Main_Page

[34] XSS (Cross Site Scripting) Prevention Cheat Sheet, https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

[35] XSS Filter Evasion Cheat Sheet, https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

[36] DOM based XSS Prevention Cheat Sheet, https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet

[37] Klein, Amit, DOM based cross site scripting or XSS of the third kind, Web Application Security Consortium, Articles 4 (2005), 2005.

[38] Stock, Ben, et al., From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting, Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.

[39] JNSA: 情報セキュリティの三大要件, http://www.jnsa.org/ikusei/basis/02_02.html

[40] IPA: 共通脆弱性評価システム CVSS 概説, https://www.ipa.go.jp/security/vuln/CVSS.html

[41] OWASP Risk Rating Methodology, https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

[42] Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities, Security and Privacy, 2006 IEEE Symposium on. IEEE(2006), pages 6–263, 2006.

[43] Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda, Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report), Secure Systems Lab, Vienna University of Technology (2006), 2006.

[44] Dahse, Johannes, Nikolai Krein, and Thorsten Holz, Code Reuse Attacks in PHP: Automated POP Chain Generation, Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security(2014), pages 42–53, 2014.

[45] Dahse, Johannes, and Jorg Schwenk, RIPS-A static source code analyser for vulnerabilities in PHP scripts, Retrieved: February 28 (2010), 2012.

[46] Wassermann, Gary, and Zhendong Su, Sound and precise analysis of web applications for injection vulnerabilities, ACM Sigplan Notices. Vol. 42. No. 6. ACM, pages 32–41, 2007.

[47] Xie, Yichen, and Alex Aiken, Static Detection of Security Vulnerabilities in Scripting Languages, USENIX Security. Vol. 6, pages 179–192, 2006.

[48] Gupta, Mukesh Kumar, et al., XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications, Advances in Computing,

Communications and Informatics (ICACCI), 2015 International Conference on. IEEE, 2015.

[49] PHP Vulnerability test suite, https://github.com/stivalet/PHP-Vulnerability-test-suite

発表文献

i 林昌吾, 松浦幹太. ”オブジェクト指向の Web アプリケーションに対する XSS 攻撃脆弱性の静的解析”, コンピュータ・セキュリティ研究会 2016 (CSEC2016), 東京, 3 月, 2016 年.

ii Shogo Hayashi, Keiichi Ishikawa, Yu Asabe, Masashi Ikarashi, Yuka Takahashi, Kanta Matsuura. ”Investigating Universal Metrics of Vulnerability Regarding Cross-Site Scripting Attacks”, International Workshop on Security2017 (IWSEC2016), Poster Session, Tokyo, September, 2016

iii 林昌吾, 松浦幹太. ”スクリプト言語によるオブジェクト指向の WEB アプリケーションにおける XSS 攻撃脆弱性に対するクラスキャッシュを用いた静的解析”, 2017 年 暗号と情報セキュリティシンポジウム (SCIS2017), 予稿集 USB メモリ, 2C3-4. 沖縄, 1 月, 2017 年.

iv 林昌吾, 松浦幹太. ”ソースコード中の XSS 攻撃脆弱性に関する評価指標の提案と実装”, 2017 年 暗号と情報セキュリティシンポジウム (SCIS2017), 予稿集 USB メモリ, 2C3-5. 沖縄, 1 月, 2017 年.