修士論文

パフォーマンスを考慮したプリミティブな Trusted Types による Client-Side XSS 防御手法

山崎 勇二
奈良先端科学技術大学院大学
先端科学技術研究科
情報理工学プログラム

主指導教員: 藤川 和利 教授
情報基盤システム学 研究室（情報科学領域）

令和 2 年 3 月 9 日提出

本論文は奈良先端科学技術大学院大学先端科学技術研究科に
修士 (工学) 授与の要件として提出した修士論文である。

山崎 勇二

審査委員：

藤川 和利 教授　（主指導教員，情報科学領域）
林 優一 教授　（副指導教員，情報科学領域）
新井 イスマイル　（副指導教員，情報科学領域）

パフォーマンスを考慮したプリミティブな Trusted Types による Client-Side XSS 防御手法*

山崎 勇二

内容梗概

攻撃者が脆弱な Web アプリケーションに悪意のあるスクリプトを挿入し実行させる Cross Site Scripting(XSS) が問題となっており，近年ではクライアント側の機能増加に伴い，XSS の一種である Client-Side XSS による攻撃が多くなっている．この問題に対処するために，テイントトラッキングによる検知・防御策が提案されているが，パフォーマンスに影響が発生し，実用的なものが無いのが現状である．本研究では，パフォーマンスと既存の Web アプリケーションへの影響を小さくした Client-Side XSS の防御を目的とした，文字列の型で安全であるかを検証する Trusted Types を，プリミティブな型として定義することによる防御手法を提案する．Trusted Types を用いることにより，JavaScript を生成する実行シンクは開発者が安全な文字列であると明示した文字列のみを許可するため，Client-Side XSS の脆弱性を塞ぐことが可能となり，加えて防御に必要な処理は文字列の型を検証するのみとなるため，パフォーマンスへの影響を軽減しながら防御が可能となる．この Trusted Types をプリミティブな型として定義し，JavaScript ソースのパースの段階で割り当てることで，既存の Web アプリケーションに変更を加えず Trusted Types を適用させる．評価結果から，既存手法はパフォーマンスへの影響として 7〜17%のオーバーヘッドが生じるが，提案手法では約 1%に軽減することが出来た．

キーワード

Web セキュリティ, Cross Site Scripting, JavaScript

*奈良先端科学技術大学院大学 先端科学技術研究科 修士論文, 令和 2 年 3 月 9 日.

Protecting Client-Side XSS with Primitive Trusted Types for Performance*

Yuji Yamasaki

Abstract

In recent years, the Web has become one of the most important platforms on the Internet. On the other hand, the vulnerability of Cross-Site Scripting (XSS), which allows an attacker to insert and execute a malicious script in a vulnerable Web application, has become a problem. In recent years, with the increase in client-side functions, Client-Side XSS, a kind of XSS, has become a problem. In order to deal with this problem, detection and protection methods using taint tracking have been proposed, but there is no practical solution since it affects performance. The purpose of this research is to protect Client-Side XSS with reduced performance and impact on existing Web applications. Specifically, we propose a protection method by using Trusted Types, which verify the type of a safe string by type, as primitive types. By using Trusted Types, the execution sink that generates JavaScript allows only the strings that the developer has specified as safe strings, so it is possible to close the vulnerability of Client-Side XSS. Since the only processing required for defense is to verify the type of the string, it is possible to protect while reducing the effect on performance. By defining this Trusted Types as a primitive type and assigning it at the stage of parsing the JavaScript source, the Trusted Types can be applied to existing Web applications without any changes. The evaluation results show that the conventional method

*Master's Thesis, Graduate School of Science and Technology, Nara Institute of Science and Technology, March 9, 2020.

has an overhead of 7–17% or more than 5% as an effect on performance, but the proposed method could reduce it to about 1%.

Keywords:

Web Security, Cross Site Scripting, JavaScript

目 次

図 目 次

表 目 次

1. はじめに

現在，Web はインターネットの中でも最も重要なプラットフォームの一つとなっており，HTML や CGI などのコンテンツを配置しているサーバの機能が増加している．その一方で，Web ブラウザなどのクライアント側でも機能が増加していることが特徴として挙げられる．その中でも JavaScript は，人気のある Web アプリケーション、ブラウザー拡張機能（またはアドオン）、HTML5 モバイルアプリケーション（WebView、Windows 8 Metro アプリ）など，クライアント側で動作するスクリプト言語として広く普及している．

その一方で問題となっているのが攻撃者が脆弱な Web アプリケーションに悪意のあるスクリプトを挿入し実行させる Cross Site Scripting (以下 XSS) の脆弱性である．攻撃者が XSS を行うことで，Cookie の窃取からセッションハイジャックを引き起こし，金融取引などの実行が可能 [13] となる．図 1 に一年で報告された XSS 脆弱性の数の推移を示す [19]．年によって差はあるものの，1999 年に最初に発見されて以来 [1]、報告された XSS 脆弱性の数が増加していることが分かる．このように，XSS は Web アプリケーションに常に存在するセキュリティ上の問題とされており，15 年以上経過した今でも完全に問題が解決されていない．

さらに近年では，クライアント側への機能の増加に伴い，JavaScript の安全でない記述によりクライアント側で発生する Client-Side XSS の脆弱性が問題となっている．2013 年の調査では Alexa Top 5000 の約 10%に少なくとも 1 つの Client-Side XSS の脆弱性を含んでいる [8] ことが示されており，この脆弱性が最新の Web アプリケーションにおいても依然として広く蔓延している [10] ことが示されている．

この Client-Side XSS の大きな問題として，攻撃者が挿入する悪意のあるコードが，被害者であるクライアント側のコンテキストで実行されるため，従来の XSS 防御策として普及している，HTML サニタイザや XSS フィルタなどで防ぐことが難しいことが挙げられ，脆弱性の発見も困難だとされている．

この Client-Side XSS の問題に対処するために，テイントトラッキングを用いて実行シンクに格納される文字列を検査し，検知・防御を行う手法 [21] や，安全でないコードを安全なコードに置換するパッチを自動生成することによる防御策 [14] などのクライアント側に施す防御手法が提案されているが，Web ブラウザや

JavaScript エンジンのパフォーマンスに影響が発生する，または防御手法の導入が容易でないなど課題が多く，依然として実用的なものが無いのが現状である．

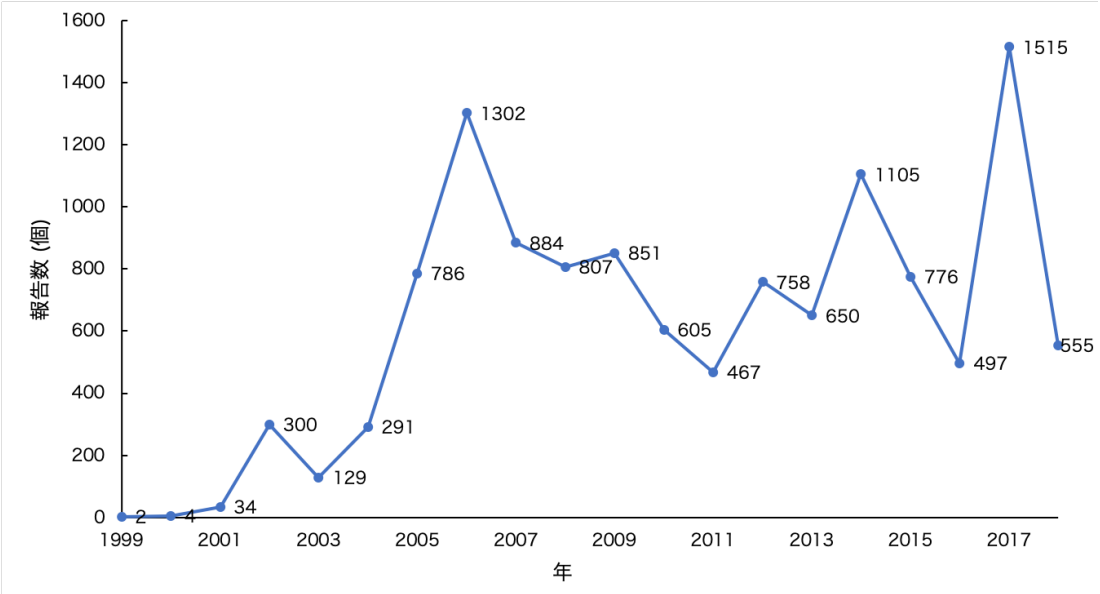

図 1: 報告された XSS 脆弱性の数の推移 [19]

このような背景で，近年 Client-Side XSS の防御策として議論されているのが，Trusted Types[6] である．この Trusted Types を用いることにより，開発者が安全な文字列であると明示した文字列のみ，JavaScript を生成する実行シンクへの格納を許可するため，Client-Side XSS の脆弱性を塞ぐことが可能となる．加えて，防御に必要な処理は文字列の型を検証するのみとなるため，パフォーマンスへの影響を軽減しながら防御が可能となる．一方で課題となるのが，Web アプリケーションの開発時に安全な文字列であると明示する手順が加わる他，Trusted Types を既存の Web アプリケーションに適用させるにはコードの変更が必要となり，近年の大規模な Web アプリケーションに対して変更を行うことが現実的でない点が挙げられている．

本研究では，パフォーマンスと既存の Web アプリケーションへの適用性を考慮した Client-Side XSS の防御を目的とし，

- パフォーマンスへの影響が小さい

- 既存の Web アプリケーションに変更を加えない
- DOM (Document Object Model) の動的追加・更新が可能

の 3 つを要件として定める．

そこで本研究では，安全な文字列であるかを型で検証する Trusted Types を，プリミティブな型として JavaScript エンジンに直接定義することにより，既存の Web アプリケーションに変更を加えずに Trusted Types を適用し，本来 JavaScript エンジンが有する型の機能を活用することで，パフォーマンスへの影響を抑えながら Client-Side XSS を防御する手法を提案する．

本稿の構成は以下の通りである．第 2 章では本研究で焦点を当てる Cross Site Scripting とその一種である Client-Side XSS について述べる．第 3 章では，Client-Side XSS の防御手法に関する先行研究を説明する．第 4 章では，パフォーマンスと既存の Web アプリケーションへの適用性を考慮したプリミティブな Trusted Types による Client-Side XSS 防御手法を提案する．第 5 章では，提案手法の JavaScript エンジンと Web ブラウザへの実装について述べる．第 6 章では，第 4 章で提案した手法の評価を行う．第 7 章では，評価実験の結果から考察を行い．先行研究との比較について述べる．第 8 章では，本稿のまとめを行う．

2. Client-Side XSS

本章では，本研究で対象とする Cross Site Scripting (XSS) の概要とその一種である Client-Side XSS の概要，従来の XSS 防御策の回避，攻撃モデルに関して述べる．

2.1 Cross Site Scripting (XSS) の概要

Cross Site Scripting (以下 XSS) とは，攻撃者が悪意のあるスクリプトコードを脆弱な Web アプリケーションに挿入し、被害者のブラウザで実行させるインジェクション攻撃の一種である．XSS を行うことで攻撃者が意図する電子メールの送信や金融取引の実行，セッションハイジャックなどの攻撃が可能 [13] となる．通常 Web アプリケーションの開発者がこの XSS から防御するには，攻撃者が操作可能な，信頼されていない全ての入力をサニタイズした上で利用することが必要だとされている．

この XSS は，大きく分けて

- 反射型 XSS
- 蓄積（格納）型 XSS
- Client-Side XSS

の 3 種類が存在するとされている．本研究で対象とする Client-Side XSS に関しては第 2.2 節で述べる．

2.1.1 反射型 XSS の概要

図 2 に反射型 XSS の標準的な攻撃モデルを示す．反射型 XSS の大まかな攻撃の流れとして，攻撃者があらかじめ反射型 XSS の脆弱性を含んだ Web サイトの情報を知った上で，

1. 攻撃者が悪意のあるスクリプトを埋め込んだリンクをメールで送信

2. 被害者がリンクをクリックし，脆弱な Web サイトへ HTTP リクエストを送信

3. サーバで悪意のあるスクリプトが構成され，HTTP レスポンスを返信

4. 悪意のあるスクリプトが実行され，攻撃者の元に被害者の情報が送信

となる．このように，悪意のあるスクリプトが HTTP リクエストの送信者へ返ることから反射型 XSS と呼ばれている [4].

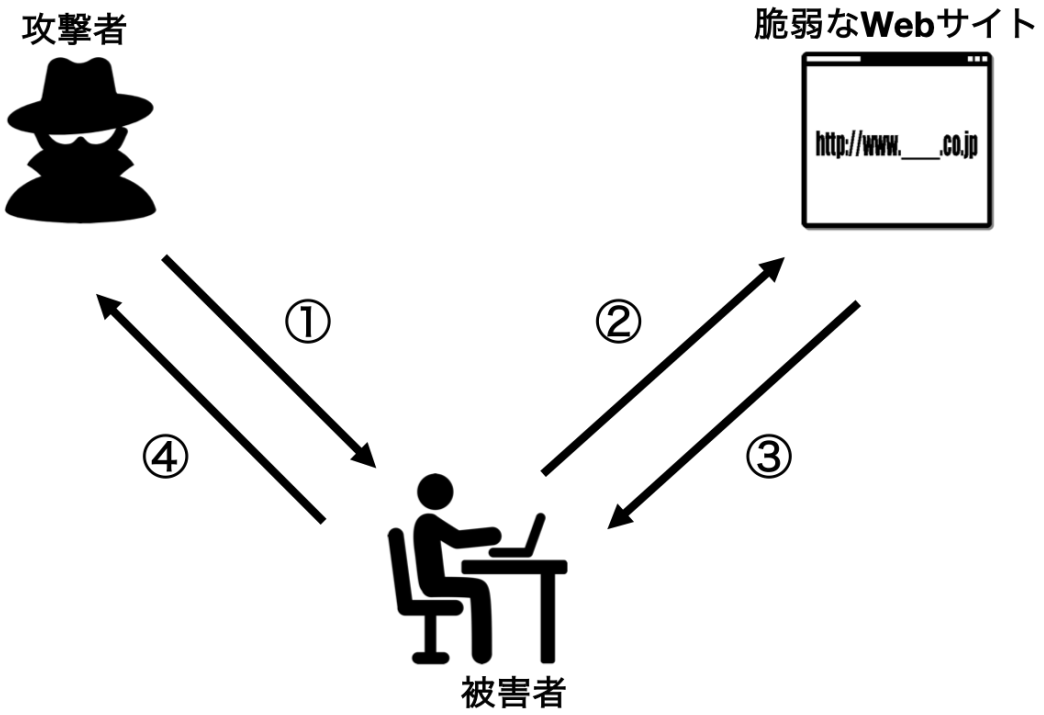

図 2: 反射型 XSS の標準的な攻撃モデル

2.1.2 蓄積（格納）型 XSS の概要

図 3 に蓄積（格納）XSS の標準的な攻撃モデルを示す．蓄積（格納）型 XSS の大まかな攻撃の流れとして，反射型と同様に攻撃者があらかじめ反射型 XSS の脆弱性を含んだ Web サイトの情報を知った上で，

1. 攻撃者が脆弱な Web サイトに悪意のあるスクリプトを送信
2. 悪意のあるスクリプトがサニタイズされずにデータベースに蓄積
3. 被害者が脆弱な Web サイトに攻撃者が登録したコンテンツをリクエスト
4. 攻撃者が登録したコンテンツを Web サイトが受け取る
5. 悪意のあるスクリプトが埋め込まれた攻撃者が登録したコンテンツを返信
6. 攻撃者に情報を送信

となる．このように，ユーザが当該ページを閲覧するたびに，保存された文字列がスクリプトとして実行されることから蓄積（格納）型 XSS と呼ばれている [4].

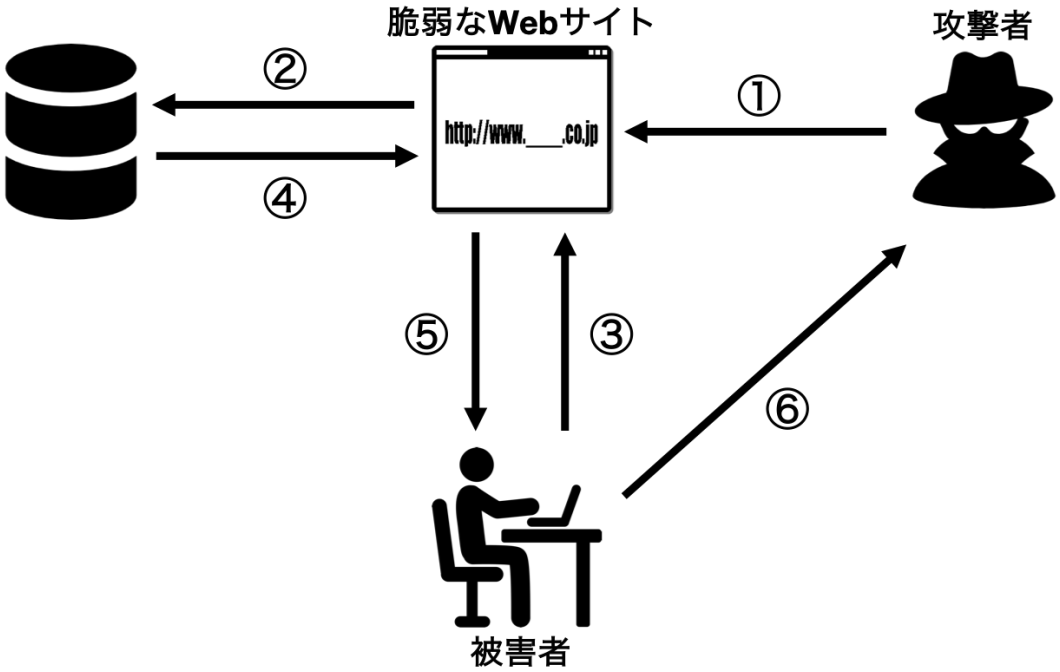

図 3: 蓄積（格納）型 XSS の標準的な攻撃モデル

2.1.3 普及している XSS 防御策

XSS の脆弱性が発見されて以降，Web アプリケーションに施す対策として様々な検知・防御策が検討されている．その中で現在広く普及しているのが，以下の 4 つの防御策である [7].

- HTML サニタイザ
  特定の記号を含ませない、含んでいる場合は別の文字に変換してエスケープ
- XSS フィルタ
  XSS に含まれる可能性の高いコードを検出してブロック
- Web Application Firewall
  悪意のあるリクエストを検出およびブロック
- Content Security Policy
  許可する JavaScript コードのみ開発者がホワイトリストに登録

これらの防御策は，基本的に以下の考え方に基づいた手法となっている．

- リクエストフィルタリング
  悪意のある HTTP リクエストをブロック (Web Application Firewall)
- レスポンスサニタイゼーション
  応答のサニタイズ (HTML サニタイザ)
- コードフィルタリング
  悪意のある JavaScript の検出 (XSS フィルタ, Content Security Policy)

2.2 Client-Side XSS の概要

Client-Side XSS は，2005 年に Amit Klein[5] によって DOM (Document Object Model) Based XSS という用語で議論が始まったものであり，脆弱なサーバ側のプログラムによって引き起こされるのではなく，ユーザが制御可能な入力がクラ

イアント側で安全に処理されない場合に発生する．したがって，eval() や document.write などの，文字列を引数として受け取り解析して JavaScript コードとして実行するシンクに，URL や location.hash などの入力ソースに格納されている文字列が安全に処理されないまま渡される場合，そのコードは Client-Side XSS の脆弱性を含んでいることとなる．

```
1 let name = "Your name is " + location.hash.slice(1);
2 document.write(name);
```

図 4: Client-Side XSS の脆弱性を含むコードの例

図 4 に Client-Side XSS の脆弱性を含むコードの例を示す．このコードが本来想定している動作は，URL のあるパラメータからユーザーの名前を抽出し DOM に動的に追加することで表示するものであるが，適切なエンコードやサニタイズが行われないため，Client-Side XSS の脆弱性を含んでいる．Web ブラウザによっては，location.hash などの URL におけるパラメータを自動でエンコードする機能を有するが，その機能を持たない Microsoft Edge などの Web ブラウザでは，URL のパラメータを<script>alert(1)</script> などのようにして攻撃者が被害者に悪意のあるスクリプトを実行させることが可能である．

先述した 3 種類の XSS の差は，反射型と蓄積（格納）型がコンテンツを配置しているサーバ側の CGI で攻撃者が挿入したスクリプトコードが構成されるのに対し，Client-Side XSS はクライアント側である被害者の Web ブラウザで構成される点が挙げられる．

また，Client-Side XSS の特徴として，単にスクリプトコードを埋め込むだけでなく，HTML の属性やタグを利用して攻撃可能なことや，攻撃に利用される入力が複雑な場合が多いことが挙げられる．その例の 1 つとして，図 5 に属性とタグを利用した Client-Side XSS の脆弱性 [21] を示す．

この例では，iframe タグが読み込まれた後の動作を location.hash に含まれる文字列で決定しようとするコードであるが，HTML の属性とタグを利用して攻撃が可能である．iframe タグが持つ onload 属性はイベントハンドラーのため，直接 JavaScript コードを記述することが可能である．そのため，図 5 の 7 行目のよう

```
1 var h = location.hash.slice(1);
2 var code = "<iframe onload='" + h + "'"
3     code += "src='http://example.org/img'></iframe>"
4 document.write(code);
5
6 // 属性への攻撃
7 "//example.org/#alert(1)"
8 // タグへの攻撃
9 "//example.org/#' srcdoc='<script>alert(1)</script>'"
```

図 5: 属性とタグを利用した Client-Side XSS の脆弱性 [21]

にスクリプトコードを挿入することで実行させることが可能となる．同様に，この例では 9 行目のように iframe の持つ srcdoc の属性を追加可能である．Web ブラウザは src 属性と srcdoc 属性が同時に指定された場合，多くは srcdoc 属性を優先するため，正常な iframe は表示されず，攻撃者が記述した悪意のあるコードが実行可能となる．

次に攻撃に利用される入力が複雑な場合の例として，図 6 に複数のパラメータを利用した Client-Side XSS の脆弱性 [21] を示す．

こちらの例も，iframe タグの id 属性と name 属性を URL のパラメータから決定するコードであるが，この 2 つのパラメータを利用して攻撃が可能である．具体的には，返却値が必ず undefined となる void 演算子を利用して，開発者が記述した文字列を無効化し，スクリプトコードを埋め込むことが可能である．したがって，8, 9 行目のようにパラメータを設定することで，11 行目以降のコードが生成，挿入され，攻撃者が意図する悪意のあるスクリプトの実行が可能となる．このように複数のパラメータに跨ってスクリプトを記述することで，入力が複雑化するため，XSS のパターンを検出する XSS フィルタなどが正しく動作しない可能性がある．

```
1  var id = getParamFromURL("id");
2  var name = getParamFromURL("name");
3  var code = "<iframe id='" + id + "'";
4      code += " name='" + name + "' src=''></iframe>";
5  document.writeln(code);
6
7  // 攻撃
8  id = "'/><script>void('"
9  name = "');alert(1)</script>"
10 // 生成されるコード
11 <iframe id=''/>
12 <script>void(' name=');alert(1)</script>
13 src=''></iframe>
```

図 6: 複数のパラメータを利用した Client-Side XSS の脆弱性 [21]

2.3 サードパーティによる脆弱性と従来の XSS 防御策の回避

Client-Side XSS に対する脆弱性は Web アプリケーションの開発者による JavaScript コードだけでなく，サードパーティの JavaScript ライブラリにも存在する可能性がある．Stock らの調査 [22] では，Alexa Top 10000 のドメイン上に存在した 1,273 個の Client-Side XSS 脆弱性のデータセットを分析し，内 273 個はサードパーティのライブラリが原因であったと示している．加えて，Lekies ら [7] は，Script Gadgets を使用することで，サードパーティライブラリに含まれる脆弱性を用いて Client-Side XSS が可能だと示している．

Script Gadgets とは、Web ページやライブラリに存在する正当な JavaScript コードのブロックのことを指すが，悪用することで Client-Side XSS を引き起こす場合が存在する．したがって，Script Gadgets 自体は正当なコードブロックなため，Script Gadgets を利用して Client-Side XSS を引き起こした場合，先述した従来の XSS 防御策である HTML サニタイザや XSS フィルタ，CSP などを回避することが可能だと示している．

図 7 に XSS Auditor を回避する Dojo Toolkit に存在する Script Gadgets を利用

```
<div
    data-dojo-type="dijit/Declaration"
    data-dojo-props="}-alert(1)-{">
</div>
```

図 7: XSS Auditor を回避する Script Gadgets を利用した例 [7]

したコードの例 [7] を示す．この例では，Dojo Toolkit という広く使われている JavaScript ライブラリに存在する Script Gadgets を利用することで，Web ブラウザである Google Chrome などに採用されている XSS フィルタの XSS Auditor を回避し，Client-Side XSS を行うことが可能である．この Script Gadgets は 2020 年 1 月現在，最新の Google Chrome 79.0.3945.117 で再現可能であり，ライブラリの修正はいまだに行われていない．実際に XSS Auditor を回避した例が図 8 である．

このような ScriptGadgets の例や，第 2.2 節で先述した例のように，Client-Side XSS では攻撃に利用される入力が複雑なことが多く，従来の XSS 対策では検知や防御が難しいため，従来のものとは別に Client-Side XSS に向けた防御策が必要だとされている．

2.4 Client-Side XSS の攻撃モデル

図 9 に Client-Side XSS の標準的な攻撃モデルを示す．Client-Side XSS は，攻撃者があらかじめ Client-Side XSS の脆弱性を含んだ Web サイトの情報を知った上で，

1. 被害者にスクリプトを埋め込んだ URL を送りつける
2. スクリプトが埋め込まれた HTTP リクエストが脆弱な Web サイトに送信される
3. HTTP レスポンスが返信される

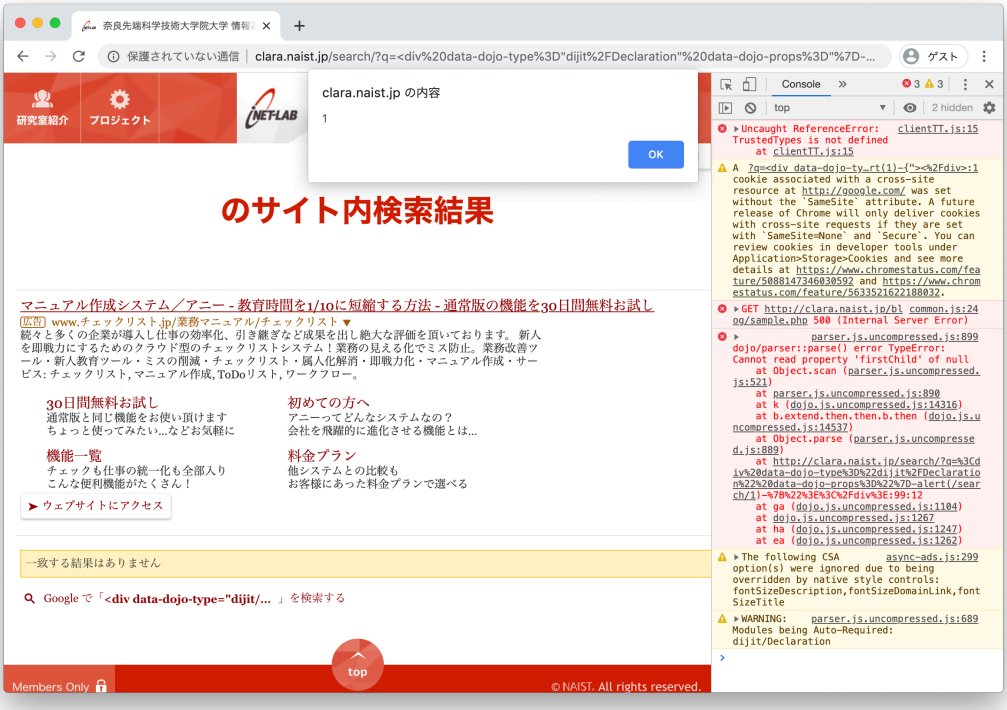

図 8: Google Chrome 79.0.3945.117 で XSS Auditor を回避した例

4. Web ブラウザでスクリプトが実行され，被害者の Cookie が攻撃者のサーバに送信される

といった流れで攻撃が行われ，最後の段階では，攻撃者が用意した Web サーバに自動で移動させられ，URL のパラメータに元の Web サイトの Cookie が付加されるため攻撃者がその Cookie を読み取り，セッションハイジャックが可能となる．

従来の XSS 防御策では，(1) 以前にサーバ側でスクリプトの埋め込みを防ぐことに重点を置いていたが，Client-Side XSS では，埋め込まれたスクリプトコードのログがサーバに残らない場合もあり，サーバ側に施す従来の XSS 防御策では対応が難しい．また，スクリプトが構成されるのがサーバではなく Web ブラウザとなるため，被害者であるクライアント側にて防御することが求められている．

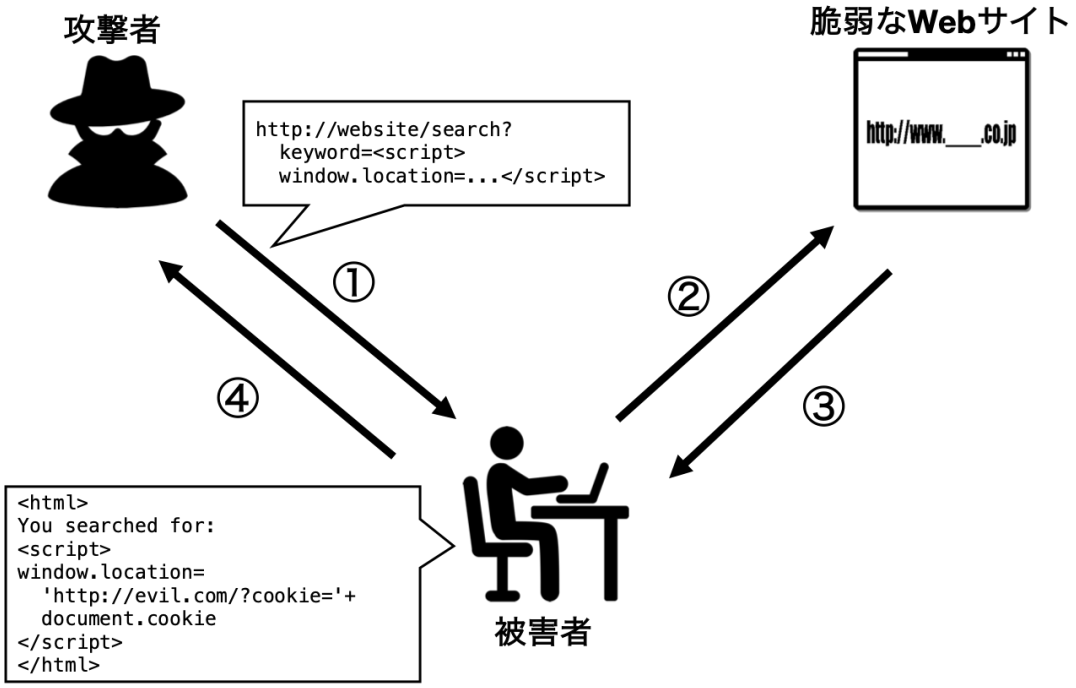

図 9: Client-Side XSS の標準的な攻撃モデル

3. 関連研究

Client-Side XSS に対する検知・防御策に関する研究について紹介する．ここでは，Client-Side XSS の直接的な要因となり得る，文字列の操作や型に焦点を当てたものをまとめる．

3.1 テイントトラッキングによる検知・防御

Stock ら [21] の研究では、JavaScript で取り扱う文字列全てにテイントを付加し，そのテイントを追跡することで Client-Side XSS の検知・防御を行う手法が提案されている．

この研究では，Google Chrome などに搭載されている XSS フィルタの XSS Auditor では，現存する Client-Side XSS の脆弱性の 73%が防御不能だと示した上で，クライアントに施す防御手法を提案している．具体的には，JavaScript エンジンを直接改変することによって，JavaScript ソースの構文解析の間に，全ての文字列にテイントを付加させる．付加されたテイントは文字列の操作や結合があった場合でも，伝播する仕組みとなっているため，ある文字列が攻撃者が制御可能とする不正な起源としたものか否か判断が可能となる．したがって，定義されている実行シンクに文字列が格納される場合に，不正な起源を検知し，実行を終了させることが可能な上，テストデータを意図的に挿入することで実行シンクに到達するか調べることで，不正なデータフローが Web アプリケーションに存在するか確認が可能である．

一方で，本来 JavaScript エンジンには存在しない機能であるテイントトラッキングが文字列の定義や操作の度に動作するため，通常の処理速度に比べて 7〜17%のオーバーヘッドが生じることや，テイントが付加された文字列を，JSON.parse() が許容せず例外処理が必要なことなどの課題が挙げられている．

3.2 安全なコードに置換するパッチを自動生成することによる防御策

Parameshwaran ら [14] の研究では，安全でないコードを安全なコードに置換するパッチを自動生成することによる Client-Side XSS の防御策が提案されている．

この手法では，Web にアクセスする際はプロキシサーバを経由させ，そのプロキシサーバにおいてユーザがアクセスしようとした Web サイトを，テイントトラッキングを利用して Client-Side XSS の脆弱性分析を行う．分析の結果，脆弱性を含んでいるコードと，そのコードを安全なコードに置換したものを生成してデータベースに保存する．図 10 に安全でないコードを安全なコードに置換する例 [14] を示す．(a) が Client-Side XSS の脆弱性を含む安全でないコードで，(b) が (a) を安全なコードに置換した結果である．

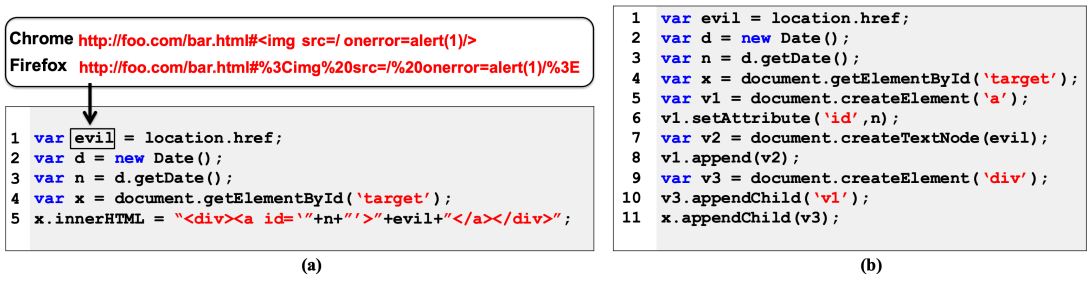

図 10: 安全でないコードを安全なコードに置換する例 [14]

そしてデータベースをもとにパッチを生成し，プロキシサーバに保存している Web サイトのファイルに適用させ，ユーザは安全なプロキシサーバ上のファイルを閲覧する．この手法により，ユーザが利用するブラウザに依存せず，Client-Side XSS の脆弱性を塞ぐことを可能とする．

一方で，テイントトラッキングによる脆弱性分析とパッチを適用させる処理のオーバーヘッドが 5%以上生じることや，プロキシサーバやデータベースを配置しなければならず，導入が容易でないことが課題として挙げられる．

3.3 サードパーティによる実行シンクの呼び出しを検知

Musch ら [12] は，Client-Side XSS の脆弱性の大きな割合がサードパーティライブラリが原因であることに着目し，Client-Side XSS の直接的な原因となる実行シンクにおいて呼び出し元を検査し，呼び出し元がサードパーティである場合に対策を実行する防御策が提案されている．

現代の Web アプリケーションは，サードパーティホストからのスクリプトコードを利用している場合が非常に多く，実際に Alexa Top 5000 をクロールして調査した結果 99%以上のサイトがサードパーティコードを含んでいたと示しており，サードパーティが Client-Side XSS の原因となる場合，Web アプリケーション開発者がファーストパーティのコードをどれだけ強固にしても脆弱性が生じることとなる．

この研究では，Web ブラウザや JavaScript エンジンに直接実装するのではなく，ScriptProtect と呼ばれる防御のための JavaScript ファイルを作成し，ページの最初にその JavaScript ファイルを読み込ませることで防御する．ScriptProtect で行われる処理は，主に実行シンクのフックによる防御とスタックトレースによる実行シンクの呼び出し元検査の 2 つである．

図 11 に実行シンクをフックする例を示す．この例では，代表的な実行シンクである `document.write` と `document.writeln` をフックし，引数をサニタイズする処理を加えている．

図 12 にスタックトレースによる実行シンクの呼び出し元検査を示す．Script-Protect では，JavaScript エンジンがエラーを示す時に使用されるスタックトレースを利用し，実行シンクの最初の呼び出し元が Web アプリケーションのホストネームと一致するかどうかを検査する．一致しない場合は実行を許可せず，一致した場合でも図 11 のように引数が全てサニタイズされた上で実行する．この Script-Protect により，ブラウザに依存せず，サードパーティによる Client-Side XSS を防ぐことが可能になる．

一方で課題として，サードパーティが原因となる脆弱性しか防げないほか，サニタイズによる負荷と DOM の動的更新に不都合が生じることが挙げられる．また，エラー時のスタックトレースを使用しているため，呼び出し元情報の信頼性

```
(function() {
    let old = {};
    function wrap(name) {
        old[name] = document[name];
        document[name] = function() {
            arguments = sanitizeAll(arguments);
            old[name]call(document, ...arguments);
        };
    }
    wrap("write"); wrap("writeln");
})();
```

図 11: 実行シンクをフックする例

```
function isAllowed() {
    //Extract all URLs from the stack trace
    var regex = /(https?:\/\/.+?):\d+:\d+/g;
    var urls = (new Error).stack.match(regex);

    if (urls && urls.length > 0) {
        //Use last entry and extrace its hostname from URL
        var topCaller = getHost(urls[urls.length - 1]);
        return topCaller == location.hostname;
    }
    return true;
}
```

図 12: スタックトレースによる実行シンクの呼び出し元検査

が低いことも課題として挙げられる.

```
1  // Content-Security-Policy: trusted-types mypolicy
2
3  const escapePolicy = TrustedTypes.createPolicy('mypolicy', {
4  createHTML: (unsafe) => {
5    return unsafe
6      .replace(/&/g, "&amp;")
7      .replace(/</g, "&lt;")
8      .replace(/>/g, "&gt;")
9    }
10 })
11 let str = "unsafe";
12 let trusted = myPolicy.createHTML(str);
13
14 document.body.innerHTML = str; // error
15 document.body.innerHTML = trusted; // success
```

図 13: Trusted Types の例

3.4 Trusted Types による Client-Side XSS の予防

Kotowicz ら [6] は，実行シンクに渡される文字列が安全な文字列であるか型で検証する，Trusted Types の提案を行っている．

この Trusted Types では，対応するコンテンツセキュリティポリシー (以下 CSP) を設定した場合，innerHTML や document.write などの実行シンクに通常の文字列が格納できなくなり，Trusted Types と呼ばれるオブジェクトのみ格納を許可する仕組みとなっている．

図 13 に Trusted Types の例を示す．既にサーバ側で 1 行目のように Trusted Types に対応する CSP の設定を行っている場合，実行シンクに通常の文字列が格納できなくなる．したがって，14 行目で通常の文字列を innerHTML に渡しているが，エラーが発生し，DOM は更新されない．Web アプリケーションの開発者が，何らかの理由で実行シンクに文字列を渡して DOM を更新する場合は，3 行目のように開発者が自身でポリシーを作成・設定し，そのポリシーに基づいて文

字列を Trusted Types と呼ばれるオブジェクトにすることで，DOM の更新が可能になる．

この Trusted Types を適用させることで、通常の文字列でさえ実行シンクに格納することが不可能になるため，Client-Side XSS の脆弱性が生じる可能性を限りなく減らすことが出来るとされているほか、防御する際も，文字列の型を検証するだけでよく，パフォーマンスへの影響も少ない手法となっている．

この Trusted Types の課題として，Web アプリケーションの開発方法に手順が追加されるため，開発者への負担が大きくなることと，あくまでも Trusted Types は Web アプリケーション開発時の手法として定義されているため，既存の Web アプリケーションに存在する脆弱性を利用した Client-Side XSS を防ぐことが出来ないことが挙げられる．仮に Trusted Types を既存の Web アプリケーションに適用させる場合でも，サードパーティのライブラリを含むコードの変更が必要で，近年の大規模な Web アプリケーションの変更は現実的でない点が大きな課題となっている．

4. プリミティブな Trusted Types による防御手法の提案

本章では，本研究の目的と要件とプリミティブな Trusted Types による提案手法，Trusted Types の割り当て，許可しない実行シンク，ユースケースのカバーについて述べる．

4.1 研究目的と要件

本研究では，パフォーマンスと既存の Web アプリケーションへの適用性を考慮した Client-Side XSS の防御を目的としている．したがって，

- パフォーマンスへの影響が小さい
- 既存の Web アプリケーションに変更を加えない
- DOM の動的追加・更新が可能

の 3 つを防御手法の要件として定める．

そのため，関連研究で用いられているテイントトラッキングは，文字列に安全であるか示す情報となるテイントを付加することで，不正なデータフローを正確に検知し防御可能だが，テイントを保持する空間や，テイントを文字列間で伝播させる処理が追加されるため，パフォーマンスが低下する恐れがあり，不向きだと考えられる．一方 Trusted Types は，安全であるかは文字列の型で検証することで，パフォーマンスへの影響を小さく防御が可能だが，開発者が明示的にポリシーと信頼できるオブジェクトを生成しなければならないため，既存の Web アプリケーションに適用させるには大幅な変更が必要であり，大きな課題となっていた．実行シンクに格納される文字列を全てサニタイズすることも方法の 1 つとして考えられるが，開発者が意図した DOM の動的追加・更新が行えなくなる可能性があるほか．サニタイズで防げない場合が Client-Side XSS では多く考えられる．

そこで本研究では，この Trusted Types をプリミティブな型として定義することで，既存の Web アプリケーションに変更を加えることなく，Trusted Types を適用させる手法を提案する．そして Trusted Types を利用することで，安全であるか示す情報は型で保持させ，Client-Side XSS の防御に必要な処理は文字列の型を検証するのみとし，DOM の動的追加・更新を可能にした上でパフォーマンスへの影響も最小限に抑える．

以降では，提案手法の概要，Trusted Types の割り当て，およびユースケースのカバーについて述べる．

4.2 プリミティブな Trusted Types

プリミティブな型とは，オブジェクトでなく，メソッドを持たないデータのことを指し，JavaScript では現在，Boolean, Null, Undefined, Number, BigInt, String, Symbol の 7 種類のプリミティブな型が定義されている．そして，JavaScript ではデータ同士の演算はこのプリミティブな型同士でのみ許可されており，演算子のオーバーロードは許可されていない．オブジェクト同士で演算を行う場合は，一度このプリミティブな型に変換された上で計算が行われる．

したがって，Trusted Types をプリミティブな型として定義することで，Web アプリケーションの開発方法で用いられる＋演算子による文字列同士の結合などが，信頼できるかどうかの情報を保持したまま行うことができ，既存の Web アプリケーション上でも動作させることが可能になる．

4.3 Trusted Types の割り当て

プリミティブな Trusted Types の割り当ては，JavaScript エンジンによる JavaScript ソースの解析の間に行う．JavaScript エンジンの動作の大まかな順序は

1. JavaScript ソースの構文解析
2. 抽象構文木の生成
3. コンパイルにより実行する機械語の生成

となる．本研究では，他のプリミティブな型と同様に，構文解析の段階で Trusted Types を割り当てる．構文解析の段階で割り当てることで，従来のように開発者が明示的に Trusted Types を割り当てる必要がなくなるほか，あらかじめ型を割り当てるため，従来の String である文字列を Trusted Types にする処理自体も減らすことが可能になる．

本研究では，Trusted Types を割り当てる文字列の方式を 2 つ定める．

4.3.1 方式 1 : JavaScript ソース中のシングルクォーテーションまたはダブルクオーテーションで囲まれた文字列

```
1 let trusted = "https://example.co.jp/"; // Trusted Types
2 let host = location.host;                // String
3 let hash = location.hash;                // String
4
5 document.writeln(trusted); // success
6 document.writeln(host);    // error
7 document.writeln(hash);    // error
```

図 14: 方式 1 のプリミティブな Trusted Types の例

図 14 に方式 1 のプリミティブな Trusted Types の例を示す．方式 1 では，JavaScript ソース中の開発者が記述した文字列のみ Trusted Types とする．具体的には，JavaScript 中のシングルクォーテーションまたは，ダブルクオーテーションで囲まれた文字列を Trusted Types としてプリミティブな型を割り当てる．したがって，URL などのグローバルオブジェクトに格納されている文字列は，通常の String を割り当てることとなる．そして，従来のものと同様に動的に DOM を更新する実行シンクには Trusted Types のみを許可し，通常の String は許可しない．そのため，図 14 では 6, 7 行目にて innerHTML に渡される文字列が，グローバルオブジェクトである location.hash と location.host を起源とする String のため，格納が許可されない．

4.3.2 方式 2：location.hash などのユーザが操作可能な文字列以外全て

```
1 let trusted = "https://example.co.jp/"; // Trusted Types
2 let host = location.host;                // Trusted Types
3 let hash = location.hash;                // String
4
5 document.writeln(trusted); // success
6 document.writeln(host);    // success
7 document.writeln(hash);    // error
```

図 15: 方式 2 のプリミティブな Trusted Types の例

表 1: 攻撃者が操作可能な入力ソース

URL	document.location
baseURI	location.hash
documentURI	location.search
window.locaiton	location.href

図 15 に方式 2 のプリミティブな Trusted Types の例を示す．方式 2 では，攻撃者が操作可能だとされるものを定義し，それ以外全ての文字列を Trusted Types とする．具体的には，表 1 に示す攻撃者が操作可能な入力ソースをあらかじめ定義しておき，それらは通常の String を割り当て，入力ソース以外の文字列を Trusted Types としてプリミティブな型を割り当てる．したがって，方式 1 とは異なり，グローバルオブジェクトに格納されている文字列も，Trusted Types が割り当たる場合が存在する．そのため，図 15 では 7 行目にて innerHTML に渡される文字列が，入力ソースである location.hash を起源とするため格納が許可されないが，6 行目はグローバルオブジェクトであるが，攻撃者が操作可能でないとする location.host を起源とするため，格納が許可される．

4.4 許可しない実行シンク

従来の Trusted Types と同様に，DOM を動的に操作可能で危険な実行シンクは，プリミティブな Trusted Types のみを許可する．

表2に許可しない実行シンクを示す．許可しない実行シンクには，従来の Trusted Types で定められているものに加え，`eval` と `HTMLParamElement.value` を追加する．

表 2: 許可しない実行シンク

`document.write`	`HTMLIFrameElement.src`
`document.writeln`	`HTMLIFrameElement.srcdoc`
`document.open`	`HTMLImageElement.src`
`location.href`	`HTMLInputElement.formAction`
`location.replace`	`HTMLObjectElement.data`
`location.assign`	`HTMLObjectElement.codeBase`
`window.open`	`HTMLParamElement.value`
`Element.innerHTML`	`HTMLScriptElement.src`
`Element.outerHTML`	`HTMLScriptElement.text`
`Element.setAttribute`	`HTMLScriptElement.textContent`
`Element.setAttributeNS`	`HTMLScriptElement.innerText`
`Element.insertAdjacentHTML`	`HTMLTag.onEventName`
`HTMLAnchorElement.href`	`Range.createContextualFragment`
`HTMLAreaElement.href`	`DOMParser.parseFromString`
`HTMLBaseElement.href`	`setTimeout`
`HTMLButtonElement.formAction`	`setInterval`
`HTMLEmbedElement.src`	`eval`
`HTMLFormElement.action`	

4.5 ユースケースのカバー

プリミティブな Trusted Types を新たに定義することで，JavaScript エンジン中に文字列を扱う型が複数になるなど変更が加わるため，従来のユースケースをカバーする必要が生じる．ここでは代表的なものを例に挙げる．

4.5.1 typeof の出力

JavaScript には，オペランドの型の名前を示す typeof 演算子が存在する．

```
1 typeof 'naist'  // 'string'
2 typeof location.hash  // 'string'
3
4 typeof 'naist' == 'string' // True
5 typeof location.hash == 'string' // True
```

図 16: typeof の出力例

図 16 に typeof の出力例を示す．提案手法により，プリミティブな文字列の型が複数存在するが，プリミティブな Trusted Types と従来の文字列のどちらも，出力は変更せず `string` とする．同様に比較演算子で文字列 `string` と比較された場合も True とする．

4.5.2 String と Trusted Types が結合される場合

上述した通り，JavaScript ソース内では，＋演算子による文字列の結合が頻繁に用いられている．提案手法によりプリミティブな Trusted Types を定義した場合，文字列の型が 2 種類存在するため，これら同士が結合される可能性がある．

図 17 に String と Trusted Types が結合する例を示す．この例では，1 行目で Trusted Types の変数が定義され，4 行目でその変数にグローバルオブジェクトである location.hash が結合されている．この例のように Trusted Types と String が結合された場合，結果となる文字列は String とし，実行シンクへの格納は許可しない．

```
let name = "Your name is ";
document.write(name);  // success

name += location.hash;
document.write(name);  // error
```

図 17: String と Trusted Types が結合する例

4.5.3 ラッパーオブジェクトのメソッドを使用する場合

プリミティブな型にはメソッドが存在しないが，JavaScript における文字列は，対応するラッパーオブジェクトである String オブジェクトに自動変換させることでメソッドを呼び出すことが可能である．

```
let trusted = " Excepteur sint occaecat";
trusted = trusted.trim();
trusted = trusted.substr(0, 9);
document.body.outrHTML = trusted; // success

let string = trusted.concat(location.search);
document.body.outrHTML = string; // error
```

図 18: ラッパーオブジェクトのメソッドを使用する例

図 18 にラッパーオブジェクトのメソッドを使用する例を示す．従来の文字列と同様に，Trusted Types にも対応するラッパーオブジェクトを定義し，String オブジェクトと同様のメソッドやプロパティを利用可能とする．これにより，既存の Web アプリケーションに影響を及ぼさないようにするほか，開発者が String との差を意識しないようにする．そのため，定義する Trusted Types のラッパーオブジェクトのメソッドの返却値は基本的に Trusted Types とし，図 18 の例のように実行シンクに格納することを許可する．例外として，concat() などのように，文字列の引数が指定可能でその引数が String であった場合，返却値は String とする．

4.5.4 サードパーティのライブラリを利用する場合

Client-Side XSS に対する脆弱性は，サードパーティのライブラリが原因となるものも多く存在することが知られている．そのため，サードパーティのライブラリを利用している場合でも，Trusted Types を適用させる．

```
<script src="jquery.min.js"></script>

<script>
let trusted = "Lorem ipsum dolor sit amet";
let string = location.search;

$("#div").innerHTML = trusted; // success
$("#div").innerHTML = string;  // error
</script>
```

図 19: サードパーティのライブラリを利用する例

図 19 にサードパーティのライブラリを利用する例を示す．この例のように，サードパーティのライブラリが利用されている場合でも，最終的に実行シンクに文字列が格納される場合，文字列の型を検証することで防御する．プリミティブな型として定義しているため，サードパーティのライブラリにも変更を加えず Trusted Types を適用させることが可能となる．

5. JavaScript エンジンと Web ブラウザへの提案手法の実装

提案手法の実装は，プリミティブな Trusted Types の定義・割り当てと，実行シンクに渡される文字列は Trusted Types のみを許可するものの 2 つを分けて行い，それぞれ JavaScript エンジンと Web ブラウザに実装を行う．

5.1 プリミティブな Trusted Types の実装

Trusted Types の定義と割り当ては，オープンソースソフトウェアの JavaScript エンジンである V8[16] バージョン 7.7.299.11 に実装する．

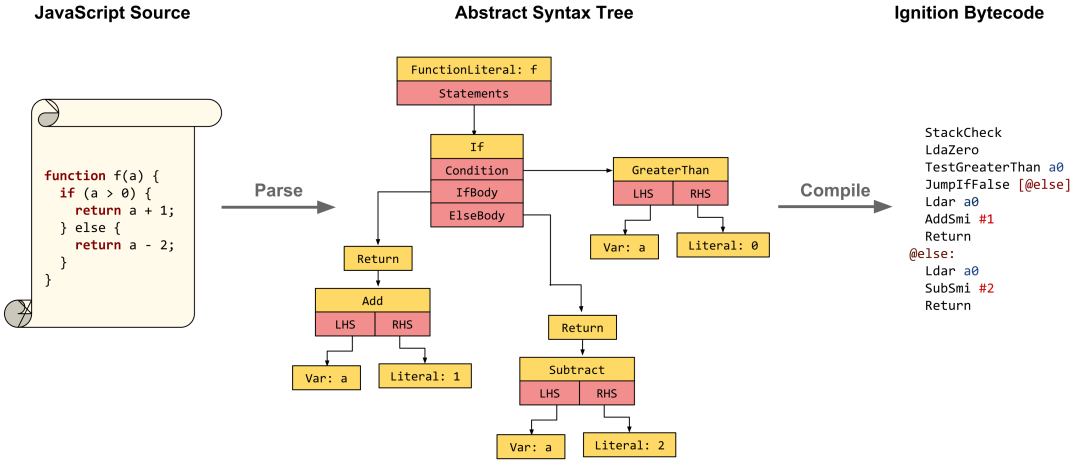

図 20: V8 による JavaScript 実行の流れ [9]

図 20 に V8 による JavaScript 実行の流れ [9] を示す．第 4.3 節で先述した通り，JavaScript エンジンの動作の大まかな順序は

1. JavaScript ソースの構文解析
2. 抽象構文木の生成
3. コンパイルにより実行する機械語の生成

であり，V8 も基本的にその順序に則っている．

本研究では，型の割り当ての実装として，最初の段階である JavaScript ソースの構文解析の段階で割り当てを行い，従来のもののように後に通常の String を新たに Trusted Types に変更させるなどの処理を減らす．またテイントトラッキングのアンテイントのように型を戻すことは無く，文字列は型を保持し続けるようにする．プリミティブな Trusted Types の割り当て方は，先述した通り 2 方式行い，方式 1 は Web ブラウザによる JavaScript エンジンの呼び出しに変更を加えることで，Web ブラウザによる JavaScript の初期化と，開発者が記述した JavaScript ソースの解析処理を分け，JavaScript ソース中の文字列にのみ Trusted Types の割り当てを行う．方式 2 は，Web ブラウザによる JavaScript の初期化の際も Trusted Types が割り当てられる場合があるため，表 1 の攻撃者が操作可能な入力ソースのみ抽出して通常の String を割り当て，文字列のデフォルトの型を Trusted Types として振る舞うよう変更を加える．

また V8 への実装は，既存の Web アプリケーションにおける文字列で用いられるメソッドなどを動作させるため，文字列のラッパーオブジェクトを再定義し，通常の String かプリミティブな Trusted Types のどちらから呼び出されたかを識別して，それに応じて返却値を変更するよう実装する．

5.2 実行シンク制限の実装

実行シンクへの制限は，オープンソースソフトウェアの Web ブラウザである Chromium[18] バージョン 77.0.3865.90 に実装する．実装の内容として，Chromium にあらかじめ実装されている従来の Trusted Types に変更を加え，CSP の設定無しで実行シンクは Trusted Types のみを許可する設定とする．また，従来のものは実行シンクの種類によって許可するオブジェクトは TrustedHTML や TrustedScript などに分けられているが，本研究では Trusted Types の 1 つのみを許可する文字列として定義する．

図 21，図 22 に提案手法の 2 つの方式を実装した Web ブラウザをそれぞれ示す．表示している内容は，図 14，図 15 に記載した JavaScript を実行するページである．このように定義した実行シンクに Trusted Types でない文字列が格納さ

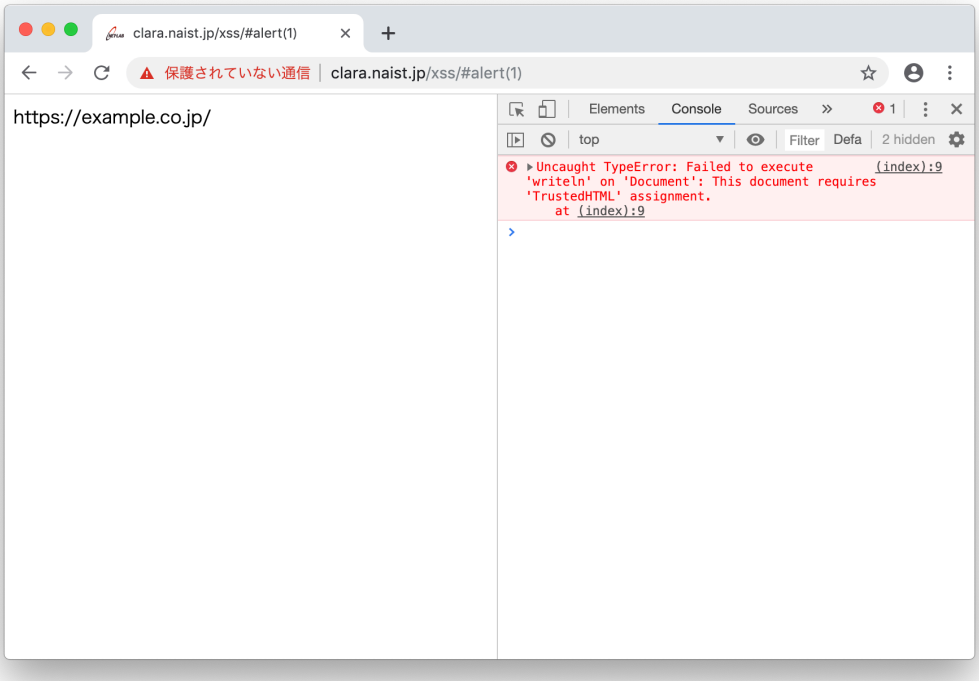

図 21: 提案手法を実装した Web ブラウザ (方式 1)

れる場合，ブラウザのコンソール画面にエラーを表示し，実際に格納もされない．また，方式の差として，図 21 では `location.host` が Trusted Types ではないため，`document.writeln` が失敗しホスト名が表示されていないが，図 22 では `location.host` が Trusted Types であるため，ホスト名が表示されていることが分かる．

次に，図 23 に方式 1 の提案手法を実装した Web ブラウザで Script Gadgets の悪用を防御した例を示す．先述した Script Gadgets を悪用する例は，文字列が実行シンクに格納されてから効果を発揮するものがほとんどのため，文字列の格納を許可しないことで防御できていることが分かる．また方式 2 でも同様に防御できることが確認できている．

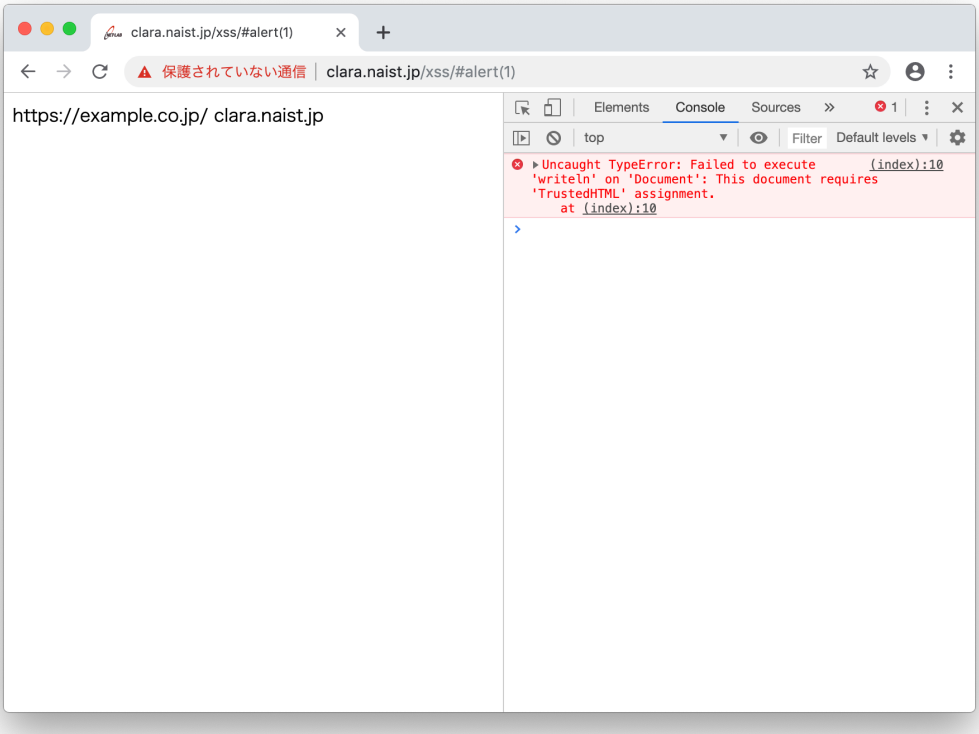

図 22: 提案手法を実装した Web ブラウザ (方式 2)

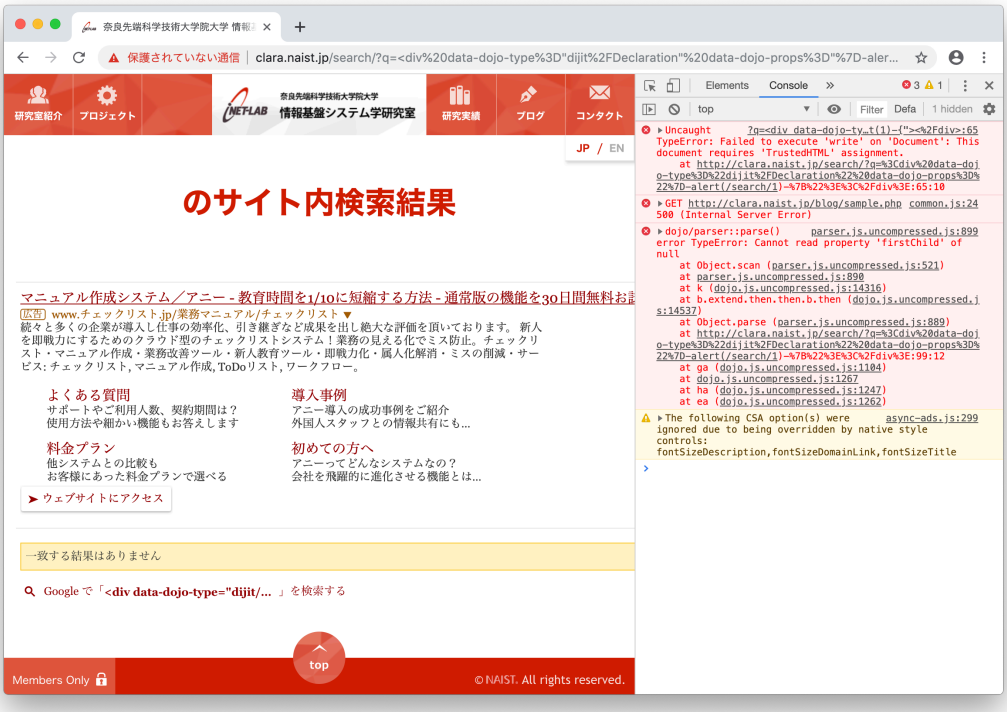

図 23: 提案手法を実装した Web ブラウザで Script Gadgets の悪用を防御した例

6. 実験と評価

本章では，本研究の Client-Side XSS の防御手法の評価項目と実験環境，評価の結果について述べる．

6.1 評価項目

本研究ではパフォーマンスと既存の Web アプリケーションへの適用性を考慮した Client-Side XSS の防御を目的としており，

- パフォーマンスへの影響が小さい
- 既存の Web アプリケーションに変更を加えない
- DOM の動的追加・更新が可能

の 3 つを防御手法の要件として定めている．第 5 章で述べたように，本研究の提案手法は JavaScript エンジンと Web ブラウザへの実装を行なったため，実装した Web ブラウザを利用するだけで防御を可能とし，既存の Web アプリケーションに変更を加えないという要件を満たしている．DOM の動的追加・更新が可能という要件についても，Trusted Types を用いることで，文字のエスケープやサニタイズを行わずに文字列の型で防御を可能とするので満たしていると考えられる．したがって，残る要件を満たすために行う提案手法の評価項目として，

- JavaScript のパフォーマンスの変化

を定め，既存手法との比較や，Client-Side XSS の防御手法としての性能を計測するために

- Client-Side XSS による攻撃に対する防御率
- 既存の Web アプリケーションへの影響

の 2 つを定め，合計 3 つの項目で評価する．提案手法において，プリミティブな Trusted Types の割り当てを 2 方式定めているが，その両方でこの 3 項目を評価する．

6.2 実験環境

実験環境として，提案手法の2つの方式を実装したChromiumバージョン77.0.3865.90を，表3に示す性能のコンピュータで動作させる．

表 3: 実験で利用するコンピュータの性能

OS	メモリ	CPU
macOS 10.14.6	16GB	Intel Core i7 2.8GHz

JavaScriptのパフォーマンスの変化は，JavaScriptベンチマークツールを用いて評価するが，ベンチマークツールによって評価する項目が異なるため，複数用いる．JavaScriptベンチマークツールは，WebKit開発プロジェクトが開発したJetStream[17], Google社が開発したOctane[3], Mozillaプロジェクトが開発したKraken[11]の3つを使用する．

防御率評価のために，Client-Side XSS脆弱性のデータセットを作成した．脆弱性のデータセットは，自身で作成したものが63個，古いサードパーティライブラリの脆弱性やGoogleが提供しているXSSのテストベッド[2]などから35個，JavaScript静的解析ツールJSPrime[15]で使用されたテストケースから45個の合計143個である．

既存のWebアプリケーションへの影響は，Alexa Top 500のトップページに実装したブラウザでアクセスし，提案手法で実行シンクへの文字列の格納をブロックした数を計測する．そして，提案手法の偽陽性率を計測するために，Alexa Top 500のトップページの全ての実行シンクへの文字列の格納数の合計を計測する．Alexa Top 500へは通常のアクセスを行なっており攻撃ではないため，格納数全体に対する提案手法によるブロック数の割合を偽陽性率とする．

6.3 評価結果

6.3.1 パフォーマンスの変化

Chromium 77.0.3865.90，方式 1 の提案手法を実装した Chromium，方式 2 の提案手法を実装した Chromium を用いて，JetStream，Octane，Kraken それぞれで 50 回ずつ計測を行った．JetStream と Octane はスコアが高い方が好スコアとなり，Kraken はスコアが低い方が好スコアを示す．

表 4: JetStream のスコア平均と性能劣化率

	平均 (高い値が好スコア)	標準偏差	性能劣化率
Chromium 77.0.3865.90	128.180	0.646	-
提案手法 (方式 1)	126.643	0.794	1.012
提案手法 (方式 2)	126.652	0.717	1.012

図 24，表 4 に JesStream を用いたスコアの計測結果と平均，性能劣化率を示す．JetStream のスコア計測提案手法を実装していない Chromium のスコア平均が 128.180 に対して，提案手法の方式 1 と方式 2 のスコア平均はそれぞれ 126.643 と 126.652 となった．従来の Chromium と方式 1，従来の Chromium と方式 2 の 2 つの組み合わせで，有意水準 $\alpha = 0.05$ において対応のある t 検定を行い，オーバーヘッドが生じているか検定すると，p 値はそれぞれ 2.924×10^{-16} と 2.411×10^{-16} となり，有意水準 α を下回っていることから，2 つの方式の両方でオーバーヘッドが生じていることが分かった．したがって，オーバーヘッドとなる性能劣化率は 2 方式ともに 1.012 なため，約 1%のオーバーヘッドが生じていることを示す結果となった．

図 25，表 5 に Octane を用いたスコアの計測結果と平均，性能劣化率を示す．提案手法を実装していない Chromium のスコア平均が 38032.560 に対して，提案手法の方式 1 と方式 2 のスコア平均はそれぞれ 38085.260 と 38200.720 となり，提案手法を実装した Chromium が従来の Chromium よりスコアが高い結果となった．従来の Chromium と方式 1，従来の Chromium と方式 2 の 2 つの組み合わせで，

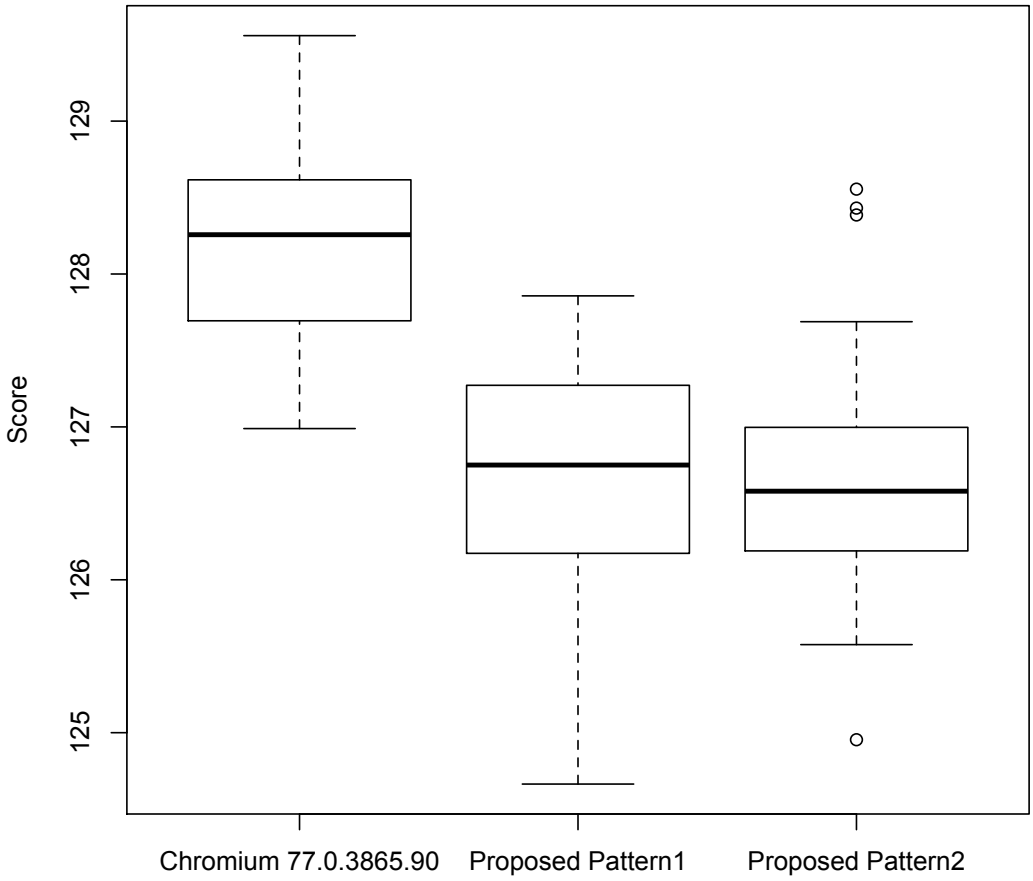

図 24: JetStream のスコア計測結果

有意水準 $\alpha = 0.05$ において対応のある t 検定を行い，オーバーヘッドが生じているか検定すると，p 値はそれぞれ 0.8567 と 0.486 となり，有意水準 α を上回っていることから，2 つの方式の両方でオーバーヘッドが生じていないことを示す結果となった．

図 26，表 6 に Kraken を用いたスコアの計測結果と平均，性能劣化率を示す．提案手法を実装していない Chromium のスコア平均が 999.434 に対して，提案手法

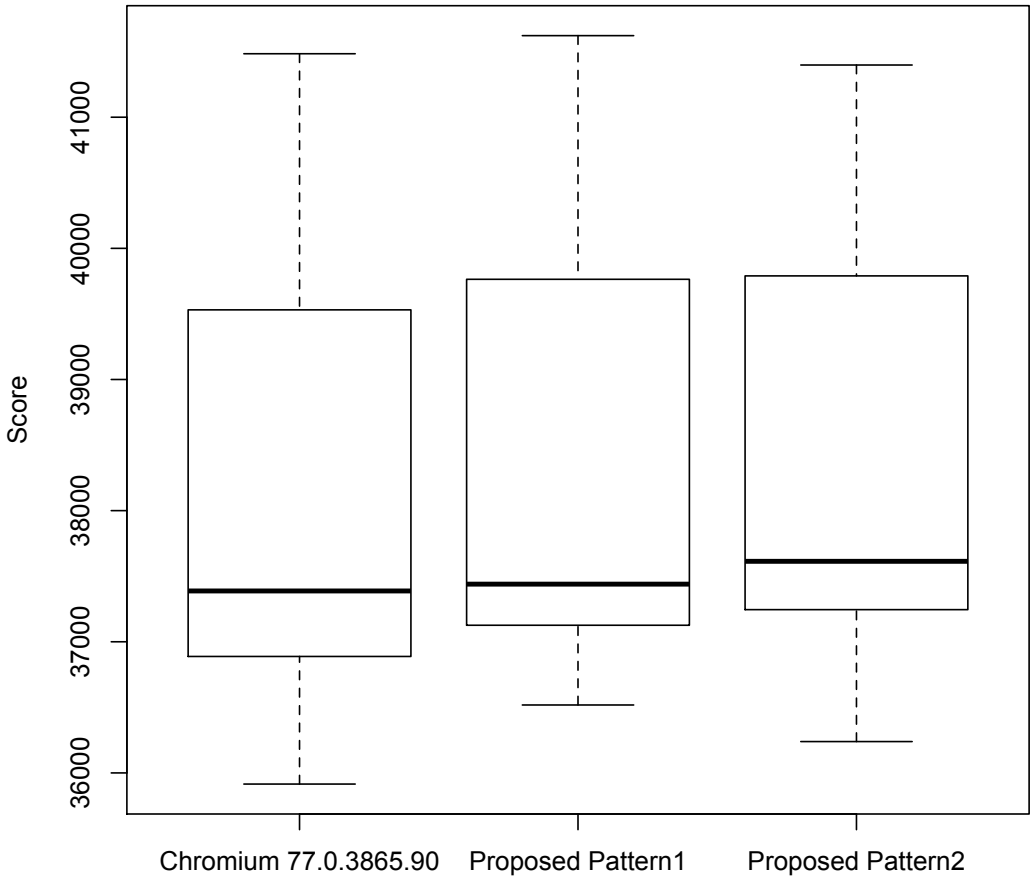

図 25: Octane のスコア計測結果

の方式 1 と方式 2 のスコア平均はそれぞれ 999.258 と 1001.848 となり，方式 1 の提案手法を実装した Chromium が従来の Chromium よりスコアが高い結果となった．従来の Chromium と方式 1，従来の Chromium と方式 2 の 2 つの組み合わせで，有意水準 $\alpha = 0.05$ において対応のある t 検定を行い，オーバーヘッドが生じているか検定すると，p 値はそれぞれ 0.8237 と 0.008377 となり，方式 1 では有意水準 α を上回っていることから，オーバーヘッドが生じていないが，方式 2 では

表 5: Octane のスコア平均と性能劣化率

	平均 (高い値が好スコア)	標準偏差	性能劣化率
Chromium 77.0.3865.90	38032.560	1490.341	-
提案手法 (方式 1)	38085.260	1388.390	0.998
提案手法 (方式 2)	38200.720	1357.460	0.995

表 6: Kraken のスコア平均と性能劣化率

	平均 (低い値が好スコア)	標準偏差	性能劣化率
Chromium 77.0.3865.90	999.434	4.234	-
提案手法 (方式 1)	999.258	4.667	0.999
提案手法 (方式 2)	1001.848	4.040	1.004

下回っていることからオーバーヘッドが生じていることが分かった．したがって，方式 2 のオーバーヘッドとなる性能劣化率はともに 1.004 なため，約 0.4%のオーバーヘッドが生じていることを示す結果となった．

パフォーマンスの変化に関する詳細な考察は第 7.1 節で述べる．

6.3.2 防御率

表 7: Client-Side XSS の脆弱性に対する提案手法で防御に成功した数

	自作	テストベッドなど	JSPrime	合計
提案手法 (方式 1)	63/63	35/35	45/45	143/143
提案手法 (方式 2)	63/63	34/35	45/45	142/143

表 7 に Client-Side XSS の脆弱性に対して，方式 1 の提案手法を実装した Chromium，方式 2 の提案手法を実装した Chromium を用いて防御に成功した数を示す．合計

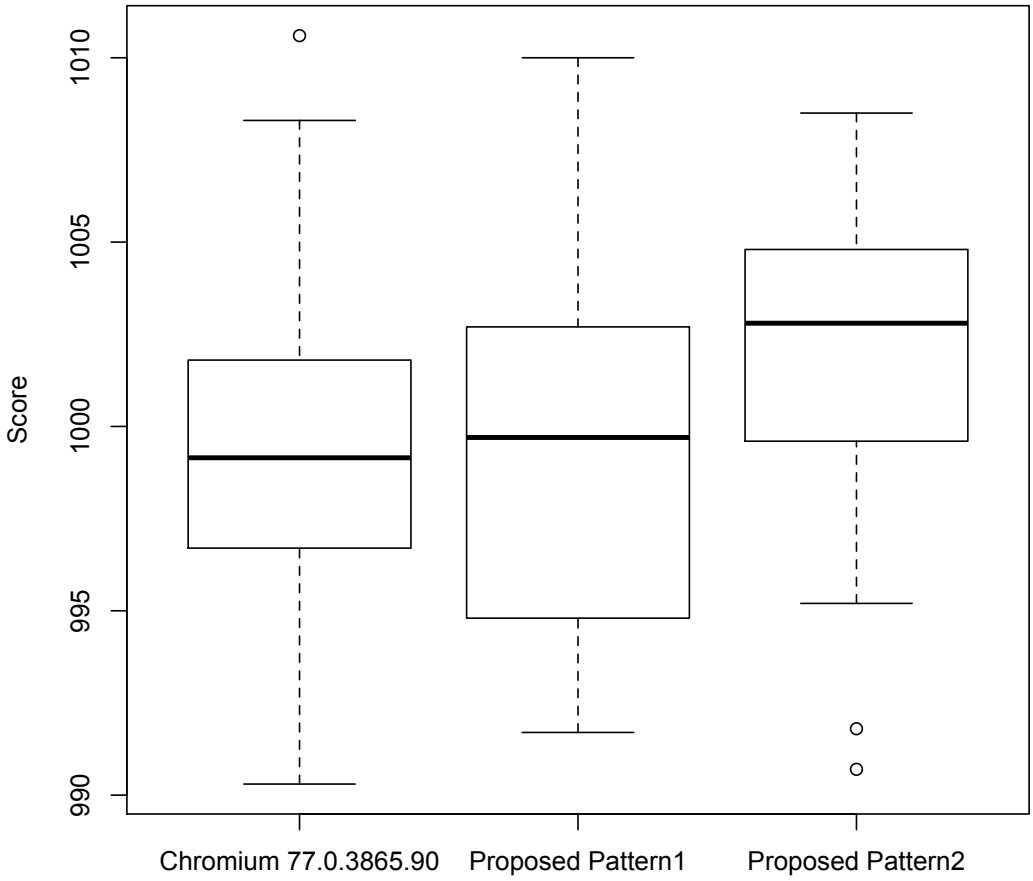

図 26: Kraken のスコア計測結果

143 個の Client-Side XSS の脆弱性に対し，方式 1 の提案手法は 143 個全て，方式 2 の提案手法はテストベッドに存在する 1 個を除いて防御に成功した．

防御率に関する詳細な考察は第 7.2 節で述べる．

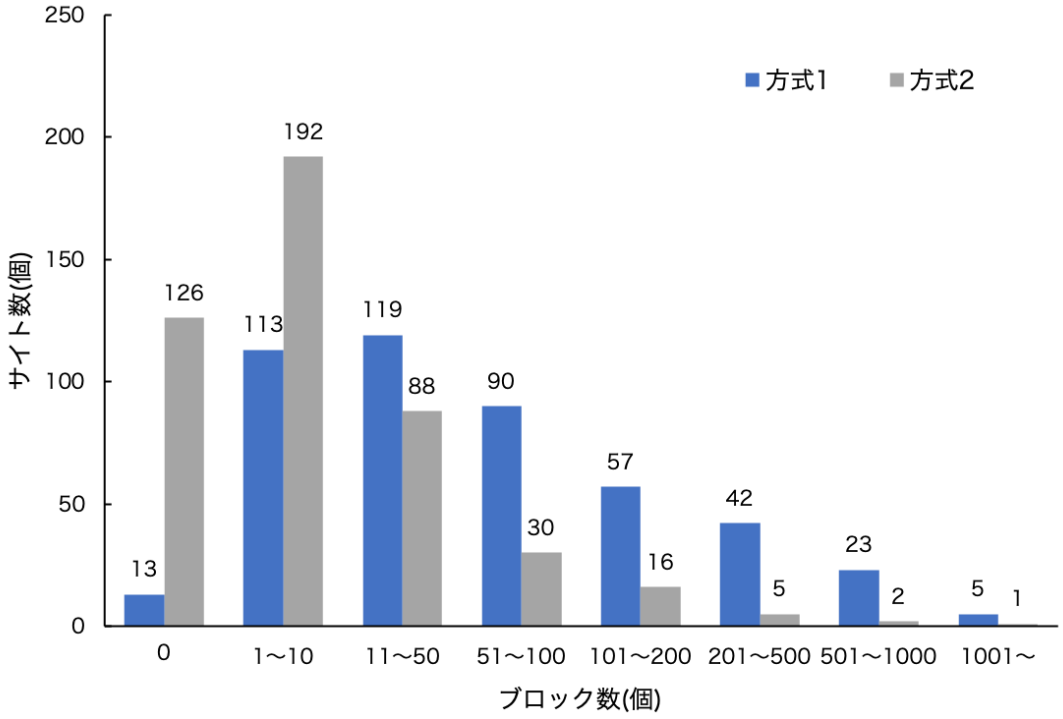

図 27: Alexa Top 500 にアクセスした際のブロック数

6.3.3 既存の Web アプリケーションへの影響

図 27 に提案手法を実装したブラウザで Alexa Top 500 にアクセスした際のブロック数を示す．Alexa Top 500 の内，方式 1 を用いたアクセス時には 38 個，方式 2 を用いたアクセス時には 40 個のドメインがセキュリティエラーやアクセスブロック，ドメインエラーなどでアクセス不能であった．

方式 1 の提案手法を実装したブラウザでは，最も多かったブロック数が 11〜50 であり，119 ドメインが該当した．全くブロックをしなかったドメインは 13 個である一方で，127 ドメインが 100 を超えるブロック数となった．

方式 2 の提案手法を実装したブラウザでは，最も多かったブロック数が 1〜10 であり，192 ドメインが該当した．全くブロックをしなかったドメインは 126 個であり，50 を超えるブロックを行なったドメインは 54 個となった．

提案手法のブロック数と偽陽性率を表 8 に示す．Alexa Top 500 で行われる実行シンクへの文字列の格納数は，アクセス不能なドメインを除き，合計で 112652

表 8: 提案手法のブロック数と偽陽性率

	ブロック数	偽陽性率
提案手法 (方式 1)	52012	0.462
提案手法 (方式 2)	12291	0.109

回であり，偽陽性率は格納数全体に対するブロック数の割合となる．したがって，方式 1 の提案手法ではブロック数が 52113 であることから，偽陽性率は約 46%となり，非常に高い偽陽性率となため，既存の Web アプリケーションへの影響が大きいことが示された．

方式 2 の提案手法ではブロック数が 12291 であることから，偽陽性率は約 11%となり，方式 1 と比較して偽陽性率は軽減できているものの高い偽陽性率となり，既存の Web アプリケーションへの影響が小さくないことが示された．

既存の Web アプリケーションへの影響に関する詳細な考察は第 7.3 節で述べる．

7. 考察

本章では，第 6 章で述べたパフォーマンスと防御率，既存の Web アプリケーションへの影響の評価結果に関する考察について述べ，その後既存手法との比較，今後の課題について述べる．

7.1 パフォーマンスに対する考察

表 4，表 5，表 6 に示す通り，3 つのベンチマークツールを使用して計測した従来の Chromium との性能劣化率が，方式 1 の提案手法が 1.012, 0.998, 0.999 であり，方式 2 の提案手法が 1.012, 0.995, 1.004 となった．また，それぞれのベンチマークツールのスコア結果を，従来の Chromium と方式 1，従来の Chromium と方式 2 の 2 つの組み合わせで，有意水準 $\alpha = 0.05$ のもとで対応のある t 検定を行なった結果，JetStream では両方，Kraken では方式 2 でオーバーヘッドが生じており，Octane では両方，Kraken では方式 1 でオーバーヘッドが生じていないことを示す結果となった．一方で，関連研究 [21] と同様に，従来のスコアとの増加分をオーバーヘッドと定義すると，オーバーヘッドが生じているとされた JetStream と Kraken においても約 1%であり，Octane においてはオーバーヘッドが生じていないことが検定で示されたため，提案手法によるオーバーヘッドは非常に小さいと考えられる．第 3.1 節と第 3.2 節で先述した通り，テイントトラッキングを用いる手法で生じるオーバーヘッドが約 7〜17%と 5%以上であることを考えると，パフォーマンスへの影響を大きく削減できていると考えられる．

大きくパフォーマンスへの影響を削減できた要因として，テイントトラッキングなどのように Web ブラウザに搭載されていない機能を新たに追加するのではなく，文字列の型という既に存在する機能を利用したことが挙げられる．特に，本研究で実装を行なった JavaScript エンジンの V8 の場合，一度 Ignition と呼ばれるインタプリタを介してバイトコードにした上でコンパイラに渡されるが，この Ignition の動作から変更を加えたため，全体としてパフォーマンスに大きな影響を与えなかったと考えられる．

7.2 防御率に対する考察

表 7 に示す通り，合計 143 個の Client-Side XSS の脆弱性に対し，方式 1 の提案手法は 143 個全て，方式 2 の提案手法はテストベッドに存在する 1 個を除いて防御に成功した．防御率は方式 1 が 100%, 方式 2 が 99.3%となり，Client-Side XSS の脆弱性に対する防御率に関しては高い水準を保てていることがわかった．

その要因として，今回は評価で使用した Client-Side XSS の脆弱性が自作したものを始め，自身で作成したデータセットであるため，一般的に想定されているような脆弱性が多かったことが考えられる．方式 2 に関しては特に顕著であり，今回のデータセットは，本研究で定義した攻撃者が操作可能な入力ソースを利用したものがほとんどなため，攻撃者が定義されていない入力ソースを利用した場合，防御は不可能である．

```
1 var payload = location.hash.substr(1);
2 window.status = payload;
3 var retrieved_payload = window.status;
4 eval(retrieved_payload);
```

図 28: 方式 2 で防げなかった Client-Side XSS 攻撃

次に，方式 2 で防げなかった Client-Side XSS 攻撃を図 28 に示す．脆弱性を持つスクリプトの内容としては，攻撃者が操作可能な入力ソースとして定義している `location.hash` の値を一度グローバルな属性である `window.status` に格納し，その後制限をかけた実行シンクである `eval` で実行しようとしているものである．この攻撃が防御できなかった原因を調べると，Chromium の動作上，グローバルな属性に文字列を渡すと，一度文字列が新たに再生成された上で格納することが原因であった．通常のグローバルオブジェクトに同様に格納する分には再生成されないが，決められた属性に格納すると再生成されることが分かった．したがって，文字列の型も新たに割り当てられ，デフォルトである Trusted Types が割り当たってしまい，防御出来なかった．当然，この `window.status` を始めとしたグローバルな属性も，攻撃者が操作可能な入力ソースとして定義すれば防御が可能だが，影響が大きいことが考えられる．

その点，方式 1 の提案手法では，JavaScript ソース上のシングルクオーテーションとダブルクオーテーションで囲まれた文字列のみが Trusted Types なため，`window.status` に格納するために再生成された文字列も通常の String であり，防御に成功していた．

7.3 既存の Web アプリケーションへの影響に対する考察

図 27 に示す通り，方式 1 の提案手法が 127 ドメインが 100 を超えるブロック数であった一方で，方式 2 の提案手法では 50 を超えるブロック数が 54 個となった．偽陽性率に関しても，方式 1 の提案手法が約 46%であったのに対し，方式 2 の提案手法が約 11%となった．したがって，Trusted Types の割り当て方の方式によって既存の Web アプリケーションへの影響が大きく変化することが分かった．

方式 1 の提案手法では，1 回でもブロックを行なったドメインが 449/462 となり，割合としては約 97%となった．100 を超えるブロックを行なったドメインは 127/462 となり割合としては約 27%であり，格納数全体に対するブロック数の割合を示す偽陽性率は約 46%と高いことから，既存の Web アプリケーションへの影響が非常に大きく，正常に動作しなくなる可能性が高いと考えられる．その要因として，方式 1 の割り当て方は JavaScript ソース中のシングルクオーテーションとダブルクオーテーションで囲まれた文字列のみを Trusted Types とするため，既存の Web アプリケーションでは他の文字列を実行シンクに代入する場合が普遍的だったと考えられる．考えられるものとしては，JSON などのファイルから読みだした文字列や，Ajax などを用いて取得したレスポンスなどである．

方式 2 の提案手法では，1 回でもブロックを行なったドメインが 334/460 となり，割合としては約 73%となった．100 を超えるブロックを行なったドメインは 24/460 となり割合としては約 5%であり，格納数全体に対するブロック数の割合を示す偽陽性率は約 11%であることから，方式 1 と比較すると既存の Web アプリケーションへの影響は小さいと考えられる．その要因として，方式 2 の割り当て方は表 1 に定義されたものを通常の String とし，それ以外を Trusted Types として振る舞うように割り当てるため，危険である可能性が高い文字列のみをブロックしているからだと考えられる．一方で，攻撃者が今回定義していない入力ソー

スを使用した場合，その攻撃は防御できないことが確実になる．

既存の Web アプリケーションへの影響に関して，両方式に共通する課題として，本研究でブロックしたもの全てが攻撃ではないことが挙げられる．本研究では，Client-Side XSS の防御に主眼を置き，攻撃者が操作可能な入力ソースなどに格納されている文字列は，実際に攻撃であるかを判断せずに全て信頼できない文字列としていたため，偽陽性率が高くなっていると考えられ，実用性を重視する場合は改善が必要となる．

その一例として，方式 2 の提案手法をベースに，URL に?か#が含まれていない場合攻撃ではないと判断し，表 1 に定義した入力ソースも Trusted Types として定義した場合の実験を行なった．

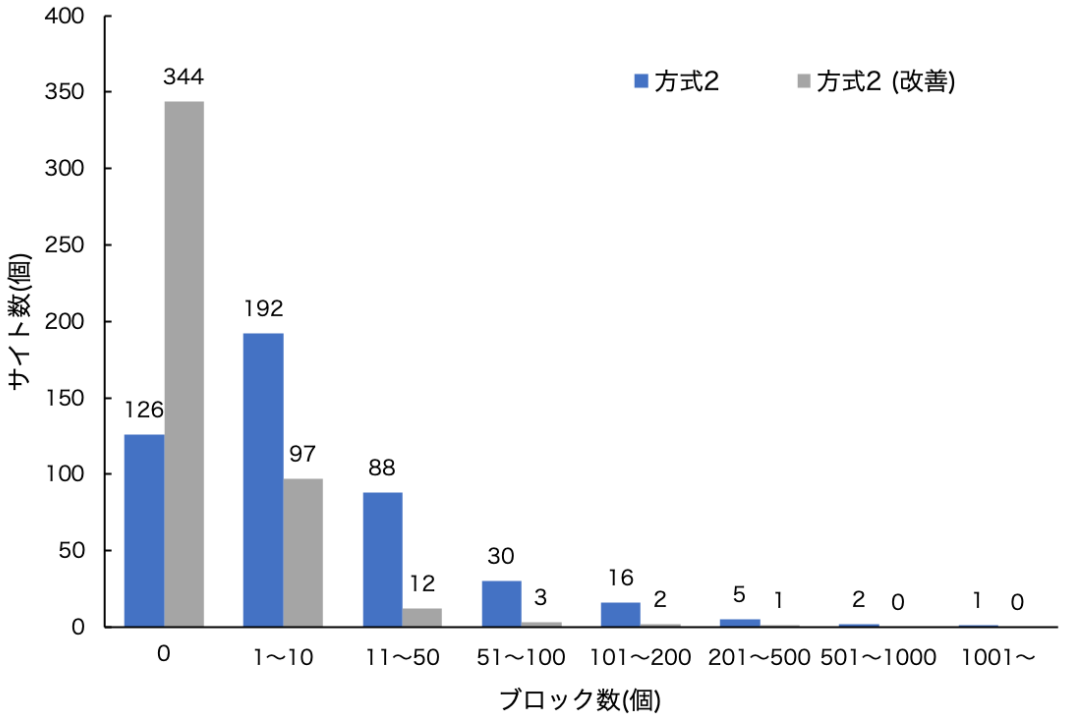

図 29: 改善した提案手法で Alexa Top 500 にアクセスした際のブロック数

図 29 に URL に?か#が含まれていない場合攻撃ではないと判断した上で，Trusted Types を割り当てた場合の Alexa Top 500 にアクセスした際のブロック数を示す．改善した方式 2 の提案手法では，最も多かったブロック数は 0 であり，344 のド

メインが該当し，10 を超えるブロックが 18 のみとなり，大きく既存の Web アプリケーションへの影響を軽減できていることがわかる．

表 9: 改善した提案手法のブロック数と偽陽性率

	ブロック数	偽陽性率
方式 2 (改善)	1269	0.011

改善した提案手法のブロック数と偽陽性率を表 9 に示す．改善した方式 2 の提案手法ではブロック数が 1269 であることから，偽陽性率は約 1.1%となり，方式 2 の偽陽性率が約 11%であるため，偽陽性率を大きく改善できていることが分かった．なお，防御率に関しては方式 2 と等しいことが確認できている．

このように，攻撃がされているかどうかを判断し，Trusted Types を割り当てる条件を追加することで大きく偽陽性率が改善できることが示され，条件を追加することでさらに偽陽性率を小さくすることが可能だと考えられる．

7.4 既存手法との比較

表 10: Client-Side XSS 防御手法の比較, Cost of Implementation (C.I), False Positive Rate (F.P.R)

	Stock ら [21]	Parameshwaran ら [14]	提案手法 方式 1	提案手法 方式 2
C.I	Low	High	Low	Low
Overhead	7–17%	More 5%	1.2%	0.4–1.2%
F.P.R	0.16%	-	46.2%	10.9%

表 10 に Client-Side XSS 防御手法の比較を示す．導入のコストに関しては，Parameshwaran らがプロキシとデータベースが必要であるのに対し，提案手法は Stock らや従来の Trusted Types と同様に実装した Web ブラウザの利用のみなため優位性があるといえる．

パフォーマンスに関しては，Stock らの手法が7〜17%のオーバーヘッド，Parameshwaran らの手法が 5%を超えるオーバーヘッドが生じるとされており，提案手法では 2 つの方式で最大約 1%だったため，パフォーマンスへの影響の大きさでは他の手法に比べて優位性があるといえる．

既存の Web アプリケーションへの影響と偽陽性率に関しては，Stock らの手法が 0.16%の偽陽性率で防御が可能だと示しており，提案手法は 2 つの方式の偽陽性率がそれぞれ約 46%と約 11%であるため，大幅に下回っていることとなった．一方で，偽陽性率に関しては，第 7.3 節で述べたように，攻撃であるかどうかを判断する条件を加えることで，約 1%に改善できることが分かっており，更に攻撃かどうかの判断条件を追加することで，既存手法と同等の偽陽性率に改善出来ると考えられる．

既存の Web アプリケーションへの適用性は，従来の Trusted Types[6] が大幅な変更が必要であるのに対し，提案手法は Stock ら [21] や Parameshwaran ら [14] と同様にアプリケーションの変更を必要としないため優位性があるといえる．

防御率に関しては，既存手法で具体的な値が示されていないものの，Stock らと Parameshwaran らの手法も提案手法の方式 2 と同様に定義した入力ソースからの文字列を許容しないため，防御率に関しては方式 2 と差は生じないと思われる．

7.5 今後の課題

提案手法の評価より，パフォーマンスへの影響を小さくした上で Client-Side XSS の防御が可能であることを示せた．一方で，特に方式 1 の Trusted Types の割り当て方では，防御に伴って既存の Web アプリケーションへの影響が生じ，正常に動かなくなる可能性も大いに考えられる．したがって，今後の課題として，第 7.3 節で示したような方式 2 の割り当て方をベースにした，既存の Web アプリケーションへの影響を考慮し，偽陽性率を下げた Client-Side XSS 防御手法を検討する．

また，Client-Side XSS には，サーバサイドの XSS と同様に，反射型と蓄積（格納）型の Client-Side XSS[20] が存在するとされている．今回行った評価では，Client-Side XSS の脆弱性のデータセットがほぼ全て反射型の Client-Side XSS 脆

弱性だったため，蓄積（格納）の Client-Side XSS に対する評価も今後の課題としてあげられる．それに伴い，方式 2 で定義した表 1 の攻撃者が操作可能な入力ソースも，蓄積（格納）の Client-Side XSS で利用され得る入力ソースの追加が必要になると考えられる．

8. おわりに

近年，Web におけるクライアント側の機能が増加するにつれて，JavaScript の安全でない記述によりクライアント側で発生する Client-Side XSS の脆弱性が問題となっている．しかし従来の Client-Side XSS 防御手法では，Web のパフォーマンスに影響が発生する，または既存の Web アプリケーションに適用が困難など課題が多かった．

そこで，本研究では，パフォーマンスと既存の Web アプリケーションへの適用性を考慮した Client-Side XSS の防御を目的とし，安全な文字列であるかを型で検証する Trusted Types を，プリミティブな型として定義することにより，既存の Web アプリケーションに変更を加えずに Trusted Types を適用し，Client-Side XSS を防御する手法を提案した．提案手法ではプリミティブな Trusted Types の割り当て方式を 2 つ定義し，評価結果としてそのどちらの方式も防御率は 99%を超え，オーバーヘッドを約 1%程度に軽減することが出来た．

今後の課題として，方式 2 の Trusted Types の割り当て方をベースに，既存の Web アプリケーションへの影響を考慮し，偽陽性率を下げた Client-Side XSS 防御手法を検討するほか，蓄積（格納）型の Client-Side XSS に対する検討を行う．

謝辞

主指導教員であり，適切な研究指導をしていただき，様々な側面から研究のサポートをしてくださいました本学情報基盤システム学研究室の藤川和利教授に心から感謝いたします．副指導教員であり，貴重なご意見をいただきました本学情報セキュリティ工学研究室の林優一教授に心から感謝いたします．副指導教員であり，研究方針や論文の執筆について熱心なご助言をいただいた本学情報基盤システム学研究室の新井イスマイル准教授に心から感謝いたします．研究だけでなく，ネットワークやシステムの運用，Web 開発などに多くのことに関してご助言をいただきました，本学情報基盤システム学研究室の垣内正年助教に心から感謝いたします．研究活動や学生生活全般にわたり，多くのご助言をいただきました，本学情報基盤システム学研究室の油谷曉助教に心から感謝いたします．

研究面や生活面において，様々な側面から支援していただいた本学総合情報基盤センターの辻元理恵女史，中野彩子女史に心から感謝いたします．研究室生活において，互いに切磋琢磨しあった本学情報基盤システム学研究室の同期をはじめとした友人の皆様に心から感謝いたします．

最後に，経済面や生活面での多大な援助や励ましをいただいた家族に心から感謝いたします．

参考文献

[1] CERT. Advisory CA-2000-02 malicious HTML tags embedded in client web requests. https://www.rigacci.org/docs/biblio/online/CA-2000-02/CA-2000-02.html. (Accessed on 28/10/2019).

[2] Google. Firig range. https://public-firing-range.appspot.com/. (Accessed on 09/01/2020).

[3] Google. Octane 2.0 javascript benchmark. https://chromium.github.io/octane/. (Accessed on 09/01/2020).

[4] IPA. IPA テクニカルウォッチ「DOM Based XSS」に関するレポート. https://www.ipa.go.jp/files/000024729.pdf. (Accessed on 28/10/2019).

[5] Amit Klein. Dom based cross site scripting or xss of the third kind. 2005.

[6] Krzysztof Kotowicz and Mike West. Trusted types. https://w3c.github.io/webappsec-trusted-types/dist/spec/. (Accessed on 28/10/2019).

[7] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. 2017.

[8] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. pages 1097–1105, 2013.

[9] Ross McIlroy. Background compilation. https://v8.dev/blog/background-compilation. (Accessed on 09/01/2020).

[10] William Melicher, Anupam Das, Mashmood Sharif, Lujo Bauer, and Limin Jia. Riding out domsday: Toward detecting and preventing dom cross-site scripting. 2018.

[11] Mozilla. Kraken JavaScript Benchmark (version 1.1). https://krakenbenchmark.mozilla.org/. (Accessed on 09/01/2020).

[12] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. Scriptprotect: Mitigating unsafe third-party javascript practices. 2019.

[13] OWASP. Cross-site scripting (xss). `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`. (Accessed on 28/10/2019).

[14] Inian Parameshwaran, Enrico Budianto, and Shweta Shinde. Auto-patching dom-based xss at scale. pages 272–283, 2015.

[15] Nishant Das Patnaik and Sarathi Sabyasachi Sahoo. Jsprime. `http://dpnishant.github.io/jsprime/`. (Accessed on 09/01/2020).

[16] The V8 project. V8 javascript engine. `https://v8.dev/`. (Accessed on 09/01/2020).

[17] The WebKit Open Source Project. Jetstream 2 - browserbench.org — browser benchmarks. `https://browserbench.org/JetStream/`. (Accessed on 09/01/2020).

[18] The Chromium Projects. Chromium - the chromium projects. `https://www.chromium.org/Home`. (Accessed on 09/01/2020).

[19] Upasana Sarmaha, D.K. Bhattacharyyaa, and J.K. Kalitab. A survey of detection methods for xss attacks. *Journal of Network and Computer Applications*, pages 113–143, 2018.

[20] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals:investigating the prevalence of persistent client-side cross-site scripting in the wild. 2019.

[21] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against dom-based cross-site scripting. 2014.

[22] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. 2015.

発表リスト

国内会議

山崎 勇二, 垣内 正年, 新井 イスマイル, 藤川 和利, “既存の Web アプリケーションへの適用性を考慮したプリミティブな Trusted Types による Client-Side XSS 防御手法の提案”, 第 87 回コンピュータセキュリティ研究会, 情報処理学会研究報告, Vol.2019-CSEC-87, No.5, pages 1–6, 2019.