情報処理学会論文誌　プログラミング　Vol. 3　No. 4　16–26 (Sep. 2010)

ブラウザで動作するウェブアプリケーションのソースコード隠蔽機構

折戸隆洋[†1]　岩崎英哉[†2]

近年，Ajax を用いて提供されるウェブアプリケーションが注目を集めている．Ajax を用いたサービスで利用されている JavaScript は，ソースコードを誰でも読むことができるという特徴を持つ．このことは，他のアプリケーションと連動させるマッシュアップに適しており，利用者や他の開発者による修正および発展を期待できるといった利点をもたらす反面，プログラムの盗用が容易であること，プログラムに脆弱性が残されていた場合に悪意ある者にこれを突かれる危険性があるという問題点がある．そこで本論文では，ブラウザで動作するウェブアプリケーションの JavaScript コードを隠蔽する機構 SCvanisher を提案する．SCvanisher は，従来クライアント上のブラウザで実行されていた JavaScript コードをサーバ上の JavaScript 実行部で処理し，結果のみをクライアントに送信する．クライアント上のブラウザでの操作を JavaScript 実行部に伝達し，両者が協調することで，ウェブアプリケーションの本来の機能はそのままにソースコードの隠蔽を実現する．SCvanisher を利用することで，提供するウェブアプリケーションの JavaScript コードは完全に隠蔽されるため，アプリケーション開発者はソースコード隠蔽を意識することなく JavaScript による開発を行うことができる．

Hiding Source Code of Web Application on Client Browser

Takahiro Orito[†1] and Hideya Iwasaki[†2]

Recently web applications that use JavaScript have become very popular. Developers of such applications cannot avoid publishing JavaScript source code, because the code has to be sent from the web server to the client to be executed on the client's browser. This causes two problems. First, the source code could be stolen by another developer. Second, if the application has a security hole, attackers could easily find out its vulnerability. In this paper, we propose SCvanisher, a mechanism that hides the source code of a web application from the clients. SCvanisher executes the original JavaScript code of the application on the web server, and sends the resultant web page that do not include the original code to the client. It achieves interactive behavior of a web application such as the text input by making both server and client sides cooperate. By using SCvanisher, the developer can easily describe JavaScript code without being annoyed with hiding its source code.

1. はじめに

近年，ウェブアプリケーションが広く認知されはじめている．ウェブアプリケーションとは，一般的なアプリケーションと同等の動作を行うウェブページの総称であり，ウェブブラウザ上で動作することが特徴である．ウェブアプリケーションにおいては，JavaScript プログラムがクライアントのブラウザ内で動作し，ブラウザの画面を書き換える．このようなウェブアプリケーションが幅広く認知された要因として，Ajax（Asynchronous JavaScript and XML）と呼ばれる技術が普及したことがあげられる．Ajax とは，動的にブラウザの画面を書き換えることができる JavaScript の特徴を活かし，画面遷移をともなわないプログラムの実行を実現する技術である．Ajax が普及したことに加え，ウェブブラウザにおける処理速度の向上が目覚ましいことにより，JavaScript が注目されている．

JavaScript は，ブラウザ内でインタプリタ方式で動作するスクリプト言語である．インタプリタ方式であるため，クライアントで動作するウェブアプリケーションは，必然的に JavaScript のソースコードをクライアントにダウンロードする．そのため，ソースコードを誰でも読むことができる，すなわちオープンソースであるという特性がある．この特性から JavaScript には，他のアプリケーションと連動させるマッシュアップに適している，利用者や他の開発者による修正および発展を期待できるといった利点がある．この利点を活かした代表例として，Google Maps があげられる．Google Maps は，単独では非同期通信を活かした地図閲覧アプリケーションであるが，マッシュアップにより，たとえば不動産屋のサイトと連動して選択した物件の周辺を表示するような実用的なアプリケーションから，小型飛行機を操作して世界中を飛び回ることができるような，趣味の範疇のアプリケーションに至るまで，幅広く利用されている．

しかし，JavaScript のオープンソースであるという特徴には問題点もある．1 つは，簡単

†1 電気通信大学大学院電気通信学研究科
Graduate School of Electro-Communications, The University of Electro-Communications
†2 電気通信大学大学院情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications

にプログラムの盗用が可能であること，もう 1 つは，プログラムに脆弱性が残されていた場合に悪意ある者にこれを突かれる危険性が非常に高いことである．これらの問題点により，ウェブアプリケーションの開発において，JavaScript が最適であるとは一概にはいえない状況にある．

これらの問題点を解決するため，本論文は，クライアントのブラウザで動作するウェブアプリケーション利用者から JavaScript コードを隠蔽する機構である SCvanisher を提案し，実装することを目的とする．SCvanisher を利用することにより，JavaScript を用いたウェブアプリケーションにおいても，クローズドソースのアプリケーションと同様の知的財産保護，および，ソースコード盗用に対する堅牢性の確保を期待できる．

SCvanisher は，従来どおりの記法の JavaScript コードそのままに，従来どおりの動作を保証しつつ，アプリケーション利用者へのソースコード流出を防ぐことを目標としている．SCvanisher を利用することで，JavaScript に明るくない人でも，混乱することなくソースコードを隠蔽でき，JavaScript に精通している人は，従来隠蔽が不可能であった部分のソースコード隠蔽の実現や，ソースコード隠蔽のためにクライアント側とサーバ側とで分けて行っていた実装を，クライアント側にまとめて実装しても問題なくなるという利点が得られる．これにより，ウェブアプリケーション開発の負担が低減されることが期待できる．

SCvanisher の基本的なアイディアは，従来はクライアントのブラウザ内で実行していた JavaScript コードを，クライアントではなくサーバにおいて実行し，実行結果の HTML をクライアントに転送する，というものである．SCvanisher の現在の実装では，サーバにおける JavaScript コードの実行を，サーバ上にもブラウザを稼働させそのブラウザ内で行わせる．すなわち，サーバ上のブラウザは JavaScript エンジンとして利用する．したがって，このブラウザによる（サーバにおける）画面表示には，クライアントを利用している利用者にとっては（そもそも利用者はサーバの画面を見ることができないので）特別な意味はない[⋆1]．

本論文の構成は次のとおりである．まず，2 章で従来のソースコード保護手法について，問題点を検証し，3 章では，SCvanisher の設計，実装を行う．4 章では実装した機構について評価を行う．続く 5 章では，SCvanisher の限界について議論し，最後に，6 章で本論文をまとめる．

⋆1 SCvanisher の開発者にとっては，サーバのブラウザにおける画面表示と，クライアントのブラウザにおける画面表示を見比べれば，SCvanisher の正常動作を確認できるという意味はあった．

2. 従来のソースコード保護手法

ソースコードの保護手法には大別して，難読化と隠蔽の 2 種類が存在する．双方について，既存のソースコード保護機構をふまえて概説し，問題点を述べる．

2.1 難　読　化

難読化とは，プログラムの意味や動作を変更することなくソースコードに編集を加えることで，可読性を下げる手法である．難読化の最大の目的は，ソースコード解読を難しくすることであり，解読を不可能にするものではない．難読化の一般的な手法としては，以下のようなものがあげられる．

- 空白文字や改行の除去
- 変数名および関数名の分かりにくいものへの変更
- ダミーコードの挿入

難読化を行う際には，一般的に専用のアプリケーションを用いる．代表的なものとして，以下のようなツールがある．

Dotfuscator[1] は，Microsoft 社の .NET Framework を利用して作成されたプログラムの中間言語に難読化処理を行うものであり，先述の一般的な手法に加え，文字列の暗号化や制御フローの難読化といった独自の処理を行うことが可能である．これらにより，中間言語をバイナリエディタで開いて行うような人力での解読はもちろんのこと，逆コンパイラに対しても，制御フローの難解さから解析を困難にすることを実現している．

SHTML[2] は，HTML および JavaScript のソースコードを対象としたものであり，先述の一般的な手法に加え，ソースコードそのものを暗号化する．

また，難読化の手法に関する研究として，以下のものがある．村山ら[3] によるバイナリコードの難読化では，複雑な命令を必要最低限の単純な命令のみで書き換えることで，各命令の出現分布を一様にすることで難読化を行っている．門田ら[4] によるソースコードの難読化では，ループを含むプログラムについて，繰返し文や制御文を複雑に変換することで難読化を行っている．

しかし，先に述べたようにこれらの手法はあくまでも解読を困難にするにすぎない．難読化の手法が日々進化するのと同様に，逆コンパイラや可読化アプリケーションの開発もまた日々進化しているであろう．これらのツールの動作を妨げるためのアンチクラックツールも普及しているが，ほとんどのアンチクラック技術は破られており，万能のアンチクラックツールは存在しないといわれている[5]．また，SHTML で用いられているような暗号化手

法は，直接ソースコードを読むことは困難だが，JavaScript に関する多少の知識を持っている人ならば，ウェブブラウザに内蔵されているデバッグツールやプラグインを用いることや，ブックマークレットと呼ばれる JavaScript の実行手法を利用することで，復号化したコードを取得できてしまう．

難読化によりプログラムをブラックボックス化することは不可能であることがすでに証明されている[6] ことからも，難読化はソースコード保護の手法としては完全とはいえない．

2.2 隠　　蔽

隠蔽とは，ソースコードそのものを利用者が閲覧できない状態にする手法である．

隠蔽の一般的な手法としては，以下のようなものがあげられる．

- ソースコードを別の形式に変換する．
- 処理はサーバで行い結果のみクライアントに送信する．

これらを実現する機構として，以下のようなものがある．

bRuby[7] および Exerb[8] は，JavaScript 同様のインタプリタ言語である Ruby コードの隠蔽を行う．Ruby のソースコードを，bRuby により独自のバイナリコードに変換し，Exerb がこれを専用のインタプリタとともにパックすることで，Windows 上で動作する単一の実行ファイルに変換するものである．しかしながら，この手法においても，強力なリバースエンジニアリングには耐えられないことが指摘されている[9]．

Jaxer[10] は，サーバで行う処理およびクライアントで行う処理の双方を 1 つの JavaScript ソースコードに記述することができる機構である．これは，本来はソースコードの隠蔽を目的とした機構ではないが，サーバ側で実行するように指定した部分は結果的に隠蔽されることになるので，隠蔽の目的で利用することができる．しかし Jaxer では，関数ごとにサーバとクライアントのどちらで実行するかの指定および，サーバで実行された関数の結果をどこで用いるかの指定を行う必要がある．つまり，Jaxer を用いたウェブアプリケーション開発を行うためには，JavaScript の記法だけでなく，Jaxer 独自の記法を習得する必要が生じる．また，サーバとの通信を行う部分や，画面の書き換えを行う部分など，この手法を用いてもすべてのソースコードを隠蔽することはできない．

3. ソースコード保護機構の設計と実装

3.1 ソースコード隠蔽手法の検討

従来の難読化および隠蔽手法の問題点をまとめると，次のようになる．

- 難読化
  - 可読化を行うツールが存在する場合がある．
  - 難読化処理を行う部分に脆弱性が見つかると無力化されたも同然である．
  - 厳重に処理を行っても完全な保護は不可能である．
- 隠蔽
  - バイナリコードへの変換では難読化同様に完全な保護は不可能である．
  - サーバ上での実行ではクライアントとの連携の考慮などにより敷居が高い．
  - サーバクライアント間の通信部分および画面書き換え部分は隠蔽できない．

ここから読み取れることは，プログラムが利用者のコンピュータ上で何らかの形で実行される限り，ソースコードの完全な保護は不可能である，という事実である．

また，本研究が対象とする言語が JavaScript であることを考えると，クライアント側での対策によるソースコード保護はきわめて難しいことが予想される．本研究の目標として，従来どおりの JavaScript の記法のままにソースコードを保護することを掲げている．クライアント上で JavaScript を実行しつつこれを実現するためには，ソースコードが解読できない状態に変換してからクライアントが読み込むか，クライアントが読み込んだ情報を完全に表示できなくする必要性がある．前者については現状において実現不可能であることを先述した．後者については，既存のウェブブラウザでは JavaScript コードをキャッシュとして保持してしまうため，新規にブラウザの開発を行う必要がある．しかし，仮に JavaScript コードをキャッシュせず，内部での処理も閲覧することができないブラウザを設計し実装したとしても，わざわざ機能の削られたブラウザを利用者が選択することは期待できない．

3.2 設　　計

前節における考察から，本研究における JavaScript コードの隠蔽機構 SCvanisher の設計は，1 章で述べたように，従来はクライアントのブラウザ内で実行していた JavaScript コードをサーバ上で代理実行し，実行結果だけをクライアントに転送する，という方針をとる．クライアントと，サーバ上でのコード代理実行部は 1 対 1 に対応するので，代理実行部を同時に複数用意することができれば，設計上は，サーバの処理能力に応じた数のクライアントに対して同時にサービスを提供することができる．

クライアントのブラウザで動作する JavaScript コードは，DOM 操作を通じて HTML を操作し，画面を書き換える．したがって，サーバ上でクライアントの代わりに JavaScript コードを実行するためには，同様の DOM 操作が可能な JavaScript エンジンが必要である．そこで SCvanisher では，既存のブラウザである Firefox をサーバ上で稼働させ，JavaScript エンジンとして利用することとした．以後混乱を避けるため，このサーバ上で稼働する JavaScript

エンジンとしてのブラウザを “サーバブラウザ” と，クライアントにおいて利用者が閲覧するブラウザを “クライアントブラウザ” あるいは単に “ブラウザ” と呼び，両者を区別する．

サーバブラウザは，DOM 操作を行うのに加え，クライアントブラウザに対して，JavaScript コードの実行結果を送信しなければならない．そこで，SCvanisher においては，これらを Telnet によって行うこととした．サーバブラウザに対する Telnet 接続は，Firefox に MozRepl というプラグインを導入することで可能になる．サーバブラウザにおける DOM 操作や実行結果の取得は，サーバとクライアントの間に介在し両者間の通信を中継する役割を持つ “中継部” が，サーバブラウザに対して Telnet 接続し，MozRepl が用意している専用のコマンドを送信して行う．これらの詳細は 3.4 節で述べる．

Firefox と MozRepl を用いることの欠点は，MozRepl の仕様上，Telnet を介して接続できるサーバブラウザが 1 つに限られてしまうということである．その結果，現状の SCvanisher では，同時利用クライアント数は 1 に限定されてしまうが，実装コストを考慮し，Firefox と MozRepl を利用することとした．SCvanisher の実装からこの制約を外し，サーバの処理能力に応じたスケーラビリティを確保することは，今後の課題である．

サーバブラウザにおける実行結果の HTML には，`script` タグの中に隠蔽すべき JavaScript のコードが残っている可能性がある．したがって，実行結果の HTML をのままクライアントブラウザに送信するのではなく，JavaScript コードを削除し，さらに削除された JavaScript コードを呼び出す（マウスボタン入力などに対する）イベントハンドラをサーバへの通信処理に置換するなどの加工を施した HTML を，クライアントブラウザに送信する．こうすることにより，クライアントブラウザでエラーなどの不都合を生じさせることなく，サーバと協調してウェブアプリケーションを動作させることができる．

なお，サーバブラウザを遠隔のサーバ上で稼働させるという SCvanisher の設計は，VNC（Virtual Network Computing）によるリモートデスクトップと類似する点があるため，VNC を利用するという方法はないのかという疑問を感じる人もいるかもしれない．しかし，ソースコード隠蔽という趣旨からすると，ユーザが閲覧するブラウザと JavaScript コードを実行する部分が分離している必要があるため，VNC によるリモートデスクトップでは，ユーザは遠隔にあるブラウザを操作することになり，隠蔽という目的を達成することができない．

3.3　動作の概要

SCvanisher の動作の概要を**図 1** に示す．SCvanisher は，JavaScript コードをクライアントブラウザではなくサーバ上で実行し，その結果をクライアントブラウザに送信する．クライアントブラウザ上で行われるのは，実行結果の表示および，利用者が行ったマウス操作

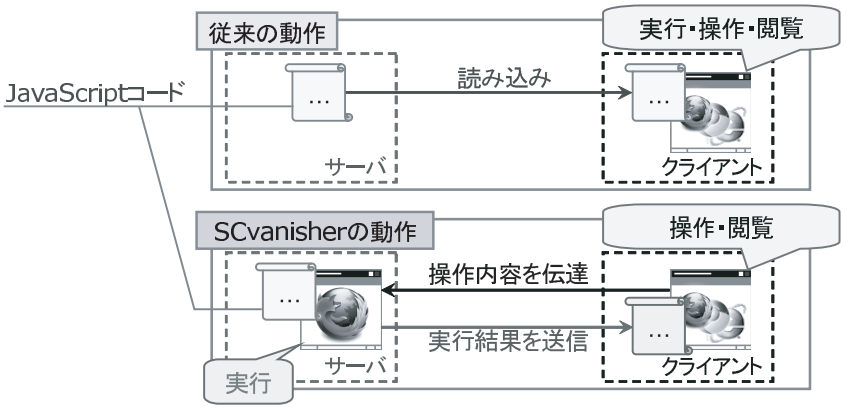


図 1　SCvanisher の動作の流れ
Fig. 1　Outline of operations of SCvanisher.

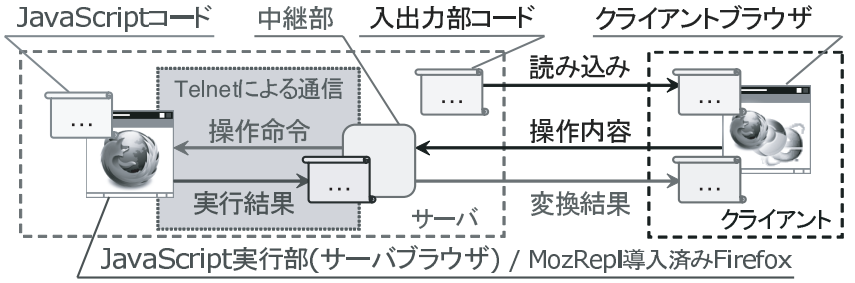


図 2　SCvanisher の構成
Fig. 2　Overall structure of SCvanisher.

などのサーバへの伝達のみである．

SCvanisher の構成要素は，**図 2** のように，クライアントブラウザ上で動作する入出力部コード（JavaScript コード），サーバ上で動作をする中継部，中継部と同じくサーバ上で動作するサーバブラウザによる JavaScript 実行部の 3 つに分けられる．利用者がこのウェブアプリケーションを利用するためにサーバにアクセスすると，サーバとの通信やサーバから送信されたデータを表示を行う入出力部コードが送信され，クライアントブラウザで開かれる．クライアントブラウザで入出力部が開かれその中のコードが実行されると，最初

に JavaScript 実行部を立ち上げるための命令がサーバ上の中継部に送信される．これによりサーバ上で JavaScript 実行部となるサーバブラウザが立ち上がり，サーバに設置されたウェブアプリケーションを開く．ウェブアプリケーションを JavaScript 実行部で実行した結果得られる HTML コードを中継部が取得し，入出力部で正しく処理できる形に変換した後クライアントブラウザ内の入出力部に送信することで，利用者はウェブアプリケーションの実行結果を得ることができる．中継部により加えられた変換により，クライアント上の入出力部でテキスト入力やボタン操作が行われると，中継部を介して JavaScript 実行部にこの操作が反映される．この結果もまた中継部により適切に処理され，クライアントブラウザに反映される．このようにして，サーバ上の JavaScript 実行部とクライアントブラウザは同期する．

ウェブアプリケーションの動作はサーバ上で完結しているため，ソースコードがサーバ外に晒されることはなく，隠蔽が実現する．

3.4 実　　装

SCvanisher の動作の概要から，各構成要素に求められる内容を整理し，実装する．

3.4.1 JavaScript 実行部

サーバ上で動作する JavaScript 実行部には，次の 4 つの内容が求められる．

- 任意のウェブページを開くことができる．
- ウェブページにテキスト入力などの変更を加えられる．
- ウェブページに実装されている JavaScript コードを任意に実行できる．
- JavaScript の実行結果が反映された HTML コードを外部から任意に取得できる．

これらを実現するために，3.2 節で述べたように，既存のウェブブラウザである Firefox にプラグイン MozRepl を導入して JavaScript 実行部（サーバブラウザ）を構成し，外部からの操作を Telnet で行えるようにした．MozRepl は Telnet 接続に用いるポートや，他の IP アドレスからの接続の可否を自由に設定できるため，万一 Telnet 接続に用いているポートが外部に漏れてしまっても，外部からの接続を許可しないよう設定しておけば，サーバブラウザである Firefox を不正に操作される恐れはない．また，JavaScript 実行部として Firefox を用いることで，ウェブアプリケーション開発者は，SCvanisher を利用した設置を考えた場合でも，Firefox で正しく動作することのみを確認すればよく，開発における負担が増加することはなくなる．

3.4.2 入 出 力 部

クライアントブラウザ内で動作する入出力部の動作の実現には，Ajax を利用する．このことで，一般的なウェブブラウザにおいて動作可能にし，SCvanisher を介したウェブアプリケーションの利用に手間がかからないようにする．

入出力部において利用者が行う操作は，次の 2 つに大別される．

- イベントハンドラを持たない要素への操作
- イベントハンドラを持つ要素への操作

前者は，操作を加えても他の要素に影響を与えないものを指し，認証用の ID 入力フォームのように，後からボタン操作などを行うことでその情報が用いられる部分が該当する．このような部分は，入出力部で行われた操作を中継部に伝達し，中継部が JavaScript 実行部に反映させるという片道通信だけで，入出力部の表示内容と JavaScript 実行部の状態を一致させることができる．

これに対し後者は，操作を加えることで，他の要素にも何らかの影響が生じる部分を指し，選択項目を変えることで，別の場所で表示する情報が変わるプルダウンメニューのようなものが該当する．こちらは，まず前者と同様に利用者により入出力部で行われた操作を，中継部を介し JavaScript 実行部へ伝達する．これにより，後者においては JavaScript 実行部では対応した関数が実行される．この時点で，入出力部での表示と JavaScript 実行部の状態には差異が生じるため，入出力部は続いて，JavaScript 実行部の状態を取得する命令を中継部へ送信する．後述する中継部の動作により，JavaScript 実行部の状態が入出力部に伝達され，これを反映することで JavaScript 実行部の状態と入出力部の表示内容は一致する．

中継部から送信された内容の反映は，入出力部の HTML 内に用意された，専用の `div` タグの内部を動的に書き換えることで実現する．書き換え前の内容と送信された内容の差分をとり，書き換え量を減らすという方法[11] も存在するが，差分をとるためにかかるオーバヘッドが書き換えにかかる時間とほぼ変わらず，全体を書き換えることでも実際の動作に支障はないと判断した．

3.4.3 中　継　部

サーバ上で動作する中継部が行う動作は，次の 2 つに大別される．

- 入出力部で行われた操作を JavaScript 実行部に反映させる．
- JavaScript 実行部の状態を入出力部に伝達する．

中継部のこれらの機能は，Perl で記述した CGI プログラムとして実現した．

前者については，Telnet を介した DOM 操作により，利用者が入出力部で行った操作を擬似的に JavaScript 実行部である Firefox 上で再現する．入出力部からは，操作対象であるタグの id および操作内容が伝達されるので，これを，Firefox が解釈可能な形に整形し，

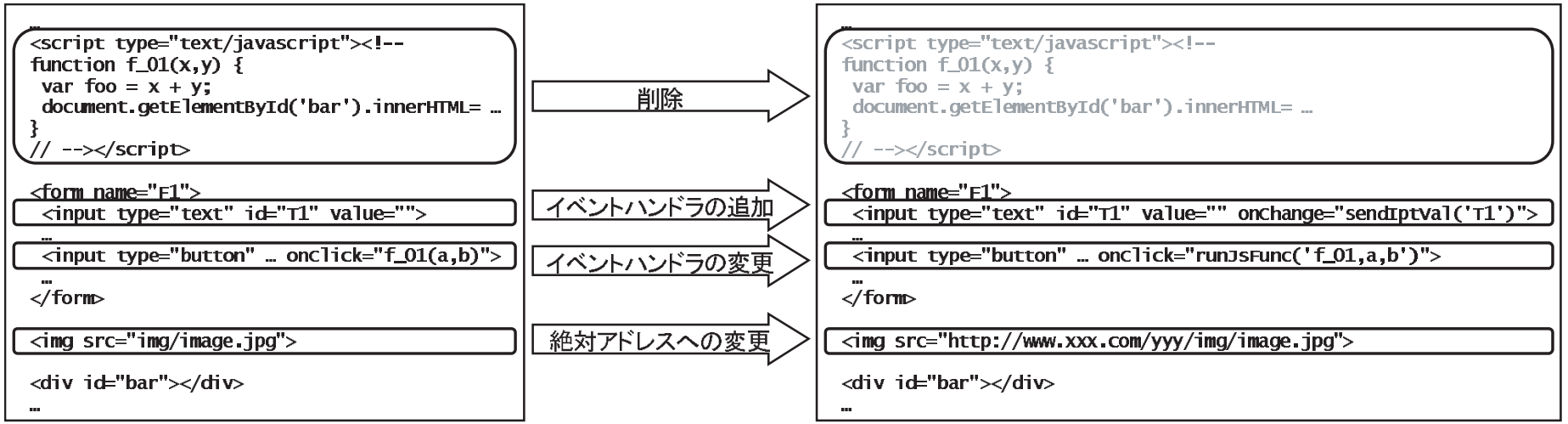


図 3　中継部によるコード変換例
Fig. 3　Conversion example by relay part.

Telnet を介して JavaScript 実行部へ伝達することで実現する．

後者についてはまず，JavaScript 実行部である Firefox のその時点での HTML コードを，やはり Telnet を介して取得する．この HTML コードは，**図 3** の左側の状態にあたり，このままでは，以下の 3 つの問題が生じる．

- ウェブアプリケーション固有の隠蔽したい JavaScript コードが残留している．
- 入出力部で行われた動作を中継部へ伝達できない．
- SCvanisher を設置したフォルダ階層次第で相対アドレスに問題が生じる．

これらを解決するため，中継部は取得した HTML コードに対し，**表 1** の規則に沿った変更を加える．まず，残留したウェブアプリケーション固有の JavaScript コードは削除する．この JavaScript コードは，JavaScript 実行部にて実行済みであるため，これを消すことによるウェブアプリケーションの動作に影響はない．また，テキストボックスやプルダウンメニューのようなフォームについては，イベントハンドラを持たないものについては，新たにイベントハンドラの付与を行い，イベントハンドラを持つものについてはイベントハンドラの内容を書き換える．これらのイベントハンドラは，行われた操作を中継部に伝達するための関数を実行するようになっており，これにより入出力部と JavaScript 実行部の内容が一致するようになる．相対アドレスで書かれた部分は，絶対アドレスに記述を変更することで問題を解決する．

表 1　中継部における変換ルール
Table 1　Conversion rule in relay part.

対象		処理内容
script タグ noscript タグ		タグ自身を含めて内包する情報をすべて削除
input タグ	イベントハンドラあり	入力内容伝達と関数実行を行うように変更
select タグ etc…	イベントハンドラなし	入力内容伝達用イベントハンドラを追加
a タグ	href オプション	（ウェブアプリケーションを別サーバに設置時）
img タグ etc…	src オプション	相対アドレスで書かれている部分を 絶対アドレスに書き換え

図 3 左側にこの変更を加えたものを，図 3 右側に示す．付与されたイベントハンドラは `sendIptVal` という関数へ id の文字列を与えた呼び出しに，変更されたイベントハンドラは `runJsFunc` という関数に本来呼び出すべき関数名と実引数を与えたものになっている．ここで新たに与えられた `sendIptVal` と `runJsFunc` は，3.3 節で説明した入出力部コード

の中に含まれる SCvanisher 用のライブラリ関数であり，隠蔽対象ではない．これらの変換を施した HTML コードを入出力部へ送信することで，ウェブアプリケーション本来の使用感を残したまま，ソースコードの隠蔽を実現する．

3.4.4　ウェブアプリケーションの設置

利用者に提供するウェブアプリケーションを構成するファイルは，原則として SCvanisher と同一のサーバに設置する．構成ファイルのうち，JavaScript コードなどの隠蔽すべきコードを含むファイルは，サーバ外からのいっさいの操作を禁止するようなパーミッション設定にする．逆に，画像，音声などのマルチメディアファイルは外部からの読み込みを許可しておかなければならない．マルチメディアファイルは，HTML コードと異なり，SCvanisher を通じての転送ができないためである．

4. 評　　価

4.1　定性的評価

SCvanisher を用いることで，実際にウェブアプリケーションのソースコードを隠蔽できているか，また，SCvanisher を介した場合でもウェブアプリケーションが問題なく動作しているかを評価するため，複数人のゲームのスコアを登録し，比較できるようなウェブアプリケーションを実装した．このウェブアプリケーションでは，最初に利用者は ID を入力して，専用の管理ページを開く．また，画面表示の切替えにはページ遷移をともなわない動的書き換えを用いた．

記述には，一般的な Ajax の実装方法として，クライアント上で動作する部分に JavaScript を，サーバ上で動作する部分に Perl を用いた．

4.1.1　ソースコードの隠蔽

用意したウェブアプリケーションを，そのままサーバに設置した場合と，SCvanisher を介して提供されるように設置した場合について，ウェブアプリケーション利用中の通信内容を調査した．調査には，Google Chrome の JavaScript コンソール機能を用いた．

従来どおりの設置を行った場合の通信内容には，JavaScript のソースコードをはじめ，各利用者の認証に用いる ID を格納したファイルまでクライアントに送信されていた．これに対し，SCvanisher を介して提供されるように設置を行った場合は，SCvanisher のソースコードを受信していることは確認できたが，ウェブアプリケーションのソースコードや，ID を格納したファイルの存在は確認できず，隠蔽に成功していることが分かった．

4.1.2　動 作 確 認

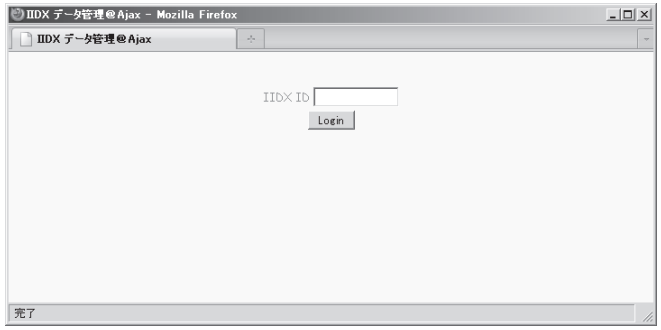
評価のために用意したウェブアプリケーションは，入力フォームの表示，非表示の切替えなどのすべての機能の切替えに JavaScript による動的書き換えを利用しているため，入出力部と JavaScript 実行部が正しく協調して動作しなければ，スコアの編集を行うことはできない．

SCvanisher の動作を確認するため，Microsoft Internet Explorer 7.0.5730.13，Mozilla Firefox 3.5.7，Google Chrome 3.0.195.38 の 3 つのウェブブラウザにおいて，以下の操作を行った．

(1)　URL を指定し，ログインページを開く．
(2)　ID を入力し，ログインボタンを押す．
(3)　管理ページでメニューが表示されるので，ゲームスコア表示ボタンを押す．
(4)　スコアが棒グラフで表示されるので，あるプレイヤの編集ボタンを押す．
(5)　スコア表示画面で編集が可能な状態になるので，スコアを編集し確定ボタンを押す．
(6)　送信結果がスコア表示画面に反映されるので，保存ボタンを押す．
(7)　保存が完了したので，別ページに移動するボタンを押す．
(8)　再びスコア表示画面に移動するボタンを押す．
(9)　ログアウトボタンを押す．

その結果，いずれのウェブブラウザにおいても，表示の切替えは正常に行われ，すべての操作が正しく行われることを確認した．

図 4 と**図 5** に，Firefox をクライアントブラウザとして用いたときの画面の表示例を，(1) の直後と (4) で棒グラフが表示された直後について示す．これらの図に示したブラウザ画面は，SCvanisher 導入時と非導入時の両者で変わりはないが，表示している HTML コードは両者では異なっている．そこで図 4 と図 5 では，両者の HTML コードの主要部を比較して掲げた．このスコア管理ウェブアプリケーションは動的書き換えを行っているため，ここでの HTML コードは，JavaScirpt の `document.body.innerHTML` を出力させて得た．双方の HTML コードを見比べると，SCvanisher 導入時には `onclick` などで指定されているイベントハンドラが，ハンドラ関数の直接呼び出しではなく，`sendIptVal` あるいは `runJSFunc` など，中継部への情報伝達に変更されていることが分かる．このようにして，クライアントブラウザにおけるイベントは，適切にサーバ側に伝えられ処理が行われる．



(a) ブラウザ画面表示

```
<div id="maintable"><table><tbody>
<tr>
<td align="center">IIDX ID</td>
<td align="center">
<input size="16" id="iidxid" value=""
 onchange="sendIptVal('iidxid')"
 type="text"></td>
</tr>
<tr>
<td colspan="2" align="center">
<input value="Login" onclick="runJsFunc('Login,')"
 type="button"></td>
</tr>
</tbody></table></div>
```

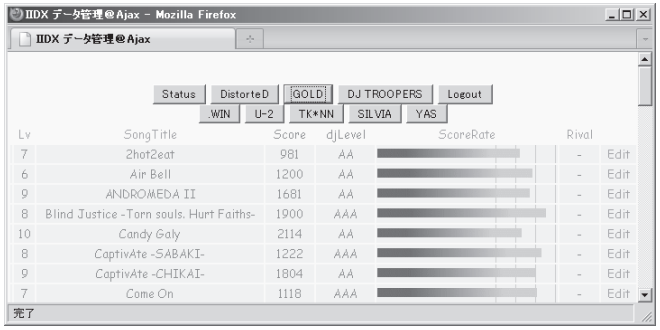
(b) SCvanisher 導入時の HTML ソース

```
<div id="maintable"><table><tbody>
<tr>
<td align="center">IIDX ID</td>
<td align="center">
<input size="16" id="iidxid" value=""
 type="text"></td>

</tr>
<tr>
<td colspan="2" align="center">
<input value="Login" onclick="Login()"
 type="button"></td>
</tr>
</tbody></table></div>
```

(c) SCvanisher 非導入時の HTML ソース

図 4　SCvanisher の動作確認 (1)
Fig. 4　Snapshot of SCvanisher (1).



(a) ブラウザ画面表示

```
<div id="uptable">
<input value="Status"
 onclick="runJsFunc('MakePlayerTable,u_data')"
 type="button">
...
</div>
<div id="midtable">
<input value=".WIN"
 onclick="runJsFunc('UpdateSongPack,14,rvl1')"
 type="button">
...
</div>
<div id="maintable"><table width="698">
<tbody><tr>
...
<td align="center">
<a href="javascript:runJsFunc('test1,14')">Lv</a></td>
```

(b) SCvanisher 導入時の HTML ソース

```
<div id="uptable">
<input value="Status"
 onclick="MakePlayerTable(u_data)"
 type="button">
...
</div>
<div id="midtable">
<input value=".WIN"
 onclick="UpdateSongPack(14,rvl1)"
 type="button">
...
</div>
<div id="maintable"><table width="698">
<tbody><tr>
...
<td align="center">
<a href="javascript:test1(14)">Lv</a></td>
```

(c) SCvanisher 非導入時の HTML ソース

図 5　SCvanisher の動作確認 (4)
Fig. 5　Snapshot of SCvanisher (4).

4.2　定量的評価

4.2.1　処 理 時 間

低負荷時のサーバと高負荷時のサーバにおいて，SCvanisher の処理により発生するオーバヘッドの測定を行った．測定を行ったのは，SCvanisher の中継部が JavaScript 実行部から JavaScript 実行後の特定の行数のコードを取得しはじめてから，再構築を完了するまでの時間である．入出力部への送信に要する時間は，利用者の通信環境により大きく左右されるために，ここでの計測時間には含ませない．

一般に，サーバの負荷を示す Load Average の値が CPU のコア数を超えると，処理が重く感じられるとされている．今回の計測では，低負荷時の計測においては Load Average の値を 0.1 程度に，高負荷時の計測においては Load Average の値を 2.5 程度に設定し，それぞれの場合について 100 行から 1,000 行まで，100 行ごとの処理時間を測定した．

測定に用いたサーバの環境は次のとおりである．

- CPU: Intel Pentium4 3.0 GHz
- RAM: 1,024 MB
- OS: Debin GNU/Linux 5.0.3

測定結果を**図 6** および**表 2** に示す．

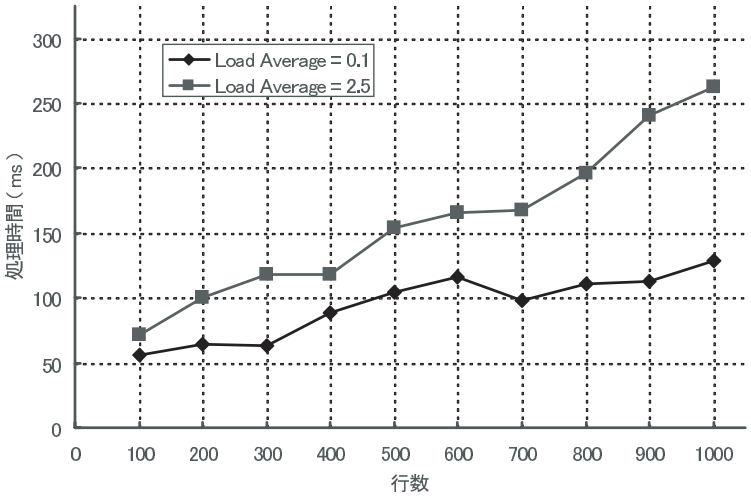


図 6　中継部がコードを取得・変換するのに要する時間
Fig. 6　Execution times in relay part.

表 2　中継部がコードを取得・変換するのに要する時間（単位：ミリ秒）
Table 2　Execution times in relay part (in milliseconds).

	100 行	200 行	300 行	400 行	500 行
Load Average : 0.1	56.0	64.7	63.4	89.0	104.5
Load Average : 2.5	71.9	100.3	118.1	118.4	153.9
LA2.5 / LA0.1	1.3	1.5	1.9	1.3	1.4
	600 行	700 行	800 行	900 行	1,000 行
Load Average : 0.1	116.2	98.2	110.4	112.9	129.1
Load Average : 2.5	165.2	167.6	196.4	240.7	262.7
LA2.5 / LA0.1	1.4	1.7	1.8	2.1	2.0

低負荷時，高負荷時のどちらにおいても，オーバヘッドは処理を行うソースコードの行数に比例することが確認できる．中継部の行う処理では，取得したソースコードの全体を走査しているためこの結果は妥当であるといえる．また，同じ行数の処理時間で比較すると，高負荷時におけるオーバヘッドは低負荷時の 1.3 倍から 2.1 倍となることが分かる．このことから，SCvanisher の動作レスポンスには少なからずサーバの負荷状況が影響を与えることが分かる．ただし，高負荷時においても中継部が処理に要する時間は，1,000 行の変換を行った場合でも 262.7 ミリ秒と，通信にかかる時間で十分に隠れる程度のオーバヘッドで済んでいる．実際の処理で 1 度に 1,000 行ものソースコードが生成されることはほぼないため，中継部の処理速度が SCvanisher 導入にあたって問題となることはないと結論づけることができる．

表 3　通信回数と転送データ量（単位：バイト，送信量と受信量は，サーバあるいは中継部から見た値）
Table 3　Number of communications and amounts of transferred data.

	SCvanisher 非導入時			SCvanisher 導入時					
	サーバ/クライアントブラウザ間			中継部/クライアントブラウザ間			中継部/サーバブラウザ間		
操作	回数	受信量	送信量	回数	受信量	送信量	回数	受信量	送信量
(1)	2	766	7,299	4	1,746	2,687	2	792	7,299
(2)	23	10,393	14,299	3	1,737	1,278	23	10,692	14,299
(3)	3	1,207	4,512	5	2,390	9,389	3	1,246	4,512
(4)	0	0	0	2	1,160	5,040	0	0	0
(5)	0	0	0	3	1,745	5,140	0	0	0
(6)	1	979	829	2	1,147	4,882	1	992	829
(7)	0	0	0	2	1,156	4,418	0	0	0
(8)	0	0	0	2	1,156	4,872	0	0	0
(9)	0	0	0	2	1,147	739	0	0	0
合計	29	13,345	26,939	25	13,384	38,445	29	13,722	26,939

4.2.2　転送データ量

SCvanisher による転送データの増加量を評価するため，前述のゲームスコア管理のウェブアプリケーションを用い，SCvanisher 非導入時と SCvanisher 導入時の双方について，転送データ量を測定した．行った操作は，4.1.2 項で述べた (1)〜(8) である．

SCvanisher 非導入時，導入時ともに，クライアントのブラウザは Firefox を利用した．またウェブサーバは Apache で，Apache のアクセスログからデータ転送量を採取した．ただし，ウェブアプリケーションの動作には本質的には関係ない `favicon.ico` の取得は，データ転送量からは除いてある．結果を**表 3** に示す．表の最左カラムは，操作手順の番号に対応している．

操作 (2) において，中継部/クライアントブラウザ間のデータ送信量が 1,278 バイトと，中継部/サーバブラウザ間の送信量 14,299 バイトより少ないのは，データ管理アプリケーションがサーバから全データを読み込むのに対し，SCvanisher は表示に必要な結果だけを転送するようにしているためである．

SCvanisher では，サーバ上で中継部とサーバブラウザが動作しているので，中継部/サーバブラウザ間は同一コンピュータ内の通信である．したがって，SCvanisher 導入時は中継

部/クライアントブラウザ間の通信，SCvanisher 非導入時はサーバ/クライアントブラウザ間の通信においてネットワークを介したデータ送受信が起こる．SCvanisher 導入時と非導入時のこのデータ転送量の比は，$(13{,}384 + 38{,}445)/(13{,}345 + 26{,}939) = 1.29$ であり，この実験においては SCvanisher 利用時の転送量増加は 30%未満であることが分かる．データ転送量増加の割合はウェブアプリケーションに依存するものの，この実験結果を見ると，あまり大きな増加にはならないと考えられる．

また，クライアントのブラウザが中継部に対してリクエストを送信してから実際に結果を受信するまでの往復時間は，サーバにおける処理時間とネットワーク上のデータ転送時間の和に支配される．前者は，4.2.1 項で示したように，通信時間に隠れる程度の時間であった．後者はサーバとクライアント間のネットワーク環境に大きく依存するが，おおむね転送データ量に比例する程度の時間が必要である．転送データ量は上で見た程度の増加量であったことから，往復時間に関しても，利用に支障をきたすような問題はないと判断できる．

5. SCvanisher の限界と今後の課題

SCvanisher の限界は，仕様上の限界と実装上の限界に分けて考えることができる．

仕様上の限界は，従来はクライアントブラウザで実行していた JavaScript コードをサーバ上の JavaScript 実行部において代理実行させることに起因する．SCvanisher を利用した場合，クライアントのブラウザに表示される内容は，中継部が JavaScript 実行部から取得した瞬間のものである．これにはウェブアプリケーション固有のコードは含まれないため，JavaScirpt 実行部から再取得を行わない限り，利用者の操作をともなわない処理は利用者側に反映されない．したがって，たとえばマウスカーソルを乗せると画像が変化するようなマウスオーバについては，画像変化の開始・終了の瞬間は SCvanisher で対応できるが，マウスカーソルの軌跡に合わせて連続的に描画するようなことには対応しない．また，ゲームのように利用者の操作をともなわない画面変化が起き続けるものにも対応できない．

SCvanisher では，あらゆる操作に対して必ずサーバとの通信が発生するため，操作が実際に反映されるまでにタイムラグが生じる．このため，レスポンスが重要なアプリケーションへの適用には向いていない．

逆にいえば，上であげたような限界が問題にならないウェブアプリケーションに対しては，SCvanisher は問題なく動作する．そのようなアプリケーションの例としては，4.1 節でとりあげたデータ管理や，Wiki のようなものがあげられる．また，音声，画像などのマルチメディアファイルに対しては SCvanisher は手を加えないので，SCvanisher 導入による影響は出ない．したがって，たとえば Flash による動画も問題なく再生される．

実装上の最大の限界は，サーバにおける JavaScript 実行部として，サーバブラウザである Firefox と MozRepl を利用したことに起因する．MozRepl の仕様上，Telnet を介して操作できる Firefox は，一番最後にアクティブ状態に移行したプロセスに限定されるため，同時利用クライアント数は 1 に制限されてしまう．この制限を外し，複数人のウェブアプリケーションの同時利用へ対応することは，今後の大きな課題である．そのため，JavaScript 実行部として利用する Firefox のプロセスは 1 つに限定し，タブ機能を用いた切替えが必要である．MozRepl によるタブの切替えは可能であるため利用者の IP アドレスとタブの紐付けを行うことで実現できると考えられる．

さらに，現在の実装では，`prototype.js` のようなライブラリを用いて，動的に追加されるイベントハンドラには対応していないため，このようなライブラリへの対応も必要である．

現在のところ SCvanisher は，DoS 攻撃への対応などのセキュリティに関する検討は十分にはなされていない．4.2.2 項で調べた SCvanisher 導入によるデータ転送量増加の程度から判断すると，DoS 攻撃に対する耐性は SCvanisher 非導入時と大きな差はないものと考えられるが，この問題を含めたセキュリティ問題の詳細な検討も，今後の課題である．

6. ま と め

本論文では，ウェブアプリケーションをサーバ上で実行し，その結果のみをクライアントに送信することで，従来は秘匿することができなかったウェブアプリケーションのソースコードの隠蔽を行う機構である SCvanisher を提案した．SCvanisher を利用したウェブアプリケーションの提供を考えた際であっても，記述にあたって SCvanisher 特有の記法は存在しないため，開発者の負担にはならない．SCvanisher の有効性を示すため，従来であれば情報漏洩の危険性を持った手法で記述されたウェブアプリケーションを SCvanisher を通して設置，実行する実験を行った．この結果，ウェブアプリケーションのソースコードはサーバ外から読むことが不可能であることを確認した．これにより，知的財産の保護および，ソースコード盗用に対する堅牢性の確保の手段として SCvanisher が有用であることがいえる．また，SCvanisher を導入することで生じるオーバヘッドと転送データ増加量の評価も行った．

SCvanisher の基本的なアイディア，すなわち JavaScript コードをクライアントではなくサーバにおいて実行し，実行結果をクライアントに転送するという方法は，ソースコード隠蔽以外の応用の可能性がある．たとえば，ブラウザ表示のマルチキャストへの利用，Ajax

のクロスブラウザ問題への対処などが考えられる．このような新たな応用範囲を開拓することも，今後の課題である．

謝辞　本論文における実験にご協力いただいた鵜川始陽氏，山田佑二氏に慎しんで感謝の意を表する．

参 考 文 献

1) Solutions, P.: Dotfuscator (2003). http://www.preemptive.com/products/dotfuscator/overview/
2) 株式会社プランセス：SHTML (2005). http://www.shtml.jp/
3) 村山隆徳，満保雅浩，岡本栄司，植松友彦：ソフトウェアの難読化について，電気情報通信学会技術研究技報（情報セキュリティ），ISEC95-25, pp.9–14 (1995).
4) 門田暁人，高田義広，鳥居宏次：ループを含むプログラムを難読化する方法の提案，電子情報通信学会論文誌，Vol.J80-D-1, No.7, pp.644–652 (1997).
5) Cerven., P.: *Crackproof Your Software — The Best Ways to Protect Your Software Against Crackers*, No Starch Press (2002).
6) Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S. and Yang., K.: On the (Im)possibility of Obfuscating Programs, Lecture Notres in Computer Science, Vol.2139, pp.1–18 (2001).
7) 加藤勇也：bRuby (2002). http://bruby.sourceforge.jp/
8) 加藤勇也：Exerb (2002). http://exerb.sourceforge.jp/
9) 竹内郁雄：独立行政法人情報処理推進機構 平成 14 年度未踏ソフトウェア創造事業採択案件評価 (2002). http://www.ipa.go.jp/NBP/14nendo/14youth/mdata/2-1.htm
10) Aptana Jaxer (2008). http://www.jaxer.org/
11) Hanakawa, N. and Ikemiya, N.: A web browser for Ajax approach with asynchronous communication model, *Proc. 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pp.808–814 (2006).

(平成 22 年 2 月 15 日受付)
(平成 22 年 5 月 9 日採録)

折戸　隆洋

1984 年生．2008 年電気通信大学電気通信学部情報工学科卒業．2010 年電気通信大学大学院電気通信学研究科情報工学専攻博士前期課程修了．プログラミング言語とその記述環境，特にスクリプト言語に興味を持つ．

岩崎　英哉（正会員）

1960 年生．1983 年東京大学工学部計数工学科卒業．1988 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年同大学計数工学科助手．1993 年同大学教育用計算機センター助教授．その後，東京農工大学工学部電子情報工学科助教授，東京大学大学院工学系研究科情報工学専攻助教授，電気通信大学情報工学科助教授を経て，2004 年より電気通信大学教授．工学博士．記号処理言語，関数型言語，システムソフトウェア等の研究に従事．日本ソフトウェア科学会，ACM 各会員．