情報処理学会論文誌 Vol.59 No.12 2155–2165 (Dec. 2018)

ユーザによるHTMLハイブリッドアプリケーションへのCSP適用が可能なXSS防御基盤Mocha

竹内 俊輝[1,†1,a)] 瀧本 栄二[2] 毛利 公一[2] 齋藤 彰一[1]

受付日 2018年2月26日, 採録日 2018年9月7日

概要：モバイル端末で利用するHTMLハイブリッドアプリケーションが普及しているが，Cross Site Scripting（XSS）による攻撃の危険性が高く，XSSへの対策が重要となっている．XSSへの対策としてContent Security Policy（CSP）が提唱されている．CSPはホワイトリスト形式でコンテンツを制御する機構で，XSSに対する包括的な保護を提供する．しかし，CSPはその利用において開発者の負担が高いことから，利用が進んでいない．また，ユーザが独自にCSPを組み込む研究も行われているが，有効な提案は行われていない．本論文では，ユーザが独自にHTMLハイブリッドアプリケーションにCSPを容易に適用できる機構Mochaを提案する．これにより，ユーザは開発者に依存することなしに，アプリケーションのXSS対策を強化することができる．

キーワード：HTMLハイブリッドアプリケーション，XSS，CSP，Android，Webセキュリティ

Mocha: XSS Prevention System Applying CSP to HTML Hybrid Applications Independent of Developers

Toshiki Takeuchi[1,†1,a)] Eiji Takimoto[2] Koichi Mouri[2] Shoichi Saito[1]

Received: February 26, 2018, Accepted: September 7, 2018

Abstract: HTML hybrid applications used on mobile devices are spreading. The HTML hybrid application has a high risk of attack by Cross Site Scripting (XSS), and countermeasures against XSS are important. Content Security Policy (CSP) has been proposed as a measure against XSS. CSP is a mechanism for controlling content by whitelist and provides comprehensive protection against XSS. However, CSP is not popular because it is burdensome for developers to use. There are also studies that users apply CSP to applications independently, but there is no effective proposal. In this paper, we propose a mechanism Mocha that allows users to easily apply CSP to HTML hybrid applications independently. This allows users to strengthen XSS countermeasures of applications without relying on developers.

Keywords: HTML hybrid application, XSS, CSP, Android, Web security

1. はじめに

モバイル端末上で動くアプリケーションとしてHTMLハイブリッドアプリケーションがあり，HTML5，JavaScript，Cascading Style Sheets（CSS）で構成される．これはモバイル端末の各OSに共通のWebViewを用いて動作するクロスプラットフォームであることから注目を集めている．しかし，JavaScriptを用いるため，HTMLハイブリッドアプリケーションではCross Site Scripting（XSS）脆弱性が懸念される．

XSSとは，攻撃者が任意のJavaScriptをWebアプリケーションで実行できる脆弱性を利用した攻撃である．攻撃者は，HTMLページの改ざんや，ブラウザに保存してあるcookieを盗むこと等が可能となる．HTMLハイブリッ

[1] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi 466–8555, Japan
[2] 立命館大学
Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[†1] 現在，日本電気株式会社
Presently with NEC Corporation
[a)] mocha@ssn.nitech.ac.jp

ドアプリケーションではない Web サイトでは，サーバから取得した JSON 形式のデータや，HTML の GET メソッドと POST メソッドから受け取る文字列で攻撃が行われる．しかし，HTML ハイブリッドアプリケーションではそれらに加え，画像や音声ファイルのメタ情報，SMS や電話帳，Wi-Fi，Bluetooth のアクセスポイント名等の様々な文字列によって攻撃可能であることが指摘されている [1]．このため HTML ハイブリッドアプリケーションでは，XSS への対策がより重要である．

XSS に対する一般的な対策として，HTML ページへのテキストデータの入出力時にテキストデータ内の特殊文字をエスケープする方法や，ブラウザに XSS フィルタ [2], [3] を搭載する方法がある．エスケープする方法では，人為的なミスであるチェックの見逃しやエスケープ自体の失敗が発生しやすく，多くの場合で不完全な対策となる．またフィルタを用いる方法ではすべての XSS を防ぐことができないことやフィルタを通過する攻撃方法を用いられる [4] といった問題がある．

これに対し，XSS への防御策として Content Security Policy（以下，CSP）[5] がある．CSP は HTML のコンテンツをホワイトリスト形式で制限する機構で，XSS に対する包括的な保護を提供する．CSP は現在多くの主要ブラウザで利用でき [6]，Web アプリケーションでの利用も広がっている．CSP をアプリケーションで利用するには，HTML レスポンスヘッダ内に CSP ヘッダを記述し，ポリシを宣言する．このポリシによって許可された HTML コンテンツのみが Web アプリケーションで利用可能となる．

CSP は XSS に対し効果的な防御策であるが，アプリケーション開発者が設定する対策であるため，利用者が CSP により保護されるか否かは開発者に依存する [7]．また CSP によりアプリケーションの動作が制限され，アプリケーションの振舞いや構造に影響が出るため，ユーザが行えるブラウザの拡張機能を用いて CSP を適用する方法では，XSS から保護することは難しい [8]．

本論文では，開発者による CSP 適用が行われていない HTML ハイブリッドアプリケーションに対しても，ユーザが独自に CSP を適用できる機構 Mocha を提案する．Mocha は，Android では HTML ハイブリッドアプリケーションのソースコードが端末内に存在していることに着目し，ユーザが Android 端末内で，開発者に依存することなしにアプリケーションに対し CSP を適用する．Mocha の機能は，Android 端末内で HTML ハイブリッドアプリケーションを静的解析し，適切なポリシの宣言を行う．さらに，ポリシを宣言したことによって生じる HTML と JavaScript の不具合の修正を行い，再度端末にアプリケーションをインストールする．Mocha は，開発者による不十分な設定不備や実装不備に対して，ユーザが独自の対策を行うことを容易にする．また，開発者が Mocha を利用することで，自らの HTML ハイブリッドアプリケーションに CSP を適用するための支援も同時に可能とする．なお Mocha は，Android OS のみを対象とする．

本論文は，2 章で防御方法である CSP について述べる．3 章で提案と設計について述べる．4 章で実装について述べ，続いて 5 章で実装した機構の評価を行う．6 章で関連研究との比較を行い，最後に 7 章でまとめる．

2. CSP

本章では CSP について述べる．はじめに CSP の概要について述べ，CSP の根幹をなすポリシディレクティブについて述べる．そして CSP の現状の調査結果について述べ，調査結果を基とした CSP の問題点について述べる．

2.1 CSP の概要

CSP は XSS を防ぐことが可能な機構であり，アプリケーションに対してホワイトリスト形式の Web コンテンツの読み込み先の制御を行う．CSP は HTTP レスポンスヘッダにヘッダ名として `Content-Security-Policy` を指定し，ヘッダのコンテンツとしてポリシディレクティブを宣言することで利用可能となる．WebView は，アプリの HTML ファイルを読み込む際，head タグ内にあるポリシディレクティブを受け取り，ポリシディレクティブの宣言どおりに動作するよう，HTML の実行を強制する．

2.2 ポリシディレクティブ

Web ページのコンテンツの読み込み先を制限するための宣言である．サーバのホスト名をポリシディレクティブに記述することで，JavaScript，外部プラグイン，CSS 等の読み込み先を指定されたサーバに限定する．ポリシディレクティブは次の形式をとる．

```
directive-name resource-domain
```

`directive-name` は同時に複数宣言できる．利用する `directive-name` の種類だけこの形式を繰り返して宣言することで，細かく制限することが可能である．

meta タグを用いたポリシディレクティブの設定例を**図 1** に示す．図 1 で宣言されたポリシディレクティブは，次の 3 種類である．

- `default-src *`
- `script-src 'self' http://www.test.jp`
- `style-src 'self' 'unsafe-inline'`

図 1 の例における `script-src` とは script タグの読み込み先に制限をかける場合の `directive-name` である．同様に `style-src` は style タグの読み込み先に制限をかけるための `directive-name` である．

`resouce-domain` 部には，任意のホストからの読み出しを許可する “*” か，読み出しを許可するホスト名か，キーワードを記述する．ホスト名指定において，`http://www.`

表 1 resource-domain 部のキーワード
Table 1 Resource-domain keywords.

キーワード	概要
none	すべてのホストからコンテンツを読み出せない
self	自ドメインのみからコンテンツを読み出せる
unsafe-eval	文字列を JavaScript として評価するメソッド（eval 等）を利用できる
unsafe-inline	HTML ファイル内でインラインスクリプトやインラインスタイル等が利用できる

```
<meta http-equiv="Content-Security-Policy" content="default-src *;
script-src 'self' http://www.test.jp; style-src 'self' 'unsafe-inline';">
```

図 1 ポリシディレクティブの例
Fig. 1 Example of policy directive.

```
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
```

図 2 DOM Based XSS の脆弱性の例
Fig. 2 Example of DOM Based XSS.

`test.jp/test/index.js` ファイルを読み込む場合は `resouce-domain` 部に `http://www.test.jp` を記述する．また，CSP レベル 2 では `http://www.test.jp/test/index.js` のようにパスを含めた厳格な指定が可能である（Path matching）．指定できるキーワードを**表 1** に示す．なお CSP レベル 2 では，インラインスクリプトやインラインスタイルの利用を厳格に適用するために，指定されたハッシュ値と一致するインラインスクリプト等が利用できる Source-hash と呼ばれる機能がある．

図 1 で示したポリシディレクティブは，次の意味となる．これを記述した HTML ファイル内において，JavaScript（`script-src`）は自ドメイン（`self`）と `www.test.jp` からの読み込みを許可する．CSS（`style-src`）は自ドメインからの読み込みと，インラインスタイルを許可する．これら以外のコンテンツに関してはどのドメインからの読み込みも許可する（`default-src *`）．

2.3 CSP による XSS 対策

HTML ハイブリッドアプリケーションで多くの場合に問題となりうる XSS は，DOM Based XSS と指摘されている [1]．DOM Based XSS は，Web ブラウザが正規の JavaScript により動的な Web ページを作成する際に，不正なスクリプトを Web ページ内に埋め込むことが可能な脆弱性である．**図 2** に文献 [9] で示されたシンプルな DOM Based XSS の例[*1]を示す．この例では，URL に含まれる “`name=`” に続く文字列を `document.write()` が Web ページに出力している．攻撃は，“`name=`” に続く文字列にスクリプトが指定された場合に，そのスクリプトが Web ページに出力されて実行されることで発生する．DOM Based XSS は，Reflected 型 XSS のように問題となるスクリプトタグが Web ブラウザとサーバの間を往復しないため，Web ブラウザが備える XSS フィルタでは防ぐことができない．

CSP は，ホワイトリスト形式の保護手法であるため，意図しないリソースを拒否することができる．XSS により攻撃者が入力する JavaScript コードからアプリケーションを CSP を用いて保護するためには，JavaScript コードを意図しないリソースとして識別して拒否することが必要である．そこで，CSP による XSS の防止は，JavaScript コード等のインライン機能を HTML ファイルから除外したうえで，ポリシディレクティブの `directive-name` の `script-src` に `unsafe-inline` と `unsafe-eval` を含めないことでインライン機能を意図しないリソースとして抑制することで実現する．これにより，攻撃者が注入したスクリプトタグは CSP ポリシによって実行されない．当該機能が必要な場合は，インライン形式ではなく外部ファイルとして独立させることで対応可能である．

2.4 CSP の現状調査

CSP の現状を知るために，実際の Web サイトでどの程度 CSP が利用されているかを調査した．Alexa のトップサイト [10] からヘッダを取得できたホストとその関連ページを含む，101,863 ページの HTTP レスポンスヘッダの調査を行った．本調査の対象は，実際に CSP を運用する際に使用する `Content-Security-Policy` をヘッダに記載しているホストとした．

2.4.1 CSP の使用状況

調査の結果，2,492 のドメイン，3,753 ページで CSP が利用されていた．2014 年 5 月に行われた調査では Alexa の Top 1M のサイトのうち，850 のドメインでのみ CSP が使用されいた [11]．また 2016 年の 5 月の調査では Alexa の Top 1M のサイトのうち 3,220 ドメインが `Content-Security-Policy` ヘッダを利用していた [12]．これらより，CSP の利用率は年を経て増加していることが分かる．

2.4.2 ポリシディレクティブの調査

CSP を利用している 3,753 ページのポリシディレクティブの調査をした結果，XSS 対策が完全なページと，CSP を利用しているが `unsafe-inline` や `unsafe-eval` を使用

[*1] 本脆弱性は，初期の脆弱性であるため，現在の URL エンコードを搭載しているブラウザでは，入力されたスクリプトがタグとして扱われず動作しない．

しているため完全な XSS 攻撃の対策を行えていないページがあることが分かった．

まず CSP を用いて XSS 対策が完全なページは 41 ページであった．XSS 対策が完全なポリシディレクティブとは，2.3 節で述べたように `default-src 'self';` であり，ファイルの読み込みを厳格に設定し，`unsafe-inline` と `unsafe-eval` を含めていないものである．

次に CSP を利用していたページのうち `unsafe-inline` を使用しているページは 1,793 ページ，`unsafe-eval` を使用しているページは 1,615 ページであった．またこの 2 つを同時に使用しているページは 1,612 ページであった．このことから CSP を用いて XSS を防ぐことができているサイトは少数であるといえる．

2.5 CSP の問題点

調査結果より，CSP を用いているサイトは増加傾向であるが少数であることが分かる．その中でも XSS を防御可能な設定を行っているサイトはごく少数である．この理由として CSP を適用するための手順に原因があると考える．アプリケーションに CSP を適用するには 2 つの手順が必要となる．まず，すべての HTML ファイルに対してポリシディレクティブの設定と宣言を行う．次に，ポリシディレクティブの宣言に沿うように HTML と JavaScript を修正する．

これらの手順のために，アプリケーション開発者と利用者の双方にとって，CSP をアプリケーションに適用させるには問題点がある．まずアプリケーション開発者が CSP を適用するには，ポリシの設定やソースコードの修正のための時間が必要になる．また，修正にともなうエラーも生じうる．このため開発者の負担を増加させるといえる．また利用者がブラウザの拡張機能等 [2], [3] を用いて CSP を利用する場合は，ソースコードの修正が行えないため，アプリケーションの動作に影響が懸念される．現状多くのアプリケーションでインラインスクリプトが使用されている [8] ため，ポリシの設定のみで XSS を防ぐのは難しいといえる．

3. 提案と設計

本章では，CSP の問題点を緩和する手法の提案を 3.1 節において行う．3.2 節で，Android 端末上で HTML ハイブリッドアプリケーションに CSP を自動適用する機構の動作概要について述べ，3.3 節以降で主要な実装事項について述べる．

3.1 提案

Android 端末上で既存の HTML ハイブリッドアプリケーションに CSP を自動適用する機構 Mocha を提案する．Mocha は，利用者に，CSP が適用されていない HTML ハイブリッドアプリケーションに CSP を適用する機構を提供する．またこの機構は，アプリケーション開発者にとっては，自らのアプリケーションに CSP を適用する際の支援となると考える．

HTML ハイブリッドアプリケーションは，サーバ側の言語が用いられていないため動的に決定する要素が少ないことから，Android 端末上での静的解析のみで CSP 適用が可能である．また動的解析のための事前の動作を必要としないため適用にかかる時間を短縮可能である．なお，動的要素に依存するアプリケーションへの対応は，Android 端末のみでの解析が難しいことから今後の課題とする．

3.2 設計

Android 端末上で HTML ハイブリッドアプリケーションに CSP を適用するために必要な処理は，ソースコードの抽出と解析，ポリシディレクティブの決定と適用，ポリシディレクティブに合わせたソースコードの修正と，修正後コードの再インストールである．これらの処理の流れを**図 3** に示し，処理概要を以下に示す．

(1) 設定画面において Android 端末上にインストールされたアプリケーションの中からユーザがインストールしたアプリケーションの一覧を利用者に提示する．利用者がアプリケーションを選択し，CSP 適用のボタンを押すことによって，Mocha によるアプリケーションへの CSP 適用を開始する．
(2) 端末上の apk をデコンパイルし HTML，JS，CSS ファイルを抽出する．
(3) ポリシディレクティブを，HTML ファイルの静的解析結果に基づいて設定する．
(4) アプリケーションの HTML ファイルを，ポリシディレクティブに一致するように自動で修正する．
(5) JavaScript についても同様に修正する．
(6) 変更したソースコードを再度 apk にまとめ，Android 端末へのインストールを行う．

また Mocha は (3)，(4)，(5) の機能を抽出した Java アプリケーションとして開発者が利用することも可能であ

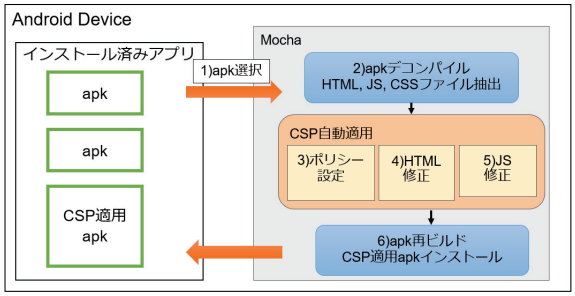

図 3 Mocha の全体図
Fig. 3 Structure of Mocha.

る．開発者は CSP を適用するアプリケーションを含んだディレクトリと CSP のレベルを入力として指定することで利用可能となる．

3.3 ポリシディレクティブの設定

ポリシディレクティブを，XSS 脆弱性が発生しないように設定する．そのため 2.3 節で述べたように，`directive-name` の `script-src` にはキーワードの `unsafe-inline` と `unsafe-eval` を含めない．アプリケーションに必要であるコンテンツが外部にある場合，ホスト名を HTML ファイルと JavaScript ファイルの解析によって取得する．なお，CSP のレベルは Android OS のバージョンに依存するため，Android の OS バージョンが 5 以上の場合には CSP レベル 2 を使用するように設定を行う．

3.4 HTML ファイルと JS ファイルの修正

3.3 節で述べたポリシディレクティブの設定を行うことで，HTML ファイルではインラインスクリプト，インラインスタイル，イベントハンドラ，javascript:URI 形式の動作ができなくなる．また，JavaScript ファイルでは動的に script タグを生成するメソッドと文字列を JavaScript として評価するメソッドが動作しなくなる．これらを修正する実装を行う．

4. 実装

本章では，Mocha の実装について述べる．Mocha は 3.2 節で述べた各処理により，HTML ハイブリッドアプリケーションに対して CSP を適用する．処理順に各機能の詳細を述べる．

4.1 対象アプリケーションの指定

設定画面を**図 4** に示す．この画面でユーザは適用対象となるアプリケーションを指定する．ユーザは，プルダウンメニューでアプリケーションを指定できる．アプリケーションは Android 端末にデフォルトでインストールされているシステムアプリケーション以外を表示する．署名ファイルはユーザが自作した署名ファイルを利用したい場合は Select File ボタンから選択可能である．

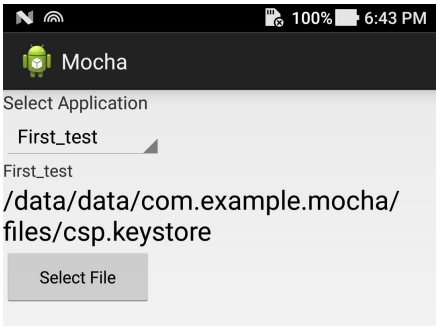

図 4 対象アプリケーションの指定
Fig. 4 User interface of selecting application.

4.2 デコンパイル

アプリケーション（apk）の実体は zip ファイルである．インストールされたアプリケーションの apk は/data/app/ ディレクトリ以下に保存されている．ユーザが指定したアプリケーションの apk を Mocha の作業ディレクトリ上にコピーおよび展開し，HTML ファイル，JS ファイル，CSS ファイルを得る．以降これらのファイル群に対して各処理を行う．展開には Java で標準に用意されている `java.util.zip` ライブラリを用いた．

4.3 ポリシディレクティブの設定

ポリシディレクティブは，基本となるポリシディレクティブを用意し，それをアプリケーションに合わせて拡張する．**図 5** はこの基本のポリシディレクティブを HTML 文書に記載するときの書式である．この基本のポリシディレクティブは，JavaScript と CSS の読み込みをアプリケーションのパッケージ内に限定する制限である．

拡張は，`resource-domain` 部に新たなホストとハッシュ値を記述することによって行う．ホストは HTML ファイルおよび JavaScript ファイルを解析した結果に基づいて記述する．以下に処理順に従って処理概要を示す．

(1) HTML 解析
  (a) HTML が参照している外部ホストの取得
    - script タグの src 属性の外部ホスト
    - link タグの href 属性の外部ホスト
  (b) ポリシディレクティブに取得した外部ホストを追加
(2) JavaScript 解析
  (a) jQuery 利用による外部ホストの取得（**図 6** 参照）
    - url 部（図 6 の 4 行目）の外部ホスト

```
<meta http-equiv="Content-Security-Policy"
content="default-src *; script-src 'self'; style-src 'self';">
```

図 5 基本となるポリシディレクティブ
Fig. 5 Base policy directive.

```
1: $.ajax({
2: type:'POST',
3: dataType:'json',
4: url: 'http://www.test.jp/api.php,
5: …});
```

図 6 jQuery を用いた外部ホストとの通信例
Fig. 6 Example code of communication with external-host using by jQuery.

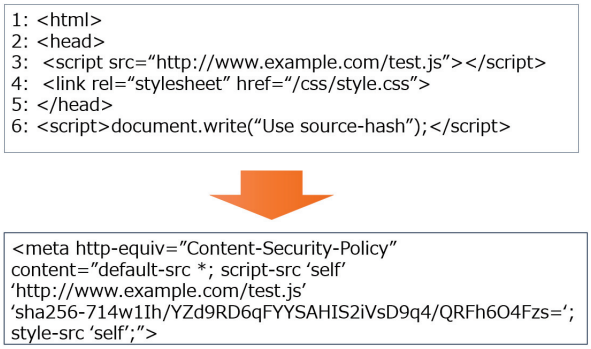

図 7 ポリシディレクティブの拡張例（レベル 2）
Fig. 7 Example of policy directive extension (level 2).

(b) ポリシディレクティブに取得した外部ホストを追加

(3) Source-hash 設定（レベル 2 の場合）

(a) インラインスクリプトのハッシュ値を取得

(b) ポリシディレクティブに取得したハッシュ値を追加

ポリシディレクティブの拡張例（レベル 2）を**図 7** に示す．図 7 の上部の HTML を解析した結果のポリシディレクティブが下部となる．図 7 の HTML の 3 行目で外部ホストである `http://www.example.com` の JavaScript ファイル `test.js` を読み込んでいるため，`script-src` に `http://www.example.com/test.js` を記載する．また 4 行目にある CSS ファイルの読み込みは自ホストであるため，`style-src` に追記する必要はない．最後に 6 行目のインラインスクリプトはハッシュ値を計算し，その結果を `script-src` に記載する．

4.4 HTML の修正

HTML ファイルをポリシディレクティブに沿う形にするには，1) インラインスクリプト，2) インラインスタイル，3) イベントハンドラ，4) javascript:URI の修正が必要である．基本的な修正方法は，HTML ファイル内に記載された当該コードを，外部ファイルに移す方法である (2.3 節参照)．本実装では，HTML ファイルのパーサとして JSOUP [13] を用いて修正を行う．JSOUP は Java で作成されたオープンソースの HTML パーサであり，DOM を利用し HTML ファイルの解析と修正に利用できる API を提供する．

4.4.1 インラインスクリプトの修正方法

インラインスクリプトの修正方法について述べる．インラインスクリプトとは，HTML ファイルに script タグを用いて JavaScript を記述した部分のことである．インラインスクリプトの修正例（レベル 1）を**図 8** に示す．CSP のレベルが 1 の場合，Mocha では script タグ内の JavaScript を正規表現で抽出し，外部ファイルへ書き出す．さらにこの外部ファイルを呼び出すように変更する．CSP のレベルが 2 の場合，4.3 節で述べた処理により，インラインスクリプトのハッシュ値をポリシディレクティブに記載することで対応する．

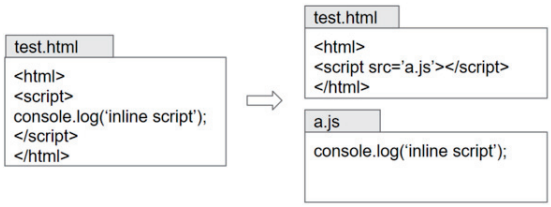

図 8 インラインスクリプトの修正例（レベル 1）
Fig. 8 Example of inline script modification (level 1).

4.4.2 インラインスタイルの修正方法

Mocha は WebView で動く HTML ハイブリッドアプリケーションを対象としているため，インラインスタイルによる XSS は発生しない．しかし，インラインスタイルによって XSS が Internet Explorer ブラウザで発生する[*2] [14]．Mocha の Web アプリケーションへの転用も考慮して，インラインスタイルの修正を行う．

インラインスタイルは，style タグもしくは style 属性を用いて HTML ファイルに CSS を記述するものである．インラインスタイルの修正方法はインラインスクリプトと同様に，該当の部分を正規表現を用いて外部ファイルに記述して参照するように修正する．

4.4.3 イベントハンドラの修正方法

イベントハンドラとは，ユーザの動作や操作に対して特定の処理を与えるための命令である．Mocha では DOM を利用した addEventListener 関数を利用し，イベントを登録し直す修正を行う．addEventListener で利用できるイベント名はイベントハンドラで使用する名称と差異があるが，どのイベント名を用いればよいかは W3C で定義されている [15]．addEventListener 関数に変更するには，実行する関数名とイベント名，そして実行する DOM のタグを指定する id が必要となる．イベントハンドラの修正例を**図 9** に示す．次の手順により登録する．1) 各イベントハンドラのイベント名を HTML ファイルより取得，2) 取得したイベントに登録されている JavaScript とイベントハンドラに対応するイベント名を取得，3) イベントハンドラ用の雛形ファイルに，2) で取得した JavaScript とイベント名を書き込む．4) HTML ファイルを雛形ファイルを読み込むように設定する．5) 連携のための id を決め，イベントハンドラが登録された DOM のタグに id を追記する．

4.4.4 javascript:URI の修正方法

javascript:URI とは，html のタグに付加可能である href 属性に “javascript:...” と記述することにより JavaScript が

[*2] `<div style="color:expression(alert('XSS'));">a</div>`

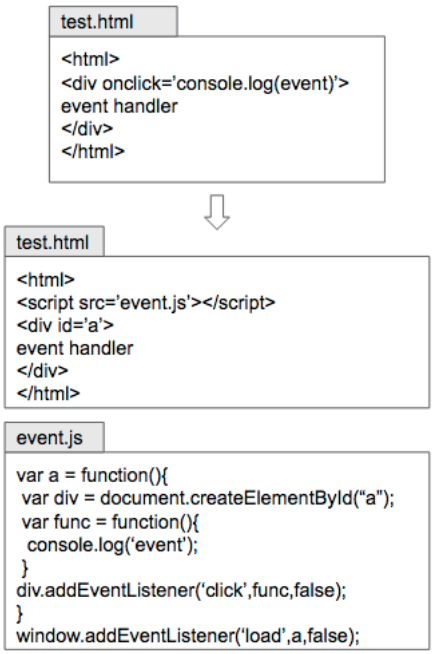

図 9 イベントハンドラの修正方法
Fig. 9 Modification of event handler.

実行可能となるものである．この形式もイベントハンドラのときと同様の手法で修正することが可能である．ただし，対応するイベントハンドラは onclick，addEventListener 関数で利用するイベント名は click となる．なお，javascript:URI 形式とイベントハンドラが同一タグに設定されているとき，javascript:URI の実行が優先的に処理されるため，矛盾が発生しないように修正する．

4.5 JavaScript の修正

JavaScript ファイルの修正について述べる．ポリシディレクティブに従う JavaScript に修正するには，動的 HTML タグ作成を利用した script タグの作成の修正と，文字列を JavaScript として評価する関数の修正が必要である．基本的な修正方法は，HTML ファイルの修正と同様に，関連コードを外部ファイルに移す方法である（2.3 節参照）．

Mocha では，独自の JavaScript パーサを Java を用いて開発し，JavaScript の修正を行う．本パーサは，修正対象メソッドを含むクラスと，JavaScript ファイル内の変数を取得するクラスを提供する．

4.5.1 動的 HTML タグ作成に関する修正方法

動的に HTML を作成する方法として，document.write，document.writeln，innerHTML，outerHTML の 4 種類が DOMAPI に存在する [1]．JavaScript ファイルがこれらの関数とメソッドを利用し script タグを作成して HTML ファイルに挿入する場合は，基本修正方法どおりに，作成した script タグが外部ホストに存在する JavaScript ファイルを参照する形式に修正する．

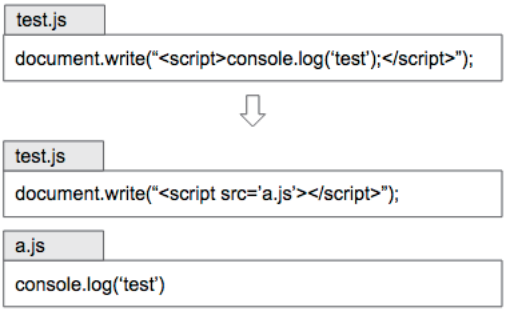

図 10 document.write 形式の修正方法
Fig. 10 Modification of document.write.

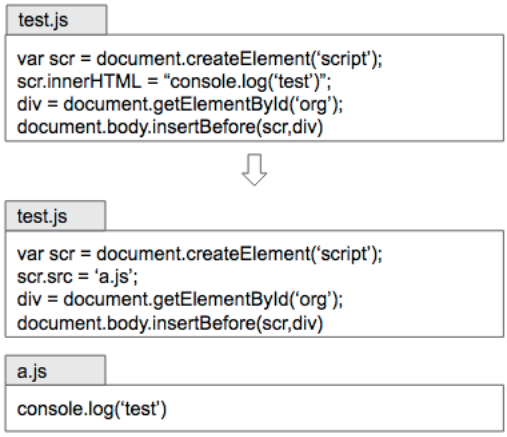

図 11 innerHTML 形式の修正方法
Fig. 11 Modification of innerHTML.

4.5.1.1 document.write と document.writeln の修正方法

document.write と document.writeln は同じ修正方法で修正を行う．document.write 形式の修正例を**図 10** に示す．修正は，ファイル内にあるすべての document.write と document.writeln について，引数を正規表現で取得し，それぞれに script タグが存在するかを調べる．存在した場合は，その内容を外部ファイルに書き出す．さらに，script タグの内容を，書き出した外部ファイルを読み込むように変更する．

4.5.1.2 innerHTML と outerHTML の修正方法

innerHTMLとouterHTMLは，document.createElement で作成したタグ内の文書を出力するメソッドであり，document.createElement は HTML のタグを作成する JavaScript のメソッドである．修正は，document.createElement で script タグを作成し，かつ innerHTML と outerHTML を使用していた場合に必要となる．innerHTML 形式の修正例を**図 11** に示す．修正方法は，innerHTML と outerHTML の内容を外部ファイルに書き出し，書き出した外部ファイルを読み込むように修正する．

4.5.2 文字列を JavaScript として評価するメソッドの修正方法

文字列を JavaScript として評価するメソッドには set-

Interval，setTimeout，eval，Funtion() の 4 種類が存在する．setInterval と setTimeout は指定した時間に登録した JavaScript を実行するメソッドである．これらのメソッドは引数として文字列と関数をとることができる．文字列を引数としてとる場合がポリシディレクティブに違反するため，無名関数を引数としてとるように修正する．

eval と Function() は使用方法が複数あり，使用方法に合わせた修正が必要である．使用方法は Richard らによると 10 種存在する [16]．そのうち 6 種類は修正方法が同文献で述べられているため，その方法を用いて修正する．そのほかの 4 種類については対処方法が述べられていない．しかし，それらは eval の引数をユニコードに変換する JavaScript エンコードを行うことによって，攻撃者の入力が実行されない形式に変更されるため XSS の発生を防ぐことができる．しかし，この方法で eval の対処を行う際は，ポリシディレクティブの `script-src` の `resource-domain` 部に `unsafe-eval` を用いなければならず XSS の危険性を排除できない．よって，完全な対応は今後の課題となる．

4.6 apk のリコンパイル

CSP を適用したアプリケーションを Android にインストールするための apk のリコンパイルについて述べる．デコンパイル時に展開したファイルのうち META-INF ディレクトリには開発者の署名情報が保存されている．これらのファイルは apk 内のファイルを展開すると利用不可能になり，かつ Mocha が署名を行う際に競合が発生する．そのため，META-INF ディレクトリ以外を zip で圧縮する．圧縮は Java で標準に用意されている `java.util.zip` を用いて行う．次に，作成した zip ファイルに対して jarsigner を用いて署名をつける．署名ファイルは Mocha に付属させたものを使用するが，ユーザが作成した署名を用いることも可能である．設定画面に署名を選択するフォームがあるため，そこでユーザが作成した署名を選択することでユーザが作成した署名を用いることが可能となる．またユーザ自身が署名を作成し，作成した署名で zip ファイルに対して署名をつけることも可能である．

5. 評価

本章では Mocha の評価について述べる．Android 端末上で Mocha を用い，CSP 適用に必要な時間の計測，CSP 適用後のアプリケーションの動作，またソースコードの変更個数と CSP ポリシの設定についてまとめと考察を行う．

5.1 評価方法

Mocha を用い，HTML ハイブリッドアプリケーションに対して評価を行う．評価には，XSS 脆弱性を含む評価用に作成したアプリケーションと，Fossdroid [17] で公開されている OSS の Android アプリケーション 27 個を用いた．

表 2 評価環境
Table 2 Evaluation environment.

Android 端末	Zenfone 3
Android 端末の OS	AndroidOS 7.0
AndroidEmulator の OS	AndroidOS 4.4.2

まず，XSS 脆弱性の存在する評価用アプリケーションに対し Mocha を適用し，適用後のアプリケーションにおいて XSS が不可能となっていることを確認する．次に，CSP を適用した HTML ハイブリッドアプリケーションが動作するか否かを確認する．なお，OS のバージョンによって CSP のレベルが変わるため，CSP レベル 1 に対応できることを OS のバージョンが 5 未満の Android Emulator を用いて確認し，CSP レベル 2 に対応できることを OS のバージョンが 5 以上の実端末により確認する．次に Java の標準関数を用い CSP 適用に要する時間を計測した．最後に CSP 適用に必要なソースコードの改変個数，および CSP ポリシの設定に関してまとめる．CSP 適用によるソースコードの改変数はポリシディレクティブの違反数と同値である．そのため HTML ファイルと JavaScript ファイルの違反数，HTML ファイルの違反数，そして JavaScript ファイルの違反数の 3 項目を違反個数ごとに分類する．本評価で使用した環境を**表 2** に示す．

5.2 XSS の防御に関する評価

Mocha を用いることで XSS を防ぐことが可能かどうかについての評価を行った．評価手順は，まず XSS 脆弱性を含む評価用アプリケーションを開発する．その評価用アプリケーションに Mocha を用いて CSP を適用し，CSP 適用後の評価用アプリケーションで XSS が発生しないことを確認する．なお，評価用アプリケーションには 2 つの致命的な DOM Based XSS 脆弱性を含むように作成した．1 つ目は innerHtml を用いた脆弱性で，2 つ目は document.write を用いた脆弱性である．

評価を行った結果，2 つの XSS を防いだことを確認した．評価用アプリケーションの実行に際して，あらかじめ含まれていた正規の JavaScript は動作し，XSS となりうる外部からの入力による JavaScript のみが遮断されていた．これにより Mocha を用いて XSS を防ぐことが可能であるといえる．

5.3 Mocha の CSP 適用に関して

すべてのアプリケーションについて CSP 適用後に起動することを確認した．これにより HTML ハイブリッドアプリケーションでは Android 端末上で改変することが可能であるといえる．しかし，アプリケーション開発者が改変を望まなく，証明書の比較を行っている場合，起動しない可能性がある．今回評価したアプリケーションではそのよ

うなアプリケーションは存在しなかった．次に CSP のレベルごとの確認を行った．まずインラインスクリプトを使用しているアプリケーションに関して，CSP レベル 2 では source-hash が利用されており，CSP レベル 1 では外部ファイルに分割されていることを確認した．また外部ドメインにあるファイルを読み込んでいるアプリケーションに関して，CSP レベル 2 では Path matching が適用されていることを確認し，CSP レベル 1 ではドメインのみ記述されていることを確認した．

5.4 CSP 適用に要する時間

ユーザが CSP 適用のボタンを押してからアプリケーションの CSP 適用が完了するまでの時間を計測した．計測した結果，平均で 14.389 秒，最長で 119.795 秒であった．今回評価に利用したアプリの CSP 適用にかかる計測時間と APK ファイルサイズとの分布を**図 12** に，計測時間と HTML および JS ファイル数との分布を**図 13** に示す．図 12 より CSP 適用にかかる時間は APK のファイルサイズに比例して長くなる．これは静的解析のためすべての HTML ファイルと JS ファイルの修正が必要なためである．また図 13 より HTML と JS ファイルの数に比例して計測時間が長くなることが分かる．APK のファイルサイズに比例し計測時間が延びるのは，APK のデコンパイルとリコンパイル，署名に時間がかかるためである．また図 12 と図 13 で計測時間が 110 秒を超えている点は同一のアプリケーションである．このアプリに要した時間は，APK ファイルサイズ別では最長ではなくファイル数別では最長である．このことから CSP 適用にかかる時間に対する影響は，HTML と JavaScript ファイルの数よりもファイルサイズの影響が大きいといえる．

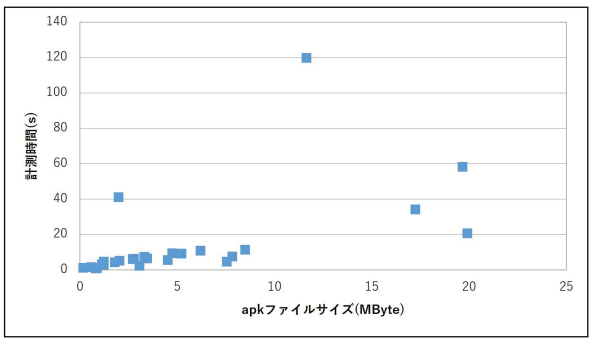

図 12 計測時間と APK ファイルサイズの分布図
Fig. 12 Relation between execution time and APK file size.

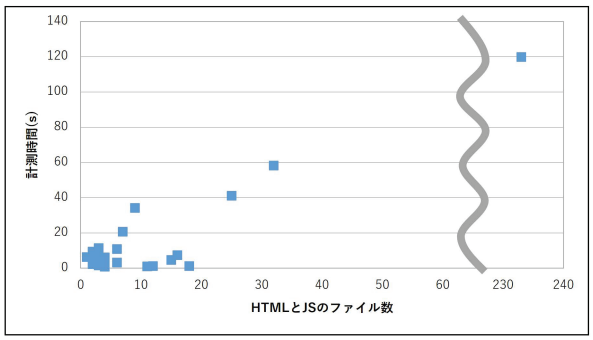

図 13 計測時間と HTML，JS ファイル数の分布図
Fig. 13 Relation between execution time and number of files (HTML, JS).

5.5 ソースコード改変結果

ソースコードの改変数について述べる．評価アプリケーションの HTML と JavaScript ファイルの CSP 違反数を**表 3** に示す．27 個のアプリケーションで評価した結果，HTML ファイルと JavaScript ファイルの両方が設定したポリシディレクティブで違反が発生せず，修正の必要のなかったアプリケーション（表中の “HTML and/or JS” で改変数が 0）は 27 個中の 9 個であり，残りの 18 個のアプリケーションは修正が必要であった．このように，多くのアプリケーションにおいて CSP ポリシの適用のみでは動作に影響が発生するといえる．また，これら 18 個のアプリケーションについて，HTML ファイルのみ改変が必要だったアプリケーションの数を “HTML のみ” により，JavaScript のみ改変が必要だったアプリケーションの数を “JavaScript のみ” で示す．それぞれ一方のみ改変が必要だったアプリケーション数は各 6 個で，残り 12 個は HTML と JavaScript の両方の改変が必要であった．

ポリシディレクティブの設定に関して，外部ホストにあるファイルを読み込んでいるアプリケーションにおいて CSP ポリシによって制限されることがなかったため，ポリシの設定は正しく行われているといえる．しかし，外部ホストの安全性が保証されていない [18] ことがあるため，今後は外部ホストにあるファイルはダウンロードし，APK ファイル内に含めることが必要である．

次に，HTML ファイルの修正に関して，HTML ハイブリッドアプリではサーバサイドの言語がないため HTML の要素が動的に決まることはないことから，修正後のアプリケーションの動作に異常が発生することはない．

最後に，JavaScript ファイルの修正に関して，静的要素の修正は正しく行われたことを確認した．しかし，JavaScript の内容は動的に決定することがあるため，場合によっては修正後のアプリケーションの動作に異常が発生することが

表 3 HTML と JS ファイル改変数
Table 3 Number of modifying HTML and JS files.

改変数	改変アプリケーション数		
	HTML and/or JS	HTML のみ	JavaScript のみ
0	9	6	6
1–10	14	8	9
11–40	3	3	2
41–	1	1	1

ある．そのため，動作時の動的要素の内容によって動作を変更できるような実装を行う必要がある．また Mocha は JavaScript の圧縮と難読化に対応できていない．ゆえにこれらの点が今後の課題である．

6. 関連研究

本章では関連研究について述べる．アプリケーションに対する CSP のアプローチに関する手法を述べ，本提案との比較を行う．

AutoCSP [19] は，PHP を利用した Web アプリケーションに対して自動で CSP を適用する手法である．テイント解析を行うことで，信頼できる JavaScript と信頼できない JavaScript に分割し処理を行うことで，Stored 型 XSS に対し高度な防御を実現している．Mocha と比較した利点として，PHP 等のサーバ側スクリプトを利用して HTML 構文を生成する Web アプリケーションにも対応している点がある．一方，Mocha は JavaScript ファイルの修正について実装していることと，ユーザが独自に利用できるという利点がある．

CSPAutoGen [20] は，ユーザが CSP を利用する手法である．サーバと Web ブラウザとの通信の間に CSPAutoGen が介入する．CSP によって必要とされるソースコードの修正を CSPAutoGen が行うことで，開発者によるポリシの設定とソースコードの修正が必要なくなる．これによりユーザは安全なアプリケーションを利用できるようになる．Mocha と比較した利点として，CSP 適用の対象を HTML ハイブリッドアプリケーションに限定しておらず一般の Web ページに適用できる点がある．一方，データベースを必要とすることから Mocha の対象である Android 端末で利用することは難しいと考える．

CSP AiDer [21] は Web アプリケーションに対するポリシディレクティブを提示する機構である．Web ページを解析することにより適切なポリシディレクティブを設定する．しかし，CSP AiDer は CSP の仕様策定段階で提唱されたものであるため，ポリシディレクティブの形式が現行利用されているものと異なる．そのため現在利用されている主なブラウザでは利用できず，Mocha に対する優位性はない．またこの機構はポリシディレクティブの提唱のみであり，HTML と JavaScript ファイルの修正機能がない点も Mocha と異なる．

UserCSP [8] は FireFox の拡張機能であり，Web ブラウザが Web ページをロードする際に動的解析を行い，CSP を適用する．開発者に対しては CSP を適用させる際の手助けとして，利用者に対しては CSP を利用し XSS からの保護の手助けを行う機構として提案されている．FireFox の拡張機能であることから，HTML ハイブリッドアプリケーション以外にも適用可能という利点がある．しかし，UserCSP ではソースコードの改変は行えないため，インラインスクリプトや eval を利用している Web ページに対しての XSS からの保護が十分であるとはいえない．Mocha では，利用デバイスが Android 端末に限定されるが，ソースコードの改変が可能であるため，XSS からの保護が十分であるといえる．

7. おわりに

本論文では，Android 端末上で既存の HTML ハイブリッドアプリケーションに対し CSP を自動適用する機構 Mocha を提案した．Mocha は，HTML ハイブリッドアプリケーションに対して Android 端末内で CSP を適用することを可能にし，利用者がアプリケーション開発者に依存することなしに CSP を用いて XSS に防ぐことを可能とした．利用者は CSP の適用を望むアプリケーションを選択するのみで CSP が自動適用されるため，利用者の負担は少ない．また Mocha の機構の一部を利用することで開発者も Web アプリケーションへ CSP を適用することが可能となる．

既存の HTML ハイブリッドアプリケーションに対し，本提案機構を用いて評価した結果，多くのアプリケーションが XSS を防ぐためのポリシディレクティブの設定では違反するコードを利用していたため，ソースコードの修正が必要であることが分かった．またポリシディレクティブの設定，HTML ファイルの修正は静的解析のみで確実かつ正確に修正できることを確認した．JavaScript ファイルの修正に関してはサーバからデータを受け取る処理や関数の受け取る引数が動的に決まる処理，そして JavaScript ファイルの圧縮と難読化以外の修正が可能であることを確認した．今後は動的に決まる引数を関数の実行直前で調査し対応する方法を検討する．

参考文献

[1] Jin, X., Hu, X., Ying, K., Du, W., Yin, H. and Peri, G.N.: Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation, *Proc. 2014 ACM SIGSAC Conference on Computer and Communications Security* (*CCS*), pp.66–77 (2014).

[2] Bates, D., Barth, A. and Jackson, C.: Regular expressions considered harmful in client-side XSS filters, *Proc. 19th International Conference on World Wide Web* (*WWW*), pp.91–100 (2010).

[3] Saxena, P., Molnar, D. and Livshits, B.: SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications, *Proc. 18th ACM SIGSAC Conference on Computer and Communications Security* (*CCS*), pp.601–614 (2011).

[4] Heyes, G.: Bypassing XSS Auditor, available from ⟨http://www.thespanner.co.uk/2013/02/19/bypassing-xss-auditor/⟩.

[5] W3C: Content Security Policy 2.0, available from ⟨http://www.w3.org/TR/CSP⟩.

[6] Deveria, A.: Can I use Content Security Policy 1.0?, available from ⟨http://caniuse.com/contentsecuritypolicy⟩.

[7] Kour, H. and Sharma, L.S.: Browser compatibility issues in implementing Content Security Policy to prevent Cross Site Scripting attacks, *International Journal of Modern Computer Science*, Vol.4, No.3, pp.108–112 (2016).
[8] Patil, K. and Frederik, B.: A Measurement Study of the Content Security Policy on Real-World Applications, *International Journal of Network Security*, Vol.18, No.2, pp.383–392 (2016).
[9] Klein, A.: DOM Based Cross Site Scripting or XSS of the Third Kind (2005), available from ⟨http://www.webappsec.org/projects/articles/071105.shtml⟩.
[10] Alexa: The Top 500 sites on the web, available from ⟨https://www.alexa.com/topsites⟩.
[11] Weissbacher, M., Lauinger, T. and Robertson, W.: Why Is CSP Failing? Trends and Challenges in CSP Adoption, *International Workshop on Recent Advances in Intrusion Detection* (*RAID*), pp.212–233 (2014).
[12] Calzavara, S., Rabitti, A. and Bugliesi, M.: Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild, *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security* (*CCS*), pp.1365–1375 (2016).
[13] jsoup: jsoup: JAVA HTML Parser, available from ⟨http://jsoup.org⟩.
[14] Ruby: Ruby on Rails Security Guide, available from ⟨http://guides.rubyonrails.org/security.html#css-injection⟩.
[15] W3C: UI Events Specification, available from ⟨https://www.w3.org/TR/DOM-Level-3-Events/⟩.
[16] Richards, G., Hammer, C., Burg, B. and Vitek, J.: The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications, *Proc. 25th European Conference on Object-oriented Programming* (*ECOOP 2011*), pp.52–78, Springer (2011).
[17] Simonin, D.: Fossdroid, available from ⟨https://fossdroid.com/⟩.
[18] Google Security Blog: CSP Evaluator, available from ⟨https://security.googleblog.com/2016/09/reshaping-web-defenses-with-strict.html⟩.
[19] Fazzini, M., Saxena, P. and Orso, A.: AutoCSP: Automatically Retrofitting CSP to Web Applications, *Proc. 37th International Conference on Software Engineering* (*ICSE*) (2015).
[20] Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y. and Zhou, T.: CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-world Websites, *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security* (*CCS*), pp.653–665 (2016).
[21] Javed, A.: CSP AiDer: An Automated Recommendation of Content Security Policy for Web Applications, *IEEE Symposium on Security and Privacy* (2011).

竹内 俊輝

2016 年名古屋工業大学工学部情報工学科卒業，2018 年同大学大学院工学研究科博士前期課程修了，同年日本電気株式会社入社，現在に至る．

瀧本 英二 （正会員）

1999 年立命館大学理工学部情報学科卒業，2001 年同大学大学院理工学研究科博士前期課程修了，2005 年同研究科博士後期課程単位取得退学，同年（株）ATR 適応コミュニケーション研究所専任研究員，2010 年立命館大学情報理工学部情報システム学科助手，2017 年立命館大学情報理工学部情報理工学科助教，現在に至る．主にシステムソフトウェア，無線通信に関する研究に従事．博士（工学）．電子情報通信学会会員．

毛利 公一 （正会員）

1994 年立命館大学理工学部情報工学科卒業，1996 年同大学大学院理工学研究科修士課程情報システム学専攻修了，1999 年同研究科博士課程後期課程総合理工学専攻修了．同年東京農工大学工学部情報コミュニケーション工学科助手，2002 年立命館大学理工学部情報学科講師，2004 年同大学情報理工学部情報システム学科講師，2008 年同准教授，2014 年同教授となり，現在に至る．博士（工学）．オペレーティングシステム，仮想化技術，コンピュータセキュリティ等の研究に従事．電子情報通信学会，ACM，IEEE-CS，USENIX 各会員．

齋藤 彰一 （正会員）

1993 年立命館大学理工学部情報工学科卒業，1995 年同大学大学院博士前期課程修了，1998 年同博士後期課程中退．同年和歌山大学システム工学部助手，2003 年同講師，2005 年同助教授，2006 年名古屋工業大学大学院助教授（准教授），2016 年同教授，現在に至る．博士（工学）．オペレーティングシステム，インターネット，セキュリティ等の研究に従事．ACM，IEEE-CS 各会員．