Better Abstractions for Secure Server-Side Scripting

Dachuan Yu Ajay Chander Hiroshi Inamura Igor Serikov
DoCoMo Communications Laboratories USA
3240 Hillview Avenue
Palo Alto, CA 94304
{yu,chander,inamura,iserikov}@docomolabs-usa.com

ABSTRACT

It is notoriously difficult to program a solid web application. Besides addressing web interactions, state maintenance, and whimsical user navigation behaviors, programmers must also avoid a minefield of security vulnerabilities. The problem is twofold. First, we lack a clear understanding of the new computation model underlying web applications. Second, we lack proper abstractions for hiding common and subtle coding details that are orthogonal to the business functionalities of specific web applications.

This paper addresses both issues. First, we present a language BASS for declarative server-side scripting. BASS allows programmers to work in an ideal world, using new abstractions to tackle common but problematic aspects of web programming. The meta properties of BASS provide useful security guarantees. Second, we present a language MOSS reflecting realistic web programming concepts and scenarios, thus articulating the computation model behind web programming. Finally, we present a translation from BASS to MOSS, demonstrating how the ideal programming model and security guarantees of BASS can be implemented in practice.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics*; K.4.4 [**Computers and Society**]: Electronic Commerce—*security*

General Terms

Languages, Security

Keywords

Server-side scripting, web application security

1. INTRODUCTION

Web applications face more security threats than conventional desktop applications [7, 19]. Some representative ones include command injection [20], cross-site scripting (XSS) [16], cross-site request forgery (CSRF) [4], and session fixation [14]. Any of these could cause serious consequences: sensitive user information could be stolen, data and associated belongings could be damaged, or service availability could be compromised.

In response to such security threats, existing languages and frameworks for server-side scripting, as well as the web application security community, largely promote secure coding practices (*e.g.,* input validation) and provide useful libraries in support (*e.g.,* filtering functions). However, there is no guarantee that programmers will follow the recommendations correctly, if they followed them at all. Furthermore, even if all programs are written with the security practices strictly enforced, the extra care that programmers spend on preventing vulnerabilities much distracts from the business functionalities. Take the implementation of online payments as an example. To securely code a web interaction of "obtaining payment details," one must correctly perform input validation, maintain program states across the interaction, and prevent CSRF.

Observing that many of the security issues are orthogonal to the business logic of specific web applications, we propose some new abstractions for writing secure server programs. These abstractions provide an ideal view of key aspects of web programming (*e.g.,* "to obtain a web input"), and hide the common security handling (*e.g.,* input validation, state maintenance, CSRF prevention). Using these abstractions, a language for server-side scripting can be given a high-level syntax and semantics that reflect secure web operations, with the enforcement of the semantics taken care of by the language implementation following established security practices once and for all. As a result, all programs written in the language, although not directly dealing with low-level security constructs, are guaranteed to be free of certain common vulnerabilities. In addition, now thinking in terms of the high-level semantics, programmers can focus more on the business logic, which results in better security and higher productivity. To some extent, the new abstractions hide security details in the same way as object creation primitives in OO languages hide low-level memory management details.

On the technical side, this paper presents two formal models (languages): one is an ideal model that we propose for future web programs to be written in; the other is the real-world model that underlies existing web programming. By comparing these two models side-by-side, and by formalizing a translation between them, we argue that web programming could benefit significantly through the use of domain-specific language abstractions. Specifically, this paper makes the following contributions.

First, we identify some common aspects of web programming that are important for security, and propose new abstractions to help implement the corresponding tasks (Section 2). The new abstractions are rigorously presented in a self-contained formal language BASS ("better abstractions for server-side scripting") (Section 3). The new BASS primitives and operational semantics shield programmers from certain malicious exploits, thus serving as an ideal model of web programming. The security guarantees of BASS are articulated in its formal semantics and meta properties.

Next, we formalize a language MOSS ("a model of server-side scripting") that reflects the programming model underlying existing

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

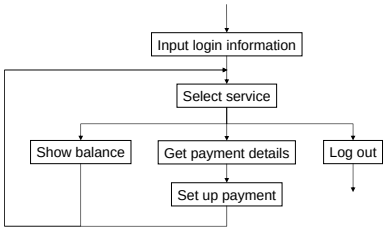

Figure 1: Idealistic work flow of online banking

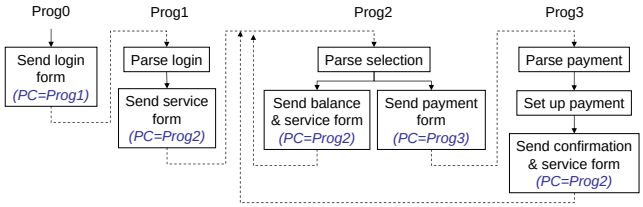

Figure 2: Actual work flow of online banking

server-side scripting (Section 4). This demonstrates the complexity of web programming and the benefit of the BASS abstractions. In addition, MOSS is of independent value—it articulates server program execution, attacker exploits, and client-server interactions all within the same model, thus serving as a foundation for studies of web programming. MOSS can be used to illustrate both secure programs and vulnerable ones. We give formal encoding of common exploit patterns and typical attacks as an example use of MOSS.

Finally, the BASS approach to security is that the operational semantics dictates "good" behaviors only, thus leaving no room for certain attacks. A BASS compiler must carry out the operational semantics correctly. We present a translation from BASS to MOSS (Section 5), where the ideal BASS model is enforced using proper MOSS primitives for security manipulations. We explain how the security guarantees of BASS are maintained. More formal translation details are given in a companion technical report (TR) [24].

2. SERVER-SIDE SCRIPTING & SECURITY

2.1 An Ideal View

We use an online-banking example to explain what is involved in secure server-side scripting, and how proper abstractions can help. This application provides two services: showing the account balance (the "balance" service) and setting up a payment (the "payment" service). A user must be logged in to access the services.

Although serving multiple users, this web application logically deals with one client at a time. In an ideal view, there are multiple orthogonal instances of the server program running, each taking care of a single client. Every single instance of the program can be viewed as a sequential program of a conventional application. A work flow graph following this ideal view is shown in Figure 1.

2.2 A Limited Mechanism

The above ideal view cannot be directly implemented, because of some limitations of the underlying HTTP mechanism for web interactions. In particular, there is no persistent channel for a server program to obtain input from a client. Instead, HTTP supports a one-shot request-response model where a client requests resource identified by a URL, and a server responds with the resource if the request is accepted. Using HTTP, web interactions are typically carried out as a sequence of requests (form submissions providing user input) and responses (web pages presenting information):

$\texttt{HTTP}\ 0 \rightarrow \texttt{HTML}\ 0 \rightarrow \texttt{HTTP}\ 1 \rightarrow \texttt{HTML}\ 1 \rightarrow \texttt{HTTP}\ 2 \rightarrow \texttt{HTML}\ 2 \ldots$

Using this model, a server program is often split into multiple fragments, each taking an HTTP request and producing an HTML response. In the response, there can be a web form with an embedded URL pointing to the next fragment, so that the next request is targeted correctly. Therefore, the work flow of our banking application is more accurately described as in Figure 2. There are 4 program fragments connected with URL embeddings, as indicated by the dashed lines. In particular, because the payment service requires user input, the structure of the service loop in the ideal view can no longer be coded as an explicit loop. Instead, a goto-style structure is exhibited through URL embeddings. Such fragmentation and low-level control structures obscure the control flow.

Besides obscurity, there is a bigger issue: since HTTP is stateless, server programs must maintain program states on their own. In the example, the user name obtained from the login input must be saved and restored explicitly across the later web interactions. In addition, one must somehow correlate incoming requests with specific clients, since multiple clients may be interacting with the server at the same time, although in logically separate transactions.

In general, a web application needs to encode states at a level above HTTP. Typically, a virtual concept of "a session" is used to refer to a logical transaction. Every session is associated with a unique ID, called SID. Saved program states and incoming client requests are both identified by the SID. As a result, much code in server programs is dedicated to managing sessions. Before generating a response, a server program must save state and embed the SID in the response. Upon receiving a request, the server program must obtain an SID from the request and load state. Based on the application, some parts of the state should be saved on the server, whereas others should be on the client via the cookie or URL embedding. These routine manipulations increase program complexity, reduce productivity, and extend the chances of programming errors.

2.3 A Dangerous World

Assuming a programmer has taken care of the above issues correctly, the result program may still not be ready for deployment. The problem is security: clients in the real world may be malicious, or attackers may trick innocent clients into making mistakes. Indeed, there have been many common vulnerabilities identified. Secure programming solutions exist, but a programmer must be aware of all the issues involved and implement the related defenses. Most of the defenses are orthogonal to the business logic of specific web applications. Their handling further complicates server-side scripting. We now briefly overview some representative security issues.

CSRF: An attacker may forge a request as if it were intended by a user. This is applicable when SIDs are stored in cookies. Given a vulnerable banking program, CSRF can be launched if a user, while logged in, opens a malicious email containing a crafted image link. Trying to load the image, the user's browser may follow the link and send a request asking for a payment to be set up to the attacker.

XSS: An attacker may inject code into the web page that a server program sends to a user. For example, an attacker sends to a user a crafted link with JavaScript code embedded; when the user loads the link, a vulnerable server program may propagate the code into the HTML response. The code, now coming from the server, gains the privilege of the server domain. It may then read the cookie set by the server and send it to the attacker. There are also second-order attacks [18] that do not require the use of forged requests.

Session fixation: An attacker may trick a user into interacting with the server using a fixed SID. This is applicable if SIDs are

embedded in URLs. A user may follow a link in an email which claims to be from our banking site. The link takes the user to our site, but using an SID fixed by an attacker. If the server programs use the same SID for later interactions after the user logs in, the knowledge of the SID will grant the attacker the user's privileges.

Others: Many other aspects affect security [19]. Since a web application is implemented as multiple program fragments, each fragment is open as a service interface. An attacker could make up requests to the interfaces without following the links in server responses. Using crafted requests, they could poison program states (*e.g.,* by modifying naïve implementations of client-side states), inject malformed input (*e.g.,* by exploiting insufficient input validation), or circumvent work flows (*e.g.,* by using the "back" button).

Programmers need to be aware of all these issues and follow the relevant security practices. In the result code, business logic is intertwined with security manipulations. Consequently, secure web programming is difficult, and web programs are hard to maintain.

2.4 A Declarative Layer

We propose to program web applications using a language that handles common security aspects behind the scene. This language benefits from new abstractions designed for web programming. A program in this language more directly reflects the business logic of the application; therefore, it is easier to write, to reason about, and to maintain. The language implementation (the compiler) will generate secure target code following established security practices.

Abstracting web interactions Much of the complication is due to the need of obtaining user input. Therefore, supporting web interactions is a key. We introduce a dedicated construct `form` for this:

$$\texttt{form}(\texttt{p} : \text{“}\texttt{username}\text{”}, \texttt{q} : \text{“}\texttt{password}\text{”});$$

The intention of this is to present an HTML page to the client and obtain some input. In a realistic deployment, such a construct would take as argument a full-fledged HTML document. In this paper, we simply let it take arguments to describe input parameters of a web form. In the above example, the `form` construct presents to the client a form with two fields: `username` and `password`. After the client fills in the form and submits it, the `form` construct assigns the values of the fields to the two variables `p` and `q`.

A few issues are handled transparently by the implementation. First, the server program is split upon a `form` construct, and URL embedding is used to connect the control flow. Second, input values are parsed from the form submission, and input validation is performed according to declared variable types. Third, security practices are followed to manage sessions, maintain states, and defend against common exploits. Details of the implementation will be discussed in Section 5. For now, it suffices to understand this construct from a programmer's point of view as an abstract and secure web input operation which does not break the control flow.

Supporting user navigation The `form` construct implicitly opens a service interface for receiving user requests. There would be vulnerabilities if it were not handled properly, or if its handling were not fully understood by the programmer. Previous work (*sans* security) [3, 5, 6] on web interaction abstractions requires the interface be "open" only once—a second request to the interface will be rejected. This much restricts user navigation [11, 12, 21]. In practice, it is common for a user to return to a previous navigation stage using the "back" button. In general, the user could revisit any item in the browser history. The validity of such an operation should be determined by the application, rather than be dismissed all together.

We allow two modes of web interactions: a *single-use* mode ($\texttt{form}_\texttt{S}$) and a *multi-use* mode ($\texttt{form}_\texttt{M}$). In the former, the interface is open for request only once; revisiting the interface results in an error. In the latter, the interface remains open for future requests. The semantics of BASS articulates the program behavior in both cases; therefore, the programmer can choose the suitable one based on the application. In either case, a request is accepted only if it follows the intended control flow of the server program to the interface. Consider our banking example. It is okay if the user reached the service selection page, bookmarked it, and reused it before logging out. However, it is not okay if the user forged a payment request without first going through the login page.

```
string  user, pass, payee;
int     sel, amnt;
formS(user : "username", pass : "password");
LoginCheck(user, pass);
while (true) do {
  formM(sel : "1 : balance; 2 : payment; others : logout");
  if (sel == 1)
  then ShowBalance(user)
  else if (sel == 2)
       then {formS(payee : "payee"; amnt : "amount");
             DoPayment(user, payee, amnt)}
       else {clear; break}
}
```

Figure 3: Simple banking in BASS

Maintaining program states Multi-use forms are sufficient to accommodate all client navigation behaviors, because any behavior can be viewed as revisiting a point in the browser history. From a programmer's point of view, the program is virtually forked into multiple parallel "threads" at a multi-use form, together with all appropriate program state. The handling of the program state is nontrivial. Some parts of the state could be local to the individual threads, whereas others could be global to all threads. Careless treatment of the state may result in logical errors [12].

The exact partitioning of the state should be determined by the application. We let programmers declare mutable variables as either *volatile* or *nonvolatile*. In the BASS implementation, volatile state can be stored in a database on the server across web interactions, thus all forked threads refer to the same volatile state. In contrast, nonvolatile state (after proper protection against client modifications) can be embedded in the URLs of web forms upon web interactions, thus every forked thread has its own nonvolatile state.

Manipulating client history Suppose the user tries to reload the service selection page after logging out of our banking application. The server program will receive a request that should not be processed. In general, we need a mechanism to disable some of the entries in the client history. In existing web applications, this is sometimes handled by embedding special tokens into web forms and checking them upon requests. While logging out, the server program expires the corresponding token, thus further requests to the service selection page will no longer be processed.

We do not wish to expose the details of token embedding to the programmer. Instead, we introduce a `clear` command for a similar purpose. From the programmer's point of view, `clear` resets the client history, so that all previously forked threads are discarded. This corresponds to the "session timeout" behaviors of many web applications. However, instead of thinking in terms of disabling the token of SID, BASS encourages programmers to think in terms of the client history. The TR discusses more general ways to manipulate the client history, which introduce no new difficulties.

Example revisited Figure 3 demonstrates the appeal of these abstractions by revisiting our banking example. The new abstractions provide an ideal world where there is only one client and the client

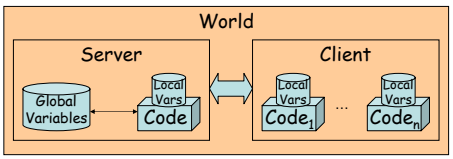

Figure 4: A virtual view of the execution environment

$$
\begin{array}{rrl}
(World) & W & ::= (\Sigma, \kappa, \vec{\kappa}) \\
(Global\ Env) & \Sigma & ::= \{\vec{a} : \vec{\tau} = \vec{v}\} \\
(Closure) & \kappa & ::= (\sigma, C) \mid \sigma \\
(Local\ Env) & \sigma & ::= \{\vec{x} : \vec{\tau} = \vec{v}\} \\
(Type) & \tau & ::= \texttt{int} \mid \texttt{str} \\
(Command) & C & ::= \texttt{skip} \mid a := E \mid x := E \mid C; C \\
 & & \quad \mid \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ E\ \texttt{do}\ C \\
 & & \quad \mid \texttt{form}_{\texttt{S}}(\vec{a}:\vec{s}) \mid \texttt{form}_{\texttt{M}}(\vec{a}:\vec{s}) \mid \texttt{clear} \\
(Exp) & E & ::= a \mid x \mid i\ (integers) \mid s\ (strings) \mid op(\vec{E}) \\
(Value) & v & ::= i \mid s
\end{array}
$$

Figure 5: BASS syntax

is well behaved. In the code, we obtain login information from the client, perform login check, and proceed with a service loop. In the loop, based on the service selection of the client, we carry out the balance service or the payment service, or log the user out. The service selection input is coded using a multi-use form, therefore the user may duplicate the corresponding web page and proceed with the two services in parallel. In addition, `clear` is used to disable all service threads when the user logs out. In this example, only the $user$ variable is live across web interactions. Its value is obtained from a single-use form, and will not be updated after the login process. Therefore, it can be declared as either volatile or nonvolatile.

This code corresponds well to the work flow of Figure 1, and is much cleaner than a version written in an existing language. More importantly, it does not sacrifice security, because the BASS implementation will take care of the "plumbings" transparently—it will split the program into fragments, maintain states across web interactions, filter input based on variable types, and carry out relevant security manipulations such as the embedding of secret tokens.

3. BASS

We formalize the above ideas in BASS, which provides a declarative view for programmers to write secure programs without worrying about the low-level details of Section 2. Assuming the semantics is enforced by an implementation, all well-formed BASS programs will be secure in a certain sense. For example, values obtained from web input will have the expected types, forged requests from attackers will be rejected, and the program control flow will be enforced. These properties will be articulated in Section 3.3.

3.1 Syntax

Figure 4 gives a graphical depiction of the virtual execution environment of BASS. We emphasize the word "virtual" because the view does not dictate how BASS is implemented; it simply provides a tractable world for the programmer. This world has one server and one client. The server consists of a global environment for volatile variables and a closure of the current execution. The closure is made up of a local environment for nonvolatile variables and some code. During execution, the server executes the code with respect to the two environments. Upon a web interaction, the server sends a proper closure to the client, and the client is then activated.

The client involves a list of closures reflecting the browsing history. When the client is active, it chooses an arbitrary closure in

$\boxed{W \rightsquigarrow W'}$

Server computation:

$$\frac{(\Sigma, \kappa) \rightsquigarrow (\Sigma', \kappa')}{(\Sigma, \kappa, \vec{\kappa}) \rightsquigarrow (\Sigma', \kappa', \vec{\kappa})} \quad (1)$$

History clearing:

$$(\Sigma, (\sigma, \texttt{clear}; C), \vec{\kappa}) \rightsquigarrow (\Sigma, (\sigma, C), \epsilon) \quad (2)$$

$$(\Sigma, (\sigma, \texttt{clear}), \vec{\kappa}) \rightsquigarrow (\Sigma, \sigma, \epsilon) \quad (3)$$

Server response:

$$\frac{C \in \{(\texttt{form}_{\texttt{S}}(\vec{a}:\vec{s}); C'), \texttt{form}_{\texttt{S}}(\vec{a}:\vec{s}), (\texttt{form}_{\texttt{M}}(\vec{a}:\vec{s}); C'), \texttt{form}_{\texttt{M}}(\vec{a}:\vec{s})\}}{(\Sigma, (\sigma, C), \vec{\kappa}) \rightsquigarrow (\Sigma, \sigma, [(\sigma, C)] \cup \vec{\kappa})} \quad (4)$$

Client request:

$$\frac{\kappa_i \in \vec{\kappa} \qquad \kappa_i = (\sigma, (\texttt{form}_{\texttt{S}}(\vec{a}:\vec{s}); C)) \qquad (\vec{a} : \vec{\tau} = \ldots) \in \Sigma \qquad \vec{v} \in \vec{\tau} \qquad update(\Sigma, \vec{a}, \vec{v}) = \Sigma'}{(\Sigma, \sigma_0, \vec{\kappa}) \rightsquigarrow (\Sigma', (\sigma, C), \vec{\kappa} - [\kappa_i])} \quad (5)$$

$$\frac{\kappa_i \in \vec{\kappa} \qquad \kappa_i = (\sigma, (\texttt{form}_{\texttt{S}}(\vec{a}:\vec{s})))}{(\Sigma, \sigma_0, \vec{\kappa}) \rightsquigarrow (\Sigma, \sigma_0, \vec{\kappa} - [\kappa_i])} \quad (6)$$

$$\frac{\kappa_i \in \vec{\kappa} \qquad \kappa_i = (\sigma, (\texttt{form}_{\texttt{M}}(\vec{a}:\vec{s}); C)) \qquad (\vec{a} : \vec{\tau} = \ldots) \in \Sigma \qquad \vec{v} \in \vec{\tau} \qquad update(\Sigma, \vec{a}, \vec{v}) = \Sigma'}{(\Sigma, \sigma_0, \vec{\kappa}) \rightsquigarrow (\Sigma', (\sigma, C), \vec{\kappa})} \quad (7)$$

$$\frac{\kappa_i \in \vec{\kappa} \qquad \kappa_i = (\sigma, (\texttt{form}_{\texttt{M}}(\vec{a}:\vec{s})))}{(\Sigma, \sigma_0, \vec{\kappa}) \rightsquigarrow (\Sigma, \sigma_0, \vec{\kappa})} \quad (8)$$

Figure 6: BASS operational semantics

the history, fills in the corresponding web form, and sends the result to the server. We point out that although many code pieces appear in the closures on the client, they would actually be realized in an implementation as program pointers embedded in the forms. Similarly, the local environments can also be realized in forms.

Figure 5 gives the syntax. We use the vector $\vec{\kappa}$ to refer to the list $[\kappa_1 \ldots \kappa_n]$. Similar notations apply to other meta variables. A world W is a 3-tuple $(\Sigma, \kappa, \vec{\kappa})$. The first element Σ is the global variable environment, which is a mapping from volatile variables to their types and values. The second element κ is the closure currently active on the server. The third element $\vec{\kappa}$ is the client history.

A closure is usually a pair of a local environment and a command. Sometimes, an environment without a command also makes a closure, which either contains useful information or signifies termination. A local environment maps nonvolatile variables to types and values. We assume that the names of volatile and nonvolatile variables are disjoint. By convention, we use $a, a', a_1, \ldots$ for volatile variables, and $x, x', x_1, \ldots$ for nonvolatile variables. We use two types: integers and strings. Booleans are simulated with integers.

Commands are common except for web interactions ($\texttt{form}_{\texttt{S}}$, $\texttt{form}_{\texttt{M}}$) and history clearing (`clear`) as described in Section 2.4. For conciseness, web forms only input volatile variables; this does not affect expressiveness, because the input values can be passed on to nonvolatile variables with assignments. Expressions and values are as expected. We use op to abstract over all operations free from side-effects, such as string concatenation and comparison. We omit function calls for space, because they introduce no new issues.

3.2 Operational Semantics

We use a big-step semantics for expressions ($\Sigma; \sigma \vdash E \Downarrow v$). Details are standard, thus omitted. We present in Figure 6 the small-step execution of a world. The key concept is a "world step"

$$\boxed{\vdash W} \qquad \frac{\Sigma \vdash \kappa \qquad \forall \kappa_i \in \vec{\kappa}. \begin{cases} \kappa_i = (\sigma_i, C_i) & \Sigma \vdash \kappa_i \\ C_i \text{ starts with } \texttt{form}_\texttt{S} \text{ or } \texttt{form}_\texttt{M} \end{cases}}{\vdash (\Sigma, \kappa, \vec{\kappa})} \quad (9)$$

$$\boxed{\Sigma \vdash \kappa} \qquad \Sigma \vdash \sigma \quad (10)$$

$$\frac{\Sigma; \sigma \vdash C}{\Sigma \vdash (\sigma, C)} \quad (11)$$

Figure 7: Selected BASS typing rules

relation $W \rightsquigarrow W'$. A multi-step relation $W \rightsquigarrow^* W'$ is defined as the reflexive and transitive closure of the world step relation.

In world step, some rules reflect server-side computation, whereas others reflect client-side input. Rule (1) describes a server computation step by referring to a single-thread step relation $(\Sigma, \kappa) \rightsquigarrow (\Sigma', \kappa')$ (given in the TR). This is applicable only when the current command starts with skip, assignment, conditional, or loop (the single-thread step relation is undefined on other cases). If the current command is `clear`, Rules (2) and (3) apply to remove all closures from the client. In these rules and the remainder of this paper, we use ϵ to denote the empty vector. Upon any `form` commands, Rule (4) applies to transfer the closure to the client. The notation $[\kappa] \cup \vec{\kappa}$ means to combine the singleton vector $[\kappa]$ with $\vec{\kappa}$.

Client input through a single-use form is captured in Rules (5) and (6). Note that there is no further command to execute on the server (the closure on the server is simply σ_0), therefore the client takes control. In Rule (5), the client selects the closure κ_i to proceed with next. This corresponds to picking an arbitrary point in the browsing history. The client may input arbitrary values to the form, as long as they are of the right types ($\vec{v} \in \vec{\tau}$). The global environment is updated using the macro $update(\Sigma, \vec{a}, \vec{v})$, which updates Σ so that the variables $\vec{a}$ will have the values $\vec{v}$. As a the result of the step, the global environment is updated, and the chosen client closure is moved to the server for execution. The notation $\vec{\kappa} - [\kappa_i]$ means to remove κ_i from $\vec{\kappa}$. Rule (6) is a variant of Rule (5) where there is no further command in the chosen closure to execute after the form input. In this case, the chosen closure is simply removed.

Rules (7) and (8) are for client input through multi-use forms. The difference from the single-use ones lies in the result browsing history. For single-use forms, the selected closure is removed so that it cannot be revisited. For multi-use forms, the history remains the same. All client request rules are non-deterministic on closure selection. A client may "abandon" a closure by never selecting it.

3.3 Typing Rules and Meta Properties

We give some non-standard typing rules in Figure 7. In Rule (9), a world $(\Sigma, \kappa, \vec{\kappa})$ is well-formed if all closures (κ and $\vec{\kappa}$) are well-formed with respect to the global environment (Σ), and every closure (κ_i) on the client side contains code which starts with a web form. The well-formedness of closures is defined in Rules (10) and (11). A trivial closure σ is always well-formed. A nontrivial closure (σ, C) is well-formed with respect to Σ if C is well-formed with respect to Σ and σ. We omit the standard handling on commands ($\Sigma; \sigma \vdash C$) and expressions ($\Sigma; \sigma \vdash E$), where types of variables are pulled from the corresponding environments.

BASS enjoys standard type soundness via preservation and progress.

Lemma 1 (Preservation) If $\vdash W$ and $W \rightsquigarrow W'$, then $\vdash W'$.

Lemma 2 (Progress) If $\vdash W$ then either there exists W' such that $W \rightsquigarrow W'$, or W is of the form $(\Sigma, \sigma, \epsilon)$.

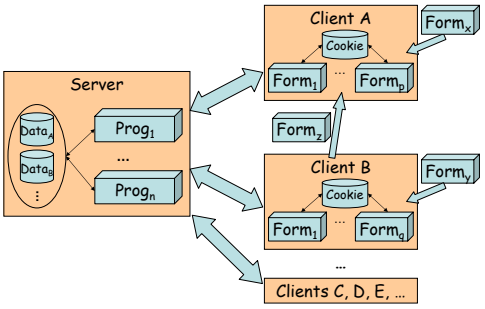

Figure 8: A model of web programming

As usual, these imply that well-formed programs will execute till termination without getting stuck. These seemingly simple theorems essentially guarantee that the program execution will be confined by the operational semantics—only the transitions defined by the operational semantics may occur. Therefore, all behaviors reflected by the operational semantics are guaranteed, including:

1. The control flow will follow the high-level structures;
2. Values obtained from web input will be well typed;
3. Program states will be kept intact across web interactions;
4. Entries in the client history will be disabled upon `clear`;
5. There will be only one client interacting with the program instance.

These free programmers from some low-level maneuvers, such as:

1. Embedding and enforcing the control flow;
2. Validating the names and types of web input parameters;
3. Recovering program states across web interactions;
4. Addressing unexpected user navigation behaviors;
5. Protecting against malicious exploits that involve multiple clients (*e.g.,* CSRF, first-order XSS, and session fixation).

We stress the point that the BASS approach to security is to directly preclude questionable behaviors from the operational semantics. BASS programmers enjoy a clean view of web programming, with common security handling transparently taken care of by the BASS implementation (the compiler). This enables programmers to focus on the business logic of specific web applications, thus improving security and productivity. Specifically on security, the BASS semantics leaves no room for CSRF, first-order XSS, session fixation, session poisoning, malformed input, and work-flow circumvention. Of course, the semantics must be enforced by an implementation. This is the topic of the next two sections, where we formalize a realistic web programming model MOSS, and translate BASS into MOSS so that the BASS semantics is faithfully carried out.

4. MOSS

MOSS articulates server program execution and client behaviors. It reflects realistic web interactions, and can be used to illustrate both well-behaved client activities and malicious exploits. We will explain how concepts in MOSS correspond to real-world entities.

4.1 Syntax

A graphical depiction of the model is given in Figure 8. On the server, there are a group of programs collaborating to implement the desired application logic. They access and update data that belong to multiple clients. The server interacts with multiple clients simultaneously, although in separate logical transactions. Every client has its own cookie, and maintains a list of forms as the browsing history. Although forms are expected to be obtained from the

$$
\begin{array}{rrl}
(\textit{World}) & \mathtt{W} & ::= (\dot{\Sigma}, \dot{\kappa}, \vec{\Phi}) \\
(\textit{Global Env}) & \dot{\Sigma} & ::= \{[\vec{\iota}\colon \vec{\varsigma}], [f = (\vec{a}) \triangleright \mathtt{C}]^*\} \\
(\textit{Session Env}) & \varsigma & ::= \{\vec{a} = \vec{\mathtt{v}}\} \\
(\textit{Closure}) & \dot{\kappa} & ::= (i, \mathtt{v}, \dot{\sigma}, \mathtt{C}) \mid (\mathtt{v}, \dot{\sigma}) \\
(\textit{Local Env}) & \dot{\sigma} & ::= \{\vec{x} = \vec{\mathtt{v}}\} \\
(\textit{Client}) & \Phi & ::= (\mathtt{v}, \vec{\phi}) \\
(\textit{Form}) & \phi & ::= \mathtt{form}\ f(\vec{s}\,\vec{a}{=}\vec{\mathtt{v}})\ \mathtt{with}\ \psi \\
(\textit{Sealed}) & \psi & ::= \langle \vec{x} = \vec{\mathtt{v}} \rangle \\
(\textit{Command}) & \mathtt{C} & ::= \mathtt{sk} \mid a \leftarrow E \mid \mathtt{C} \,;; \mathtt{C} \mid \mathtt{cond}(\mathtt{E}, \mathtt{C}, \mathtt{C}) \\
 & & \mid \mathtt{loop}(\mathtt{E}, \mathtt{C}) \mid f(\vec{\mathtt{E}}) \mid \mathtt{input}(\mathtt{E}) \mid \mathtt{setCk}(\mathtt{E}) \\
 & & \mid \mathtt{unpack}(\mathtt{E}) \mid \mathtt{verif}(\vec{\mathtt{E}}, \vec{\tau}) \mid \mathtt{err} \\
(\textit{Exp}) & \mathtt{E} & ::= a \mid x \mid i\ (\textit{integers}) \mid s\ (\textit{strings}) \mid op(\vec{\mathtt{E}}) \\
 & & \mid \{\vec{\mathtt{E}}\} \mid \mathtt{in}(\mathtt{E}, \mathtt{E}) \mid \mathtt{ins}(\mathtt{E}, \mathtt{E}) \mid \mathtt{del}(\mathtt{E}, \mathtt{E}) \\
 & & \mid \mathtt{form}\ f(\vec{s}\,\vec{a}{=}\vec{\mathtt{v}})\ \mathtt{with}\ \mathtt{E} \mid \mathtt{getCk}() \\
 & & \mid \psi \mid \mathtt{pack}() \mid \iota\ (\textit{tokens}) \mid \mathtt{newTk}() \\
(\textit{Value}) & \mathtt{v}, \mathtt{w} & ::= i \mid s \mid \{\vec{\mathtt{v}}\} \mid \phi \mid \psi \mid \iota \\
(\text{BASS } \textit{Type}) & \tau & ::= \mathtt{int} \mid \mathtt{str}
\end{array}
$$

Figure 9: MOSS syntax

server through web interactions, a client may also forge a form on its own (*e.g.,* $\mathtt{Form}_\mathtt{x}$ and $\mathtt{Form}_\mathtt{y}$). This reflects real-world exploits where a client makes up an HTTP request without going through the intended application logic. In addition, a malicious client B may trick an innocent client A by inserting a form from B's history (part of which may be forged) into A's history (*e.g.,* $\mathtt{Form}_\mathtt{z}$). In the real-world counterpart, an attacker may send crafted links to a victim in an email; sometimes, these links are image links that may be automatically loaded by a browser.

Whereas the virtual environment in Figure 4 provides an ideal view for the programmer, the model in Figure 8 reflects real-world web application scenarios, where server programs interact with and maintain data for multiple clients, who are no longer always well-behaved. To implement secure web applications in this model, the server programs need special primitives for manipulating security concepts and constructs. For example, special tokens (*e.g.,* SID) can be used to identify users and logical transactions, and those tokens can be embedded in the cookies and forms on the clients.

We present the details of this model as a language MOSS. The syntax is given in Figure 9. Vector notations (*e.g.,* $\vec{\Phi}$) are sometimes used when referring to a list of items (*e.g.,* $[\Phi_1 \ldots \Phi_n]$). The symbols for certain language constructs are borrowed from BASS to show correspondence of the underlying concepts, but minor modifications are applied to avoid confusion. In particular, we use a different font for reused English letters (W, C, E and v), and add a dot on top for reused Greek letters ($\dot{\Sigma}$, $\dot{\kappa}$, and $\dot{\sigma}$).

A world W in this language consists of three elements. The first is a global environment $\dot{\Sigma}$ on the server. The second is a closure $\dot{\kappa}$ currently active on the server. The third is a list of browsing histories $\vec{\Phi}$, each belonging to a different client.

The global environment $\dot{\Sigma}$ contains both data and code. The data part is essentially a list of session environments ς organized by some tokens ι for identifying sessions (*i.e.,* SIDs). Every session environment ς captures the values of the volatile variables (*e.g.,* a, a', a_1) for a session. The code part is simply a collection of functions, with every function consisting of a name f, a list of input parameters $\vec{a}$, and a body command C. Here, the symbols [] are used as parentheses of the meta language, rather than as part of MOSS.

The closure $\dot{\kappa}$ is usually a tuple consisting of a client identifier i (indicating that the current computation is for the ith client; this corresponds to the transient but dedicated connection established between the client and the server for an HTTP request-response), a value v as the content of the cookie (transferred from the client to the server in a request and passed back to the client in a response), a local environment $\dot{\sigma}$, and a command C. Sometimes a pair of a cookie value and a local environment also makes a closure; such a closure cannot be executed, but the information therein may be propagated to other closures. The local environment $\dot{\sigma}$ collects the values of non-volatile variables (*e.g.,* x, x', x_1). We assume that $\dot{\sigma}$ contains a special variable x_{sid} dedicated for storing the SID of the underlying session. Alternatively, one may wish to store the SID in the cookie. Since the cookie is shared by all forms on a client, that would prevent the same client from accessing multiple sessions simultaneously. Therefore, we choose to store the SID in $\dot{\sigma}$.

The clients vector $\vec{\Phi}$ deserves attention. In BASS, the single client is idealized as a list of closures. In MOSS, a world involves multiple clients. Each client Φ consists of a list of forms $\vec{\phi}$ and a cookie with the value v. A form ϕ is made up of input parameters $\vec{a}$ and their textual explanations $\vec{s}$ and default values $\vec{\mathtt{v}}$ (*i.e.,* input fields in a web form), a target f (*i.e.,* the web link behind the "submit" button), and some sealed client-side state ψ. Note that ψ is not meant to be modified by the client. This restriction will be enforced in the operational semantics. In the syntax, we use the notation $\langle \ldots \rangle$ to indicate that it is a "package" that cannot be taken apart by the client. In a real language, such sealed state can be signed by the server to prevent client modifications. We also point out that the forms in a client may belong to different sessions, because the client may initiate different sessions in different browser windows. Although reserving a dedicated variable x_{sid} in the local environments for session identification, MOSS does not automatically maintain it (or any other non-volatile variables) across web interactions. As in the case of real web programming, the programmer may maintain the relevant information in the sealed state.

Commands C involve the common ones and some new primitives adapted from real-world entities. For distinction from BASS, we use a different syntax for the common commands of skip (sk), assignment ($a \leftarrow E$), sequential composition (C ;; C), conditional (cond(E, C, C)), and while-loop (loop(E, C)). $f(\vec{\mathtt{E}})$ is a function call to f with the arguments $\vec{\mathtt{E}}$. For web input, input(E) is used to send a form computed from E to the client. setCk(E) sets a new value for the cookie in the current closure. unpack(E) unpacks the sealed state from E. verif($\vec{\mathtt{E}}, \vec{\tau}$) validates $\vec{\mathtt{E}}$ against BASS types $\vec{\tau}$. Finally, err exits the program upon an error; we use this abrupt program termination for simplicity, although real web applications typically perform more sophisticated error handling (*e.g.,* giving error messages, redirecting to other pages).

MOSS reuses the variables a and x, integers i, strings s, and abstract operations $op(\vec{\mathtt{E}})$ of BASS for ease of exposition. It introduces some set expressions: $\{\vec{\mathtt{E}}\}$ is a set with elements $\vec{\mathtt{E}}$, $\mathtt{in}(\mathtt{E}, \mathtt{E}')$ checks if E belongs to the set $\mathtt{E}'$, $\mathtt{ins}(\mathtt{E}, \mathtt{E}')$ inserts E into the set $\mathtt{E}'$, and $\mathtt{del}(\mathtt{E}, \mathtt{E}')$ removes E from the set $\mathtt{E}'$. form constructs a form by putting together some input parameters, a target function, and an expression carrying the sealed state. getCk() obtains the cookie value. ψ is a sealed state, and it is typically created using pack(). Finally, tokens ι are unique and unforgeable entities created using newTk(); newTk() will never create two identical tokens, and the tokens created cannot be guessed or forged by clients. The exact form of tokens is abstract. We have mentioned the use of tokens as SIDs. Later in Section 5, we will also use tokens to identify clients, enforce control flows, and prevent request forgeries.

4.2 Operational Semantics

A big-step semantics of the expressions ($\varsigma; \mathtt{v}_{ck}; \dot{\sigma} \vdash \mathtt{E} \Downarrow \mathtt{v}$) is given in the TR. The evaluation is carried out with respect to a session environment ς, a cookie value $\mathtt{v}_{ck}$, and a local environment

$\dot{\sigma}$. The definition is straightforward, noting that values for volatile and nonvolatile variables are pulled from ς and $\dot{\sigma}$, respectively.

Selected rules of the remainder of the operational semantics is given in Figure 10. Similar to BASS, the key concept is a "world step" relation $\mathtt{W} \rightsquigarrow \mathtt{W}'$. A multi-step relation $\mathtt{W} \rightsquigarrow^* \mathtt{W}'$ is defined as the reflexive and transitive closure of the world step relation.

Most of the server-side computation is captured in Rule (12), which delegates the task to a single-thread step relation $(\dot{\Sigma}, \dot{\kappa}) \rightsquigarrow (\dot{\Sigma}', \dot{\kappa}')$. Upon an $\mathtt{input(E)}$ command (Rule (13)), the expression $\mathtt{E}$ is evaluated to a form ϕ, and the result is transferred to the client. Note that the special variable x_{sid} is used to obtain the SID ι, and ι is used to locate the session environment ($\dot{\Sigma}(\iota)$ refers to the session environment identified by ι in $\dot{\Sigma}$). In addition, the client identifier i is used to locate the client history Φ_i. Φ_i may contain a different cookie value $\mathtt{v}$, because the server may have updated the cookie to $\mathtt{v}_{ck}$ in the current closure $\dot{\kappa}$. We update the client history from Φ_i to Φ_i' using the new cookie $\mathtt{v}_{ck}$ and form ϕ. In the result of the step, the clients vector is updated by removing the old Φ_i and inserting the updated Φ_i'. Any command $\mathtt{C}$ following $\mathtt{input(E)}$ is discarded. This reflects the case that the server execution stops after sending a response to the client, and resumes after receiving the next request; the control flow is connected by a link embedded in the web form $\mathtt{E}$, not by sequencing with another command $\mathtt{C}$. This Rule (13) is the only one that describes the execution upon an $\mathtt{input(E)}$ command. Therefore, the execution of a MOSS program gets stuck if $\mathtt{E}$ is not a form, if ι is not recognized by $\dot{\Sigma}$, or if x_{sid} does not contain a valid token at all. Similar observations apply to all other rules.

The remainder rules of the world step relation concern things initiated by a client. An ideal case is that a client fills in a form with values. However, in the real-world handling of a web form, the target function name and input fields are subject to malicious modifications, because they are directly embedded behind the "submit" button for the browser to recognize and produce an HTTP request. Such a scenario is captured in Rule (14), where a client Φ_i forges a new form ϕ' (the ideal form filling is a special case of this). Here, the client may make up arbitrary things on the function name and input fields, but not the sealed state ψ (*e.g.,* the client cannot forge the server's signature); ψ must come from some form ϕ in the history of Φ_i. In addition, the cookie value may also be modified by the client from $\mathtt{v}_{ck}$ to $\mathtt{v}'$. As a result of the step, the new client Φ_i' includes the forged form ϕ' and cookie $\mathtt{v}'$, and the clients vector $\vec{\Phi}$ is updated accordingly. The server side remains unchanged.

In Rule (15), the ith client picks a form from the history and submits it. On the server side, the function f must be recognized by the global environment, and the function must expect the proper number of arguments (a_{srl} is a special argument for the sealed state). In the result of the step, a new closure is composed for execution, where the cookie is obtained from the client, and the function f is applied to the sealed state and the input values. No input validation is done here. A program which does not perform proper input validation on its own may go wrong in later computation.

Sometimes an attacker tricks a victim into sending a request composed by the attacker. In Rule (16), the attacker client i picks an arbitrary form ϕ from Φ_i, and inserts it into the victim client j's history. Now that the form is injected, a later request following Rule (15) may submit it to the server, thus completing the attack.

An existing client may initiate a new session using Rule (17). On the server, a fresh token ι is created as the SID, an empty session environment is added into the global environment $\dot{\Sigma}$, and a new closure $\dot{\kappa}$ is composed. In the closure $\dot{\kappa}$, the local environment is initialized so that x_{sid} is bound to ι, and the code starts from f_{start}, which is a special function reserved as the program entry point. The session and local environments do not contain entries

$\boxed{\mathtt{W} \rightsquigarrow \mathtt{W}'}$

Server computation:

$$\frac{(\dot{\Sigma}, \dot{\kappa}) \rightsquigarrow (\dot{\Sigma}', \dot{\kappa}')}{(\dot{\Sigma}, \dot{\kappa}, \vec{\Phi}) \rightsquigarrow (\dot{\Sigma}', \dot{\kappa}', \vec{\Phi})} \tag{12}$$

Server response:

$$\frac{\begin{array}{l}\dot{\kappa} = (i, \mathtt{v}_{ck}, \dot{\sigma}, \mathtt{input(E)}) \text{ or } \dot{\kappa} = (i, \mathtt{v}_{ck}, \dot{\sigma}, \mathtt{input(E)} \mathbin{;;} \mathtt{C}) \\ \dot{\sigma}(x_{sid}) = \iota \qquad \dot{\Sigma}(\iota); \mathtt{v}_{ck}; \dot{\sigma} \vdash \mathtt{E} \Downarrow \phi \\ \Phi_i = (\mathtt{v}, \vec{\phi}) \qquad \Phi_i' = (\mathtt{v}_{ck}, [\phi] \cup \vec{\phi})\end{array}}{(\dot{\Sigma}, \dot{\kappa}, \vec{\Phi}) \rightsquigarrow (\dot{\Sigma}, (\mathtt{v}_{ck}, \dot{\sigma}), (\vec{\Phi} - [\Phi_i]) \cup [\Phi_i'])} \tag{13}$$

Client forging (form filling):

$$\frac{\begin{array}{l}\Phi_i \in \vec{\Phi} \qquad \Phi_i = (\mathtt{v}_{ck}, \vec{\phi}) \\ \phi \in \vec{\phi} \qquad \phi = \mathtt{form}\ f(\vec{s}\ \vec{a}{=}\vec{\mathtt{v}})\ \mathtt{with}\ \psi \\ \phi' = \mathtt{form}\ f'(\vec{s'}\ \vec{a'}{=}\vec{\mathtt{w}})\ \mathtt{with}\ \psi \qquad \Phi_i' = (\mathtt{v}', [\phi'] \cup \vec{\phi})\end{array}}{(\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}), \vec{\Phi}) \rightsquigarrow (\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}), (\vec{\Phi} - [\Phi_i]) \cup [\Phi_i'])} \tag{14}$$

Client request:

$$\frac{\begin{array}{l}\Phi_i \in \vec{\Phi} \qquad \Phi_i = (\mathtt{v}_{ck}, \vec{\phi}) \qquad \mathtt{form}\, f(\vec{s}\ \vec{a}{=}\vec{\mathtt{v}})\ \mathtt{with}\ \psi \in \vec{\phi} \\ (f = (a_{srl}, \vec{a}) \triangleright \mathtt{C}) \in \dot{\Sigma}\end{array}}{(\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}), \vec{\Phi}) \rightsquigarrow (\dot{\Sigma}, (i, \mathtt{v}_{ck}, \{\}, f(\psi, \vec{\mathtt{v}})), \vec{\Phi})} \tag{15}$$

Attacker tricking:

$$\frac{\begin{array}{l}\Phi_i \in \vec{\Phi} \qquad \Phi_i = (\mathtt{v}', \vec{\phi}) \qquad \phi \in \vec{\phi} \\ \Phi_j \in \vec{\Phi} \qquad \Phi_j = (\mathtt{v}_{ck}, \vec{\phi'}) \qquad \Phi_j' = (\mathtt{v}_{ck}, [\phi] \cup \vec{\phi'})\end{array}}{(\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}), \vec{\Phi}) \rightsquigarrow (\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}), (\vec{\Phi} - [\Phi_j]) \cup [\Phi_j'])} \tag{16}$$

New session:

$$\frac{\begin{array}{l}\Phi_i \in \vec{\Phi} \qquad \Phi_i = (\mathtt{v}_{ck}, \vec{\phi}) \qquad fresh(\iota) \\ \dot{\Sigma}' = \dot{\Sigma} \cup \{\iota : \{\}\} \qquad \dot{\kappa} = (i, \mathtt{v}_{ck}, \{x_{sid} = \iota\}, f_{start}())\end{array}}{(\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}), \vec{\Phi}) \rightsquigarrow (\dot{\Sigma}', \dot{\kappa}, \vec{\Phi})} \tag{17}$$

New client:

$$\frac{\begin{array}{ll}fresh(\iota) & \dot{\Sigma}' = \dot{\Sigma} \cup \{\iota : \{\}\} \qquad |\vec{\Phi}| = n \\ fresh(\iota') & \dot{\kappa} = (n+1, \iota', \{x_{sid} = \iota\}, f_{start}())\end{array}}{(\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}), \vec{\Phi}) \rightsquigarrow (\dot{\Sigma}', \dot{\kappa}, \vec{\Phi} \cup [(\bullet, \epsilon)])} \tag{18}$$

$\boxed{(\Sigma, \kappa) \rightsquigarrow (\Sigma', \kappa')}$ (selected cases only; see TR)

$$\frac{\begin{array}{l}(f = (\vec{a}) \triangleright \mathtt{C}) \in \dot{\Sigma} \qquad |\vec{\mathtt{E}}| = |\vec{a}| \\ \dot{\sigma}(x_{sid}) = \iota \qquad \dot{\Sigma}(\iota); \mathtt{v}_{ck}; \dot{\sigma} \vdash \vec{\mathtt{E}} \Downarrow \vec{\mathtt{v}}\end{array}}{(\dot{\Sigma}, (i, \mathtt{v}_{ck}, \dot{\sigma}, f(\vec{\mathtt{E}}))) \rightsquigarrow (\dot{\Sigma}, (i, \mathtt{v}_{ck}, \dot{\sigma}, \mathtt{C}[\vec{\mathtt{v}}/\vec{a}]))} \tag{19}$$

$$\frac{\dot{\sigma}(x_{sid}) = \iota \qquad \dot{\Sigma}(\iota); \mathtt{v}_{ck}; \dot{\sigma} \vdash \mathtt{E} \Downarrow \mathtt{v}}{(\dot{\Sigma}, (i, \mathtt{v}_{ck}, \dot{\sigma}, \mathtt{setCk(E)})) \rightsquigarrow (\dot{\Sigma}, (\mathtt{v}, \dot{\sigma}))} \tag{20}$$

$$\frac{\begin{array}{l}\dot{\sigma}(x_{sid}) = \iota \qquad \dot{\Sigma}(\iota); \mathtt{v}_{ck}; \dot{\sigma} \vdash \mathtt{E} \Downarrow \langle \vec{x} = \vec{\mathtt{v}} \rangle \\ update(\dot{\sigma}, \vec{x}, \vec{\mathtt{v}}) = \dot{\sigma}'\end{array}}{(\dot{\Sigma}, (i, \mathtt{v}_{ck}, \dot{\sigma}, \mathtt{unpack(E)})) \rightsquigarrow (\dot{\Sigma}, (\mathtt{v}_{ck}, \dot{\sigma}'))} \tag{21}$$

$$\frac{\begin{array}{l}\dot{\sigma}(x_{sid}) = \iota \qquad \dot{\Sigma}(\iota); \mathtt{v}_{ck}; \dot{\sigma} \vdash \vec{\mathtt{E}} \Downarrow \vec{\mathtt{v}} \\ \forall k.\ (\exists i.\mathtt{v}_k = i \text{ and } \tau_k = \mathtt{int}) \text{ or } (\exists s.\mathtt{v}_k = s \text{ and } \tau_k = \mathtt{str})\end{array}}{(\dot{\Sigma}, (i, \mathtt{v}_{ck}, \dot{\sigma}, \mathtt{verif}(\vec{\mathtt{E}}, \vec{\tau}))) \rightsquigarrow (\dot{\Sigma}, (\mathtt{v}_{ck}, \dot{\sigma}))} \tag{22}$$

Figure 10: MOSS world execution

for programmer-defined variables yet. Instead, those variables will be declared implicitly upon their first use (articulated in the TR).

The last client-initiated step, Rule (18), applies when a new client joins. Similar to Rule (17), a new session with SID ι is created. In addition, a fresh token ι' is created as an ID for uniquely identifying the client (we will refer to this as a CID). ι' is stored in the cookie of the new closure $\dot{\kappa}$, and will be propagated to the client upon a web interaction using Rule (13). As a result of the step, the global environment is updated, the new closure is activated for execution, and the clients vector is updated to include the new client, which has the transient client identifier $n{+}1$, an empty cookie indicated by the special value $\bullet$, and an empty history vector ϵ.

The bottom of Figure 10 shows a few cases of the single-thread step relation $(\dot{\Sigma}, \dot{\kappa}) \rightsquigarrow (\dot{\Sigma}', \dot{\kappa}')$. Rule (19) carries out a function call using capture-avoiding substitution, after checking the number of and evaluating the arguments. Rule (20) updates the cookie of the current closure. Rule (21) evaluates E to a sealed state, and updates the local environment $\dot{\sigma}$ accordingly. Rule (22) validates $\vec{\mathtt{E}}$ against BASS types $\vec{\tau}$. There is no rule for the case where the validation fails—the execution gets stuck upon a failed validation.

4.3 Example Use: Patterns of Exploits

MOSS is designed to be flexible, and program execution could result in stucks. Some causes of stucks include unrecognized SIDs or function names, unmatched number of function parameters, illegal usage of values (*e.g.,* unpacking an integer), and failed input validations. Indeed, MOSS is meant to reflect real-world scenarios, rather than to confine web program behaviors using a restricted semantics or a type system. Similar to other languages, MOSS can be used to write both secure programs and vulnerable ones.

Besides serving as the target of our BASS translation, MOSS is of independent value to understanding web programming, because it helps to articulate the essence of common exploits and attacks. We demonstrate this by encoding two exploit patterns, which cover all the example attacks discussed in Section 2.3.

The first is an *attacker-victim* pattern, where an attacker takes advantage by tricking a victim into submitting an unintended request, which in turn triggers a vulnerability in a server program. We assume the application stores the SID in the client cookie.

$$
\begin{array}{ll}
& (\dot{\Sigma}, (\mathtt{v},\dot{\sigma}), [(\iota_A, \vec{\phi_A}), (\iota_B, \vec{\phi_B})]) \\
\rightsquigarrow & (\dot{\Sigma}, (\mathtt{v},\dot{\sigma}), [(\iota_A, \vec{\phi_A}), (\iota_B, \vec{\phi_B} \cup [\phi])]) & \text{(forge by Rule (14))} \\
\rightsquigarrow & (\dot{\Sigma}, (\mathtt{v},\dot{\sigma}), [(\iota_A, \vec{\phi_A} \cup [\phi]), (\iota_B, \vec{\phi_B} \cup [\phi])]) & \text{(trick by Rule (16))} \\
\rightsquigarrow & (\dot{\Sigma}, \dot{\kappa}, [(\iota_A, \vec{\phi_A} \cup [\phi]), (\iota_B, \vec{\phi_B} \cup [\phi])]) & \text{(request by Rule (15))}
\end{array}
$$

Here the attacker B forges a form ϕ in the first step, and injects it into A (*e.g.,* by emailing a link) in the second step. When A submits the form (*e.g.,* by following the link) in the third step, the server program composes a closure $\dot{\kappa}$ to process the request. Whereas a careful server program may inspect ϕ to identify injected requests (Section 5), a vulnerable one might simply identify the session based on the SID ι_A transferred from A's cookie. Therefore, the request is processed as if it were intended by A.

This pattern captures different attacks when ϕ and $\dot{\kappa}$ are instantiated with different entities based on different program vulnerabilities. For instance, suppose a banking program has a CSRF vulnerability—it accepts payment requests solely based on the SID stored in the cookie. The attacker B may instantiate the above pattern with $\phi = \mathtt{form}\ f_{pay}(\text{“}payee\text{”}\ p{=}\text{“}B\text{”}, \text{“}amnt\text{”}\ a{=}100)\ \mathtt{with}\ \langle\rangle$. Upon receiving the request, the server program would compose a closure to set up a payment from A. Similarly, a program with an XSS vulnerability would propagate a crafted string (*e.g.,* cookie-stealing JavaScript code) from B's forged form to an HTML page displayed in A's browser, and a program with a session fixation vulnerability would set A up to use an SID fixed by B.

The second is a *malicious client* pattern, where a malicious client crafts a request to exploit server program vulnerabilities.

$$
\begin{array}{ll}
& (\dot{\Sigma}, (\mathtt{v},\dot{\sigma}), [(\iota_A, \vec{\phi_A})]) \\
\rightsquigarrow & (\dot{\Sigma}, (\mathtt{v},\dot{\sigma}), [(\iota_A, \vec{\phi_A} \cup [\phi])]) & \text{(forge by Rule (14))} \\
\rightsquigarrow & (\dot{\Sigma}, \dot{\kappa}, [(\iota_A, \vec{\phi_A} \cup [\phi])]) & \text{(request by Rule (15))}
\end{array}
$$

Since a malicious client A may use arbitrary target function names, parameter names and values, and sealed states (*e.g.,* ones obtained from other forms in the history) to compose the request ϕ, a server program which does not carefully code against these possibilities would be subject to attacks including session poisoning, malformed input, and work-flow circumvention.

5. TRANSLATION

By comparing BASS (our proposed ideal programming model) and MOSS (the actual programming model in use today), the advantage of BASS is clear—none of the above attacks can happen to a BASS program, simply because BASS has a “well-behaved” semantics. We now discuss how this semantics can be implemented. Specifically, we will point out how the two patterns of attacks are prevented. Although focusing on a particular translation, we note that other pertinent secure coding practices could be applied instead. The bottom line is, with the BASS abstractions implemented following secure coding practices once and for all, programmers enjoy an ideal model that is free from common exploits.

In particular, BASS has an ideal model where the control flow is not disrupted by web input, there is only a single session, and the client is well-behaved. These no longer hold in the MOSS model of real-world scenarios. To connect the two models, the translation takes care of program splitting, form making, session management, state maintenance, forgery prevention, and input validation. Recall that a client in MOSS consists of a cookie and a list of forms. In our translation, we maintain the invariant that the cookie stores the CID of the client, and the forms are of the following shape:

$$
\begin{array}{l}
\mathtt{form}\ f(\vec{s}\ \vec{a}{=}\vec{\mathtt{w}}) \\
\quad \mathtt{with}\ \langle x_{cid}{=}\iota_{cid}, x_{sid}{=}\iota_{sid}, x_{tok}{=}\iota_{tok}, x_{fun}{=}s_f, \vec{x}{=}\vec{\mathtt{v}} \rangle
\end{array}
$$

A key to the translation, this is in essence an encoding of BASS client closures in MOSS forms. Besides the target function f and input parameters $\vec{a}$, we also encode information about the local environment (nonvolatile variables $\vec{x}{=}\vec{\mathtt{v}}$). In addition, some special variables are used for session management and forgery prevention.

The first special variable, x_{cid}, stores the CID that the server assigned to the client in Rule (18). In a translated program, before sending a response to the client, we obtain the CID from the cookie in the current closure, and store it in variable x_{cid}. On the other hand, when processing a request from a client, we inspect x_{cid} and check it against the client cookie. This ensures that the request indeed comes from the client, because: (1) a sealed state cannot be modified by a client; (2) the cookie of a client cannot be updated by others; (3) although a client may modify her cookie, she cannot change it to the value of another client's unforgeable CID. These together prevents attacker-crafted forms, inserted into a victim's history using Rule (16), from being accepted by the server. As a result, the attacker-victim pattern will not succeed.

The variable x_{sid} stores the SID. The variable x_{tok} stores a token for history control. The validity of this token will be checked when the server program processes the request. For single-use forms, the token will be invalidated after the first use, thus preventing future resubmissions. In addition, the token can also be invalidated if we choose to expire all existing forms (*cf.* a BASS `clear` operation).

Note that values in x_{sid} and x_{tok} only need to be unique; they do not have to be unforgeable on their own, because they are stored in the sealed state and thus cannot be forged. Although we reuse the same token construct, a realization may relax this.

Besides the three tokens, there is a special variable x_{fun} that stores the target function name. This is to prevent certain request forgeries where the client modifies the target function of the form. Upon a request, x_{fun} will be inspected and the request will be processed only if it matches the actual target function. This effectively binds the target function name and sealed state together in a form. Together with some type-based input validation code generated by the translation, this prevents attacks of the malicious client pattern. The exact formal translation details are given in the TR.

Correctness and prototyping We summarize some key arguments on the preservation of all the properties discussed in Section 3.3, which can be easily established by inspecting the form encoding. Since the sealed state is protected, the only form component subject to client modification is the input parameters. Modifications on the target function name will not succeed, because the function name is also stored in the sealed state. As a result, the control flow of a BASS program will be enforced, since clients cannot forge target links—they can only follow the server-provided forms to access the server programs. In addition, an attacker cannot masquerade as another client—although an attacker may modify the cookie, she cannot set it to the unforgeable CID of another client. More interestingly, CSRF, first-order XSS and session fixation, which require an attacker to inject a form into a client (Rule (16)), are prevented because the injected form will have a different CID than that stored in the client's cookie. Finally, input validation, session state recovery, and history control are all properly carried out.

We have implemented a prototype compiler of BASS in OCaml. The compiler takes well-formed BASS programs and produces secure Perl code. SSL is used for web interactions. All constructs of MOSS and the translation have clear counterparts in the prototype, demonstrating that our modeling and translation are faithful to real-world scenarios. The compiler consists of 1456 lines of OCaml code (about 49KB), and uses a runtime library of 342 lines of Perl code (about 7KB). The executable in FreeBSD ELF format is about 346KB. The compiler translated the banking example of Figure 3 into 99 lines of Perl code. Focusing on the formal BASS language, the scale of this prototype is small. We plan to experiment with a larger scale prototype in the future, where the ideas of this paper will be implemented for an existing language as a library.

6. RELATED AND FUTURE WORK

Declarative web programming MAWL [2, 3] and its descendants (<bigwig> [5], JWIG [6]) use domain-specific constructs to program web applications. They view web applications as form-based services, and provide abstractions on some key aspects such as web input and state management. These abstractions hide implementation details (e.g., input validation, embedding of continuation and state in URL), thus preventing certain programming errors. Graunke *et al.* [10] propose the design and implementation of an I/O construct for web interactions. This construct helps to program web applications in a more traditional model of interactions, and avoids the manual saving and restoring of control state.

Although similar in spirit to BASS on these aspects, the above work does not provide a formal semantics with the same security guarantees. However, security should not be overlooked for declarative web programming—now that the details of web interactions are hidden by new abstractions, programmers can no longer carry out the secure coding practices by themselves. As a result, a naïve application of new abstractions could suffer from security vulnerabilities such as CSRF. It is thus crucial that the proposed abstractions and their implementation provide related security guarantees.

On expressiveness, MAWL and descendants enforce a strict control flow where every form is, in the BASS terminology, single-use. For example, users will be redirected to the beginning of a session if they hit the back button. In contrast, BASS leaves the design decision to the programmer, rather than disabling "whimsical navigation" [12] altogether. This flexibility is important [11, 12, 21].

We emphasize that the goal of BASS is to facilitate secure web programming with abstractions more suitable for the domain. Besides having declarative support on web interactions, single-/multi-use forms, state declarations, and history control, it is important that the features are all modelled within an original and self-contained semantic specification. BASS has an intuitive and formal programming model with an explicit notion of a client, and its meta properties are articulated. This allows programmers to fully grasp how BASS programs behave. The common task of following secure coding practices, which is orthogonal to the specific application logic, is carried out by a BASS implementation once and for all.

There has also been work developing domain-specific languages or frameworks for web programming as libraries of existing type-safe languages. Examples include the Curry-based server-side scripting language by Hanus [13], Haskell-based WASH/CGI by Thiemann [21], and Smalltalk-based Seaside by Ducasse *et al.* [8]. These provide useful abstractions in the form of libraries to handle some common aspects of web programming, such as structured HTML generation, session management, and client-server communication, but do not provide the formal security guarantees of BASS. There is no stand-alone formal semantics for the new abstractions, although in principle the behaviors could be inferred from the implementations and the semantics of the host languages. Finally, they are tied to the host languages, thus the ideas are not easily applicable to other languages. In contrast, BASS is translated into a flexible model underlying web programming in general. The rigorous BASS and MOSS allow us to formally derive security guarantees.

Models of web programs Besides building new abstractions, it is also important to understand what the existing web programming model is and how to write secure programs within it. Although there are many security recommendations and coding practices available, web programming has rarely been formally studied from the language principles. An exception is the work by Graunke *et al.* [12], which models web interactions in presence of whimsical navigation behaviors of well-intended (non-malicious) users, identifies two classes of errors (form field mismatch, client-server state mismatch), and proposes to catch the errors with a static type system (typed web forms) and dynamic checks (time-stamped states).

In comparison with our work, Graunke *et al.* do not address the wider security questions, where the server interacts with multiple clients, some of whom may be malicious. Specifically, they do not have security-related primitives in their server-side computation, they model the presence of a single client, and their transitions lack the counterparts of the forging (Rule (14)) and tricking (Rule (16)) behaviors of MOSS. In contrast, we more accurately model web interactions and server computations. This allows us to encode common exploit patterns (Section 4.3) for formal reasoning. Instead of using a type system or dynamic checks to close up the two classes of errors, we use principled but flexible abstractions in BASS for web programming and present their realization in MOSS. This hides some common details of secure web programming and precludes some different classes of errors. Although we do not propose techniques preventing errors associated with whimsical navigations, the related error scenarios can be rigorously illustrated in

BASS and MOSS; therefore, programmers are fully aware of the issues, and the techniques of Graunke *et al.* can be applied.

Future work We believe that web programming will benefit significantly from the use of domain-specific abstractions, and much can be done in the area. For starters, the `form` constructs of BASS can be generalized to take an HTML page as a parameter. Sometimes an HTML page may contain multiple forms, each with a separate submit button linking to a different target. A more interesting aspect is to include client-side scripting. This is orthogonal to the focus of server-side scripting in this paper, and previous work on the formal aspects of JavaScript [22, 1, 25] and web application development frameworks [9, 15] provide some good starting points.

BASS provides some security guarantees (Section 3.3) using a few new abstractions. These abstractions are not meant to be "complete," and there are other desirable properties uncovered. It is useful to explore abstractions for other areas, such as dynamic HTML generation [13, 6, 17, 21], privilege management, and dynamic SQL construction [20, 23]. More details are given in the TR.

Designed for web programming in general, BASS addresses only common security aspects, rather than issues on the specifics of an application. For example, directory traversal [7] (accessing the parent directory using the path "`..\`") is not prevented by common type-based input validation, and programmers must perform additional filtering. Application-specific security analysis will still be necessary. However, with the new abstractions closing up some common vulnerabilities and clarifying the control flow, such analysis should be easier. In general, the new abstractions should help the analysis, reasoning, and testing of web programs, because they provide an ideal model amenable to established language techniques.

In this paper, MOSS serves mainly as the target of the BASS translation. However, it is also of independent value to web programming studies. In MOSS, a client may take control only if the server contains no active closure. This is similar in spirit to non-preemptive multi-threading. A preemptive version of MOSS is interesting future work. In addition, type systems and program logics for confining and reasoning about MOSS programs are useful.

7. CONCLUSION

Web applications reflect a different computation model than conventional software, and the security issues therein deserve careful study from both language principles and practical implementations. This paper serves as a useful step towards building a formal foundation for secure server-side scripting. In particular, we propose two self-contained formalizations on the topic, using familiar language concepts such as continuations, threads, and small-step semantics.

The first is a formal language BASS for server-side scripting. BASS provides an ideal programming model where the server interacts with a single client, using some dedicated constructs to obtain web input and manipulate client history. The meta properties and formal guarantees of BASS allow programmers to focus on the application logic without being distracted by common implementation details, thus improving productivity and security.

The second is a formal model MOSS characterizing realistic web programming concepts. As is the case of existing server-side scripting languages, MOSS can be used to write both secure programs and vulnerable ones. We present a translation from BASS to MOSS, demonstrating how the BASS abstractions and guarantees can be enforced using a few common primitives for manipulating security concepts. Our prototype shows much promise on the idea of better abstractions for secure server-side scripting, and we hope to experiment more on the topic with real-world web programming languages and frameworks.

8. REFERENCES

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. 19th European Conference on Object-Oriented Programming*, pages 429–452, July 2005.

[2] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proc. 1997 Conference on Domain-Specific Languages*, 1997.

[3] D. L. Atkins, T. Ball, G. Bruns, and K. Cox. MAWL: A domain-specific language for form-based services. *IEEE Trans. on Software Engineering*, 25(3):334–346, 1999.

[4] R. Auger. The Cross-Site Request Forgery FAQ. `http://www.cgisecurity.com/articles`, 2007.

[5] C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *ACM Trans. on Internet Technology*, 2(2):79–114, 2002.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Trans. on Programming Languages and Systems*, 25(6):814–875, Nov. 2003.

[7] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. `http://cve.mitre.org/docs/vuln-trends`, 2007.

[8] S. Ducasse, A. Lienhard, and L. Renggli. Seaside — a multiple control flow web application framework. In *Proc. 12th International Smalltalk Conference*, pages 231–257, Sept. 2004.

[9] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, May 2007.

[10] P. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. In *Proc. 16th International Conference on Automated Software Engineering*, pages 211–222, Nov. 2001.

[11] P. Graunke and S. Krishnamurthi. Advanced control flows for flexible graphical user interfaces. In *Proc. 2002 International Conference on Software Engineering*, pages 277–296, 2002.

[12] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Modeling web interactions. In *Proc. 2003 European Symposium on Programming*, pages 122–136, 2003.

[13] M. Hanus. High-level server side web scripting in Curry. In *Proc. 3rd International Symposium on Practical Aspects of Delcarative Languages*, pages 76–92, 2001.

[14] J. Kolšek. Session fixation vulnerability in web-based applications. `http://www.acrossecurity.com/papers.htm`, 2002.

[15] B. Livshits and Ú. Erlingsson. Using web application construction frameworks to protect against code injection attacks. In *Proc. Programming Languages and Analysis for Security*, June 2007.

[16] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying XSS vulnerabilities in web applications. In *Proc. 6th International Workshop on Web Site Evolution*, pages 71–80, 2004.

[17] K. Nørmark. Web programming in Scheme with LAML. *Journal of Functional Programming*, 15(1):53–65, 2005.

[18] G. Ollmann. Second-order code injection attacks. `http://www.nextgenss.com/papers`, 2004.

[19] OWASP Foundation. The ten most critical web application security vulnerabilities. `http://www.owasp.org`, 2007.

[20] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proc. 33rd Symposium on Principles of Programming Languages*, pages 372–382, Jan. 2006.

[21] P. Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. on Internet Technology*, 5(1):1–46, Feb. 2005.

[22] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. 2005 European Symposium on Programming*, pages 408–422, Apr. 2005.

[23] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. 2007 Conference on Programming Language Design and Implementation*, pages 32–41, June 2007.

[24] D. Yu, A. Chander, H. Inamura, and I. Serikov. Better abstractions for secure server-side scripting. DCL-TR-2007-0035, July 2007.

[25] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. 34th Symposium on Principles of Programming Languages*, pages 237–249, Jan. 2007.