Types and Access Controls for Cross-Domain Security in Flash

Aseem Rastogi[1*], Avik Chaudhuri[2], and Rob Johnson[3]

[1] University of Maryland, College Park and Adobe Systems Inc.
aseem@cs.umd.edu
[2] Adobe Systems Inc.
avik.chaudhuri@adobe.com
[3] Stony Brook University
rob@cs.stonybrook.edu

Abstract. The ubiquitous Flash platform enables programmers to build sophisticated web application "mash-ups" that combine Flash executables loaded from multiple trust domains with complex, asymmetric trust relationships. Flash provides APIs and run-time checks to help programmers declare and enforce trust relationships between different domains, but there is currently no formal security model for Flash.
This paper presents the first formal security model for the Flash platform. Our formal model instantly reveals that the run-time checks performed by the Flash runtime are not sufficient to enforce data integrity – we present simple example programs that are vulnerable to attacks. We then develop a static type system for Flash programs that lets programmers specify fine-grained trust relationships, and we show that, combined with the run-time checks already performed by the Flash runtime, well-typed programs cannot violate data integrity at run-time.

Keywords: Flash Security, Security Model, Dynamic Access Control, Type System

1 Introduction

Adobe Flash is a widely-used multimedia platform for building interactive Internet applications. Flash is frequently used for video and audio applications, advertisements, and games, making it nearly ubiquitous on the web. Flash applications can load additional Flash applications from the same or other web domains, allowing developers to create mashups that combine functionality provided by several different domains.

Flash implements the "same origin policy" that is also used in JavaScript, but Flash applications can explicitly grant access to some trusted domains, enabling scripts running in the context of one domain to access data and functions of scripts loaded in another domain. This enables Flash components from different domains to communicate with each other in a controlled manner.

[*] Work performed while the author was at Stony Brook University

However, there is currently no formal specification of the Flash security model. Without a formal specification, the guarantees of the model are unclear. Programmers cannot reason about the security of their applications – e.g. they cannot easily verify that a security-critical function in their application cannot be called with an argument coming from an untrusted application.

To that end, we make the following contributions in this paper:

- We present a formal specification of the Flash security model. The specification includes carefully modeled semantics for the dynamic access control checks and the APIs for dynamically loading other Flash applications.
- We show that the run-time checks performed by the Flash runtime are not by themselves sufficient to guarantee data integrity for Flash applications. We formalize the data integrity property and give simple examples that violate the property under the current semantics.
- We present a security extension to the Flash static type system that enables programmers to express and enforce data integrity invariants. We give a soundness theorem of our type system and prove that the runtime access control checks are sufficient to maintain the data integrity invariants of the well-typed Flash applications.
- We strengthen our threat model by allowing an attacker to by-pass the static security checks and show how our type system, in conjunction with the run-time checks, still guarantees that data integrity is maintained for a well-typed victim application executing in the same environment as the attacker's application.

We mainly focus on the data integrity property in Flash applications. This helps us prevent Cross Site Scripting attacks (and other injection attacks [1]) which are instances of data integrity violations (Section 3), and more importantly, one of the most common types of real world attacks on Flash applications [2].

Our type system is based on the crucial observation that in the Flash security model, if an application is compromised, only itself and other applications from the domains it trusts, are to be blamed. Hence, it is possible to impose a static discipline on the trusted programs, that combined with the existing run-time checks, can prevent data integrity violations.

The type system adds labels $\mathcal{D}$ to the types in the language, where each label is a set of some domains. The interpretation of the labels is that, a type with label $\mathcal{D}$ must admit only those values that *come from* one of the domains in $\mathcal{D}$. We present a static consistency relation over types, that enforces the following safety property for all the flows in the program: if a term with label $\mathcal{D}_1$ flows to a context that expects a term with label $\mathcal{D}_2$, then $\mathcal{D}_1 \subseteq \mathcal{D}_2$ must hold.

Interestingly, we can afford to allow the attacker to by-pass the safety checks mentioned above. We prove that as long as the victim makes the *worst* assumptions about the attacker's code – a value coming from the attacker's code could have originated anywhere – while type checking its own code, the data integrity property is maintained at run-time.

We present an overview of the Flash security model in Section 2. We give our threat model and some example data integrity violation attacks in Section 3. We use these examples throughout the paper to exemplify the runtime semantics (Section 4) and the type system (Section 5). We state the soundness theorem of our type system in Section 5.1. We discuss related work in Section 6 and finally conclude in Section 7.

2 Overview

Adobe Flash platform [3] is a complete application development stack. The programmer writes code in the ActionScript language and compiles it to a bytecode representation, called the swf. The swfs are typically deployed over HTTP and are included by their URL inside the HTML pages. When the HTML pages are rendered by the web browser, the Flash Player plugin in the browser loads the swfs from their URL and executes them.

Flash platform enables programmers to design and develop rich functionality applications. In particular, the platform provides APIs to load and execute other swfs, and possibly access their data and functions. A detailed account of the API can be found in [1].

Domain Based Sandboxing The Flash Player partitions the execution environment into multiple security contexts that are defined by the source domains of the comprising swfs. For every source domain in the environment, there exists a security context, in which all the swfs loaded from that source domain execute.

The Flash Player always allows a swf to access data and functions of any other swf in the same security context. This is called the *same origin policy*, wherein the swfs loaded from the same domain can freely script into each other. However, by default, any attempt by a swf in one security context to access data and functions of a swf in another security context, generates a runtime security error.

Extension to the Same Origin Policy To handle the situations where cross-domain scripting is sometimes desirable, the Flash platform allows for an extension to the same origin policy. A swf may express trust on some domain(s) (other than its own source domain). The implication is that, any swf from the trusted domain(s) is allowed to freely script into the trusting swf.

However, it only opens up a one way communication channel, from the trusted domain swfs to the trusting swf. For the communication in the other direction, the receiving swf has to explicitly express trust itself.

Content Loading and Import Loading When a swf loads another swf, the loaded swf executes in the security context defined by its own source domain. This regular loading mechanism is called *Content Loading*. The Flash platform provides another loading mechanism, known as *Import Loading*, in which the loaded swf is executed in the same security context as that of the loader swf.

Thus, the effect is as if the loaded swf comes from the same domain as the loader swf.

However, the source domain of the loaded swf has to allow the loader domain to import load – by placing on its webserver a crossdomain policy file having an entry for the loader domain. Before import loading, the Flash Player checks that the webserver of the loaded swf has a crossdomain policy file and that this file contains an entry for the loader domain. On the other hand, the loader must also be careful before import loading a swf from another domain, since import loading grants the loaded swf all the privileges of the loader domain.

3 Threat Model and Examples

Threat Model We assume that the attacker owns some web domains, which are not trusted by the victim. Furthermore, it creates some swfs using the Flash platform tools and hosts them on its web domains. Thus, the attacker's swfs are capable of doing all that is allowed by the language. Moreover, we allow the attacker's programs to by-pass the static security checks of our type system. Since the victim's swfs and the attacker's swfs can load one another, the Flash runtime environment could contain the victim's swfs and the attacker's swfs together. Our goal is to maintain the data integrity property at run-time. We formalize the data integrity property in the next section.

Examples We now give some examples of possible data integrity violations in the Flash programs. We use these examples throughout the rest of the paper to show whether and how they are prevented by the runtime semantics and our type system.

For the purpose of the examples, the functions are written as `fun(x) { .. }`, where `x` is the function parameter. The object literals are written as $\{\texttt{f}_1 = \texttt{t}_1, \texttt{f}_2 = \texttt{t}_2, \ldots\}$, where $\texttt{f}_1, \texttt{f}_2, \ldots$ are the property names and $\texttt{t}_1, \texttt{t}_2, \ldots$ are the terms. The object properties are evaluated left to right. The function `load(i)` loads a swf `i`. We denote the attacker's swf and domain by `a` and `d` respectively, and the victim's swf and domain by `v` and `d'` respectively. A swf is modeled like an object literal and written as $[\![\texttt{f}_1 = \texttt{t}_1, \texttt{f}_2 = \texttt{t}_2, \ldots]\!]$.

As mentioned above, we assume that the victim does not trust the attacker. However, the attacker trusts the victim since it's in the attacker's interest to allow the victim to interact with it as much as possible to create the opportunities for exploiting the victim's swf (as we see in the Example 2 below).

Example 1. Attacker's swf directly scripts into the victim's swf:

```
v = [[s_fn = fun(x) { /* transfer $100 to account x */; return 0 }]]
a = [[v_swf = load(v), f = v_swf.s_fn(a_acc)]]
```

The attacker's swf loads the victim's swf and directly calls the sensitive function in that swf with an arbitrary argument. This attack is known as the Cross Domain Scripting attack [1], and is prevented by the run-time checks in the Flash Player.

Example 2. Victim's swf passes a sensitive function to the attacker's swf

```
v = ⟦s_fn = /* as before */, a_swf = load(a), f = a_swf.f'(s_fn)⟧
a = ⟦f' = fun(g) { g(a_acc) }⟧
```

The victim's swf loads the attacker's swf and ends up passing a sensitive function to the attacker. The attacker can then call that sensitive function with an arbitrary argument. This example is a more general form of the Cross Domain Scripting attack, but is not pointed out in the existing documentation on Flash security [1]. Not surprisingly, this attack is not prevented by the Flash Player. However, our type system catches this violation in the victim's code at compile time.

Example 3. Victim's swf passes a non-sensitive function to the attacker's swf

```
v = ⟦ns_fn = fun(x) {return x + 1}, a_swf =.., f = a_swf.f'(ns_fn)⟧
a = ⟦f' = fun(g) { g(a_acc) }⟧
```

As before, the victim's swf loads the attacker's swf but this time, passes a non-sensitive function to the attacker. Since it's not a violation of data integrity, this code is successfully type checked in our type system (and runs successfully too).

Example 4. Cross Site Scripting

Finally, the Cross Site Scripting attacks [1], which are one of the most common types of real world attacks on the Flash applications, are also instances of data integrity violations. In such attacks, the victim uses the untrusted run-time inputs, called FlashVars, as arguments to the function that navigates to a URL. The attacker can then provide malicious JavaScript code in the FlashVars, and that code ends up executing with the end-user privileges. Our type system can flag Cross Site Scripting vulnerabilities in the victim's swfs at compile time.

4 Evaluation Semantics

In this section we formalize the Flash security model.

Language Syntax The language syntax is shown in Figure 1. The set of web domains is denoted by $\mathbb{D}$. We assume that every swf has a unique identifier coming from a set $\mathbb{I}$. The identifiers are used as arguments to the loading functions.

The basic types consists of the base type ($\perp$), the function type ($\sigma_1 \to \sigma_2$), the object type ($\{x_i^{\kappa_i} : \sigma_i\}^{i \in [m]}$), and the type for the top level swf term ($[\![x_i^{\kappa_i} : \sigma_i]\!]^{i \in [m]}$). We model the top level swf terms like the objects. The properties in the object and the swf type have read-write capability κ, which is a subset of $\{\mathsf{r}, \mathsf{w}\}$. In particular, a property can be read (resp. written) if it has read (resp. write) capability r (resp. w). Types in the language are the basic types augmented with the integrity labels, $\mathcal{D}$. Each label is a set of some domains coming from $\mathbb{D}$. The interpretation of the labels is that a type with label $\mathcal{D}$ should admit only those values at run-time that *come from* one of the domains in $\mathcal{D}$.

Domain	d	$\in$	$\mathbb{D}$
Identifier	a, b, c	$\in$	$\mathbb{I}$
Capability	κ	$\subseteq$	$\{\mathsf{r}, \mathsf{w}\}$
Label	$\mathcal{D}$	$\subseteq$	$\mathbb{D}$
Basic Type	τ	$::=$	$\bot \mid \sigma_1 \to \sigma_2 \mid \{x_i^{\kappa_i} : \sigma_i\}^{i \in [m]} \mid [\![x_i^{\kappa_i} : \sigma_i]\!]^{i \in [m]}$
Type	σ	$::=$	$\tau_{\mathcal{D}}$
Term	t	$::=$	$\mathsf{null} \mid \mathsf{fun}\ (x : \sigma)\ t : \sigma' \mid t(t')$
		$\mid$	$\{x_i^{\kappa_i} : \sigma_i = t_i\}^{i \in [m]} \mid t.x \mid t.x = t'$
		$\mid$	$[\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]} \mid x \mid x = t$
		$\mid$	$\mathsf{if}\ t\ \mathsf{then}\ t'\ \mathsf{else}\ t'' \mid \mathsf{ld}(a) \mid \mathsf{i\_ld}(a)$

Fig. 1. Language Syntax

Although ActionScript is a gradually typed language [4], we do not model the dynamic type here. In our previous work [5], we have designed a type inference algorithm that can be used to evolve a gradually typed program to a statically typed program. In the current setting, we assume that the programmer has already removed all the dynamic types from his code and start from a statically typed language, focusing mainly on the security aspects of the platform.

The terms in the language include (read from left to right and top to bottom in Figure 1): the base value, function definitions, function applications, object literals, property reads and writes (from both objects and top level swfs), top level swf, variables reads and writes, and null checks. There are two special variables in the language x_{parent} and x_{self}. x_{parent} is used to access the swf that loads the current swf and x_{self} is used to access the current swf. For the first swf loaded in the runtime, x_{parent} and x_{self} evaluate to the same value.

Finally, the terms $\mathsf{ld}(a)$ and $\mathsf{i\_ld}(a)$ represent the content loading and import loading, respectively, of the swf a. We note that the term $[\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]}$ cannot be a subterm of any swf. A swf does not contain the code for another swf, it just loads or import loads another swf at run-time.

Evaluation Semantics The evaluation judgments are shown in Figure 2 and Figure 3. The syntax of the values is as follows:

$$\text{Value } v ::= \mathsf{null}^d \mid \ell^d \mid \mathcal{L}_d \mid \lambda_{S,d} x^{\sigma}.\ t^{\sigma'} \mid \mathsf{abort}$$

The d annotation in the values represents the run-time domain of the swf where the corresponding value originates. The value null^d represents the value of the base type, ℓ^d denotes a location, and $\mathcal{L}_d$ denotes the top level swf location for a swf executing in domain d.

S represents a stack: a sequence of locations. The value $\lambda_{S,d} x^{\sigma}.\ t^{\sigma'}$ denotes a function definition with the closure S. The type annotations σ and σ' are the static type annotations on the function parameter and the return type.

The d annotations in null^d, ℓ^d, and the σ and σ' annotations in $\lambda_{S,d} x^{\sigma}.\ t^{\sigma'}$ (all the superscripts in the values), are not carried and used by the actual Flash

Evaluation judgment $S \vdash^d_\rho (H, t) \Downarrow (H', v)$

(E-Null)

$$\frac{}{S \vdash^d_\rho (H, \mathsf{null}) \Downarrow (H, \mathsf{null}^d)}$$

(E-VarR)

$$\frac{H[x]_S = v}{S \vdash^d_\rho (H, x) \Downarrow (H, v)}$$

(E-VarW)

$$\frac{S \vdash^d_\rho (H, t) \Downarrow (H', v) \qquad H'[x \Rightarrow v]_S = H''}{S \vdash^d_\rho (H, x = t) \Downarrow (H'', v)}$$

(E-Fun)

$$\frac{}{S \vdash^d_\rho (H, \mathsf{fun}\ (x : \sigma)\ t : \sigma') \Downarrow (H, \lambda_{S,d} x^\sigma.\ t^{\sigma'})}$$

(E-Obj)

$$\frac{\begin{array}{c}\ell^d \text{ is fresh} \\ H_1 = H[\ell^d \mapsto [x_1^{\kappa_1,\sigma_1} \mapsto \mathsf{null}, \ldots, x_m^{\kappa_m,\sigma_m} \mapsto \mathsf{null}]] \\ \forall i \in [m].\quad S :: \ell^d \vdash^d_\rho (H_i, t_i) \Downarrow (H_i', v_i) \quad H_i'[x_i \Rightarrow v_i]_{S::\ell^d} = H_{i+1}\end{array}}{S \vdash^d_\rho (H, \{x_i^{\kappa_i} : \sigma_i = t_i\}^{i \in [m]}) \Downarrow (H_{m+1}, \ell^d)}$$

(E-PropR)

$$\frac{S \vdash^d_\rho (H, t) \Downarrow (H', \ell^{d'}) \qquad v_j = H'(\ell^{d'})(x_j^{\kappa_j,\sigma_j})}{S \vdash^d_\rho (H, t.x_j) \Downarrow (H', v_j)}$$

(E-PropW)

$$\frac{\begin{array}{c}S \vdash^d_\rho (H, t) \Downarrow (H', \ell^{d'}) \\ S \vdash^d_\rho (H', t_j) \Downarrow (H_j, v_j) \qquad H'' = H_j[\ell^{d'} \mapsto H_j(\ell^{d'})[x_j^{\kappa_j,\sigma_j} \mapsto v_j]]\end{array}}{S \vdash^d_\rho (H, t.x_j = t_j) \Downarrow (H'', v_j)}$$

(E-App)

$$\frac{\begin{array}{c}S \vdash^d_\rho (H, t) \Downarrow (H', (\lambda_{S',d'} x^\sigma.\ t_2{}^{\sigma_2}) \qquad S \vdash^d_\rho (H', t_1) \Downarrow (H_1, v_1) \\ \ell^{d'} \text{ is fresh} \quad H'' = H_1[\ell^{d'} \mapsto [x^{\{\mathsf{r},\mathsf{w}\},\sigma} \mapsto v_1]] \\ S' :: \ell^{d'} \vdash^{d'}_\rho (H'', t_2) \Downarrow (H_2, v_2)\end{array}}{S \vdash^d_\rho (H, t\ (t_1)) \Downarrow (H_2, v_2)}$$

(E-If)

$$\frac{\begin{array}{c}S \vdash^d_\rho (H, t) \Downarrow (H', v) \\ v \neq \mathsf{null} \Rightarrow i = 1 \qquad v = \mathsf{null} \Rightarrow i = 2 \qquad S \vdash^d_\rho (H', t_i) \Downarrow (H'', v')\end{array}}{S \vdash^d_\rho (H, \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \Downarrow (H'', v')}$$

Fig. 2. Evaluation Judgments

Evaluation judgment $S \vdash^d_\rho (H, t) \Downarrow (H', v)$

(E-SwfR)

$$\frac{S \vdash^d_\rho (H, t) \Downarrow (H', \mathcal{L}_{d'}) \qquad d \in \rho(d') \qquad v_j = H'(\mathcal{L}_{d'})(x_j^{\kappa_j, \sigma_j})}{S \vdash^d_\rho (H, t.x_j) \Downarrow (H', v_j)}$$

(E-SwfW)

$$\frac{\begin{array}{c} S \vdash^d_\rho (H, t) \Downarrow (H', \mathcal{L}_{d'}) \\ d \in \rho(d') \qquad S \vdash^d_\rho (H', t_j) \Downarrow (H_j, v_j) \qquad H'' = H_j[\mathcal{L}_{d'} \mapsto H_j(\mathcal{L}_{d'})[x_j^{\kappa_j, \sigma_j} \mapsto v_j]] \end{array}}{S \vdash^d_\rho (H, t.x_j = t_j) \Downarrow (H'', v_j)}$$

(E-Swf)

$$\frac{\begin{array}{c} \mathcal{L}_{d'} = H[x_{\mathsf{self}}]_S \qquad \mathcal{L}_d \text{ is fresh} \\ H_1 = H[\mathcal{L}_d \mapsto [x_{\mathsf{parent}} \mapsto \mathcal{L}_{d'}, x_{\mathsf{self}} \mapsto \mathcal{L}_d, x_1^{\kappa_1, \sigma_1} \mapsto \mathsf{null}, \ldots, x_m^{\kappa_m, \sigma_m} \mapsto \mathsf{null}]] \\ \forall i \in [m]. \quad \mathcal{L}_d \vdash^d_\rho (H_i, t_i) \Downarrow (H_i', v_i) \quad H_i'[x_i \Rrightarrow v_i]_{\mathcal{L}_d} = H_{i+1} \end{array}}{S \vdash^d_\rho (H, [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]}) \Downarrow (H_{m+1}, \mathcal{L}_d)}$$

(E-Load)

$$\frac{\begin{array}{c} d' \text{ is domain of } a \\ \delta_{\mathsf{ld}}(a) = [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]} \qquad S \vdash^{d'}_\rho (H, [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]}) \Downarrow (H', \mathcal{L}_{d'}) \end{array}}{S \vdash^d_\rho (H, \mathsf{ld}(a)) \Downarrow (H', \mathcal{L}_{d'})}$$

(E-ImportLoad)

$$\frac{\begin{array}{c} \text{verify that there is an entry for } d \text{ in the crossdomain policy file on } a\text{'s webserver} \\ \delta_{\mathsf{ld}}(a) = [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]} \qquad S \vdash^d_\rho (H, [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]}) \Downarrow (H', \mathcal{L}_d) \end{array}}{S \vdash^d_\rho (H, \mathsf{i\_ld}(a)) \Downarrow (H', \mathcal{L}_d)}$$

Fig. 3. Evaluation Judgments - SWF Loading and Cross-Scripting

runtime semantics. We keep them around for formalization purposes. The rules in Figure 2 and Figure 3 do not depend on these.

The special value abort denotes a failed run-time access control check, terminating the Flash Player execution.

A record is a map from variables to values. As with the superscripts in the values, we carry around the capability κ and the static type annotation σ, with the variables. A heap H is a map from locations to records.

When a swf tries to query or update the heap, under a stack S, the following rules apply:

$$H[x \Rrightarrow v]_{S::\ell^d} = \begin{cases} H[\ell^d \mapsto H(\ell^d)[x^{\kappa,\sigma} \mapsto v]] \text{ if } x^{\kappa,\sigma} \in \mathtt{dom}(H(\ell^d)) \\ H[x \Rrightarrow v]_S \qquad \text{otherwise} \end{cases}$$

$$H[x]_{S::\ell^d} = \begin{cases} H(\ell^d)(x^{\kappa,\sigma}) \text{ if } x^{\kappa,\sigma} \in \mathtt{dom}(H(\ell^d)) \\ H[x]_S \qquad \text{otherwise} \end{cases}$$

We say $H[x]_S$ resolves to $x^{\kappa,\sigma}$ whenever the query operation above finds a $x^{\kappa,\sigma}$ in the scope.

We model trust assumptions by $\rho : \mathbb{D} \to 2^{\mathbb{D}}$, where $\rho(d)$ is the set of domains that d trusts. In the actual Flash API, ρ is a function of swf identifiers $\mathbb{I}$ rather than $\mathbb{D}$. However, since same domain swfs can freely script into each other, it is advisable to host swfs with different trust assumptions on different domains [1]. Also, the actual Flash API builds up the trust relationship dynamically. We, on the other hand, assume ρ to be fixed. We believe this is a reasonable assumption for security critical swfs.

The evaluation judgments are of the form $S \vdash^d_\rho (H, t) \Downarrow (H', v)$, where d is the domain under which t is evaluating.

The rules (E-VarR) and (E-VarW) use the heap query and update operations defined above. The rule (E-Fun) records the stack S and the current domain d with the function value. A function, upon application, evaluates in the context of the domain where it originates. In the rule (E-App), the function body evaluation takes place in the context of d', even if the application term t (t') is executing in d. Thus, irrespective of whether d' trusts d or not, if a swf from domain d can get access to a function defined in d', it can execute the function on arbitrary arguments with the privileges of d'.

The rule (E-Obj) evaluates each property in the object. In the rules (E-PropR), and (E-PropW), there are no access checks (just like (E-App)). As long as a swf from domain d can get access to a location $\ell^{d'}$, it can read or write to it freely, irrespective of whether d' trusts d or not.

The rules for the swf loading and the top level swf accesses are given in Figure 3. When a swf from domain d tries to script directly into another swf from d' through the top level location $\mathcal{L}_{d'}$, the semantics of (E-SwfR) and (E-SwfW) verify that d' must trust d, i.e. $d \in \rho(d')$. If these checks do not succeed, the execution terminates and results in the value abort (we do not show the rules for abort for space limitation).

The rule (E-Swf) evaluates the top level swf term. It also sets up the variable x_{parent} for the new swf as the value of x_{self} in current stack, which is the top level swf location of the loading swf. For the rule (E-Load), $\delta_{\mathsf{ld}}(a)$ denotes the runtime operation of loading the swf a over the network. The loading operation results in a term of the form $[\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i \in [m]}$. The term is then evaluated under a's own domain.

For import loading, as mentioned in Section 2, the Flash Player verifies that crossdomain policy file on a's webserver contains an entry for d. The swf a is then evaluated in the context of d.

We now revisit the examples introduced earlier to see their behavior under the runtime semantics.

Example 1 During the execution of the attacker's swf `a`, the victim's swf `v` is loaded and evaluated using the (E-Load) rule. The rule results in a location $\mathcal{L}_{\texttt{d'}}$ s.t. in the heap, the variable `s_fn` in $H(\mathcal{L}_{\texttt{d'}})$ is mapped to the sensitive function. The variable `v_swf`, during the evaluation of the property `f` in the attacker's swf, evaluates to $\mathcal{L}_{\texttt{d'}}$. But the read of `s_fn` fails in the rule (E-SwfR), because $\texttt{d} \notin \rho(\texttt{d'})$, as the victim does not trust the attacker. This way, the attack is prevented by the runtime semantics.

Example 2 In the victim's swf `v`, the property `a_swf` evaluates to a location $\mathcal{L}_{\mathtt{d}}$. During the evaluation of `f` in the victim's swf, the access of `f'` in the attacker's swf succeeds, since `d'` $\in \rho(\mathtt{d})$. Thus, the code in attacker's swf, `g(a_acc)`, executes in the attacker's context, as per the rule (E-App). The variable `g` evaluates to the victim's sensitive function which then executes in the victim's context with the attacker provided argument `a_acc`. Thus, the runtime semantics fails to prevent this attack.

Example 3 As with Example 2, the victim's (and the attacker's) swf executes successfully in this case too.

Cross Site Scripting Since the semantics does not reason about data integrity, it fails to prevent the Cross Site Scripting attacks also.

4.1 Formal Definition of Data Integrity

We first define an origin function on the values as follows:

Definition 1 (Value Origin). *The* origin *function for the values is defined as:* $\mathsf{origin}(\mathsf{null}^d) = d$, $\mathsf{origin}(\ell^d) = d$, $\mathsf{origin}(\mathcal{L}_d) = d$, $\mathsf{origin}(\lambda_{S,d} x^{\sigma}.\ t^{\sigma'}) = d$

We define a location as trusted if for all the variables in the location, the contained value has an origin consistent with the static type of the variable.

Definition 2 (Trusted Location and Trusted Heap). *A location* ℓ^d *in heap* H *is trusted if* $\forall x^{\kappa,\sigma} \in dom(H(\ell^d))$, *whenever* $H(\ell^d)(x^{\kappa,\sigma}) = v$, *we have* $\mathsf{origin}(v) \in \mathcal{D}$ *s.t.* $\sigma = \tau_{\mathcal{D}}$ *for some* τ. *A heap* H *is trusted, written as* $H\surd$, *if* $\forall \ell^d \in dom(H)$, ℓ^d *is trusted.*

The data integrity property is now defined as:

Definition 3 (Data Integrity for a Code Execution). *Let* $S \vdash^d_\rho (H, t) \Downarrow (H', v)$ *be an execution. Suppose* $H\surd$. *Then, either* $v = \mathsf{abort}$ *or* $H'\surd$.

In Example 2 above, since `s_fn` is expected to be called with a trusted argument, its parameter type has the integrity label {`d'`}. However, at run time, the value `a_acc` which has label {`d`}, since it originates in the attacker's swf, is able to flow into the `s_fun` parameter. Thus, the data integrity property is violated.

5 Type System

In this section we present a type system that, in conjunction with the Flash run-time checks, enforces the data integrity property.

We first define a static consistency relation, $\preceq$, over the basic types τ, and the types σ. As with the usual subtyping relation, the interpretation of $\sigma_1 \preceq \sigma_2$ is that it is safe for a term of type σ_1 to flow to a context that expects a term of type σ_2. The static consistency relation verifies that the types obey the *comes*

from invariants at each level, by checking that the sub-parts in the types also satisfy the relation.

The judgments for $\preceq$ are shown in Figure 4. The rule (S-Type) checks that $\tau_1 \preceq \tau_2$ and $\mathcal{D}_1 \subseteq \mathcal{D}_2$. The label check ensures that if the context expects a value originating in one of the domains from $\mathcal{D}_2$, then σ_1 should satisfy this invariant. The rules (S-Fun) and (S-Obj) are standard, and (S-Obj) also admits the usual record subtyping. As a minor technical convenience, we do not allow record depth subtyping in the rule (S-Swf).

Type Checking The typing judgments are shown in Figure 4. These judgments are of the form $\Gamma \vdash^d_\rho t :: \sigma$, where d is the expected run-time domain of the swf. The type environment Γ has two components: (Γ_s, Γ_v). Γ_s contains the type assumptions about the parent swf that would load the current swf at run-time, and about the swfs that the current swf might load itself. Γ_v contains the standard type bindings for the free variables in t.

The rule (C-Null) assigns the type $\bot_{\{d\}}$ to the value null, which denotes that this base value *comes from* the domain d. Similarly the rules (C-Fun) and (C-Obj) assign the integrity label $\{d\}$ to the final type. The rules (C-Parent), (C-Load), and (C-ImportLoad) look up the top level swf type from Γ_s. (C-ImportLoad) also ensures the swf a comes from a trusted domain.

The rules for objects and functions are standard and use the static consistency relation, $\preceq$, in place of the subtyping relation. The rules (E-SwfR) and (E-SwfW) additionally verify that the swf access is allowed per the trust assumptions, i.e. $d \in \rho(d')$, where d is the current domain and d' is the domain of the swf being accessed. The rule (C-Swf) type checks the top level swf term.

The rules (C-SwfR') and (C-SwfW') are interesting. As a first step towards strengthening the attacker capabilities, we allow the attacker to evade the trust assumptions, ρ, when type checking its own code. If the attacker's code tries to script into a swf, which does not trust the attacker's domain, we do not raise a type error. Since the Flash Player runtime already has dynamic access control checks, we can allow for these relaxed static rules. We define a lowering operation on the types that changes all the labels in a type to $\mathbb{D}$, unless it's a top level swf type.

Definition 4 (Lowering Type). *The lowering operation, $\sigma\downarrow$, is defined as:*

- $\bot_{\mathcal{D}}\downarrow = \bot_{\mathbb{D}}$, $(\sigma_1 \to \sigma_2)_{\mathcal{D}}\downarrow = (\sigma_1\downarrow \to\ \sigma_2\downarrow)_{\mathbb{D}}$
- $\{x_i^{\kappa_i} : \sigma_i\}^{i\in[m]}_{\mathcal{D}}\downarrow = \{x_i^{\kappa_i} : \sigma_i\downarrow\}^{i\in[m]}_{\mathbb{D}}$, $[\![x_i^{\kappa_i} : \sigma_i]\!]^{i\in[m]}_{\{d\}}\downarrow = [\![x_i^{\kappa_i} : \sigma_i]\!]^{i\in[m]}_{\{d\}}$

We use this lowering operation in the rules (C-SwfR') and (C-SwfW') to assign the final type as $\sigma_j\downarrow$, where x_j is the property that the attacker's code accesses. This choice gives the attacker maximum flexibility in type checking its code.

Note that a top level swf type remains unchanged across lowering. It can be easily seen that this does not reduce the attacker's capability.

Static Consistency $\tau_1 \preceq \tau_2 \qquad \sigma_1 \preceq \sigma_2$

(S-Type)
$$\frac{\sigma_1 = (\tau_1)_{\mathcal{D}_1} \quad \sigma_2 = (\tau_2)_{\mathcal{D}_2} \quad \tau_1 \preceq \tau_2 \quad \mathcal{D}_1 \subseteq \mathcal{D}_2}{\sigma_1 \preceq \sigma_2}$$

(S-Base)
$$\frac{}{\bot \preceq \bot}$$

(S-Swf)
$$\frac{n \leq m}{[\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d\}}^{i\in[m]} \preceq [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d\}}^{i\in[n]}}$$

(S-Fun)
$$\frac{\sigma_3 \preceq \sigma_1 \qquad \sigma_2 \preceq \sigma_4}{\sigma_1 \to \sigma_2 \preceq \sigma_3 \to \sigma_4}$$

(S-Obj)
$$\frac{n \leq m \qquad \forall i \in [n].\quad \kappa_i' \subseteq \kappa_i \quad \sigma_i \preceq \sigma_i' \text{ when } \mathsf{r} \in \kappa_i' \quad \sigma_i' \preceq \sigma_i \text{ when } \mathsf{w} \in \kappa_i'}{\{x_i^{\kappa_i} : \sigma_i\}^{i\in[m]} \preceq \{x_i^{\kappa_i'} : \sigma_i'\}^{i\in[n]}}$$

Typing Judgment $\Gamma \vdash_\rho^d t :: \sigma$

(C-Null)
$$\frac{}{\Gamma \vdash_\rho^d \mathsf{null} :: \bot_{\{d\}}}$$

(C-Parent)
$$\frac{}{\Gamma \vdash_\rho^d x_{\mathsf{parent}} :: \Gamma_s(x_{\mathsf{parent}})}$$

(C-VarR)
$$\frac{}{\Gamma \vdash_\rho^d x :: \Gamma_v(x)}$$

(C-Load)
$$\frac{}{\Gamma \vdash_\rho^d \mathsf{ld}(a) :: \Gamma_s(a)}$$

(C-VarW)
$$\frac{\Gamma_v(x) = \sigma \qquad \Gamma \vdash_\rho^d t :: \sigma' \qquad \sigma' \preceq \sigma}{\Gamma \vdash_\rho^d x = t :: \sigma}$$

(C-Fun)
$$\frac{\Gamma[x \mapsto \sigma_1] \vdash_\rho^d t_2 :: \sigma_2' \qquad \sigma_2' \preceq \sigma_2}{\Gamma \vdash_\rho^d \mathsf{fun}\ (x{:}\sigma_1)\ t_2{:}\sigma_2 :: (\sigma_1 \to \sigma_2)_{\{d\}}}$$

(C-ImportLoad)
$$\frac{d' \text{ is } a\text{'s domain} \qquad d' \in \rho(d)}{\Gamma \vdash_\rho^d \mathsf{i\_ld}(a) :: \Gamma_s(a)}$$

(C-Obj)
$$\frac{\Gamma' = \Gamma[x_1 \mapsto \sigma_1, \ldots, x_m \mapsto \sigma_m] \qquad \forall i \in [m].\quad \Gamma' \vdash_\rho^d t_i :: \sigma_i' \quad \sigma_i' \preceq \sigma_i}{\Gamma \vdash_\rho^d \{x_i^{\kappa_i} : \sigma_i = t_i\}^{i\in[m]} :: \{x_i^{\kappa_i} : \sigma_i\}_{\{d\}}^{i\in[m]}}$$

(C-PropR)
$$\frac{\Gamma \vdash_\rho^d t :: \{x_i^{\kappa_i} : \sigma_i\}_{\mathcal{D}}^{i\in[m]} \qquad j \in [m] \quad \mathsf{r} \in \kappa_j}{\Gamma \vdash_\rho^d t.x_j :: \sigma_j}$$

(C-PropW)
$$\frac{\Gamma \vdash_\rho^d t :: \{x_i^{\kappa_i} : \sigma_i\}_{\mathcal{D}}^{i\in[m]} \qquad j \in [m] \quad \mathsf{w} \in \kappa_j \qquad \Gamma \vdash_\rho^d t_j :: \sigma_j' \qquad \sigma_j' \preceq \sigma_j}{\Gamma \vdash_\rho^d t.x_j = t_j :: \sigma_j}$$

(C-App)
$$\frac{\Gamma \vdash_\rho^d t :: (\sigma_1 \to \sigma_2)_{\mathcal{D}} \qquad \Gamma \vdash_\rho^d t_1 :: \sigma_1' \qquad \sigma_1' \preceq \sigma_1}{\Gamma \vdash_\rho^d t\ (t_1) :: \sigma_2}$$

(C-If)
$$\frac{\Gamma \vdash_\rho^d t :: \sigma \qquad \Gamma \vdash_\rho^d t_1 :: (\tau)_{\mathcal{D}_1} \qquad \Gamma \vdash_\rho^d t_2 :: (\tau)_{\mathcal{D}_2}}{\Gamma \vdash_\rho^d \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 :: (\tau)_{\mathcal{D}_1 \cup \mathcal{D}_2}}$$

(C-SwfR)
$$\frac{\Gamma \vdash_\rho^d t :: [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d'\}}^{i\in[m]} \qquad d \in \rho(d') \quad j \in [m] \quad \mathsf{r} \in \kappa_j}{\Gamma \vdash_\rho^d t.x_j :: \sigma_j}$$

(C-SwfW)
$$\frac{\Gamma \vdash_\rho^d t :: [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d'\}}^{i\in[m]} \qquad d \in \rho(d') \quad j \in [m] \quad \mathsf{w} \in \kappa_j \qquad \Gamma \vdash_\rho^d t_j :: \sigma_j' \qquad \sigma_j' \preceq \sigma_j}{\Gamma \vdash_\rho^d t.x_j = t_j :: \sigma_j}$$

(C-SwfR')
$$\frac{\Gamma \vdash_\rho^d t :: [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d'\}}^{i\in[m]} \qquad d \notin \rho(d') \quad j \in [m] \quad \mathsf{r} \in \kappa_j}{\Gamma \vdash_\rho^d t.x_j :: \sigma_j \downarrow}$$

(C-SwfW')
$$\frac{\Gamma \vdash_\rho^d t :: [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d'\}}^{i\in[m]} \qquad d \notin \rho(d') \quad j \in [m] \quad \mathsf{w} \in \kappa_j \qquad \Gamma \vdash_\rho^d t_j :: \sigma_j' \qquad \sigma_j' \preceq \sigma_j \downarrow}{\Gamma \vdash_\rho^d t.x_j = t_j :: \sigma_j \downarrow}$$

(C-Swf)
$$\frac{\Gamma' = \Gamma[x_1 \mapsto \sigma_1, \ldots, x_m \mapsto \sigma_m] \qquad \forall i \in [m].\quad \Gamma' \vdash_\rho^d t_i :: \sigma_i' \quad \sigma_i' \preceq \sigma_i}{\Gamma \vdash_\rho^d [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i\in[m]} :: [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d\}}^{i\in[m]}}$$

Fig. 4. Type Checking Rules

Value Typing $\vdash_H v :: \sigma$

$$\frac{}{\vdash_H \mathsf{null}^d :: \bot_{\{d\}}}\ (\text{V-Null}) \qquad \frac{x_i^{\kappa_i,\sigma_i} \in \mathtt{dom}(H(\ell^d))}{\vdash_H \ell^d :: \{x_i^{\kappa_i} : \sigma_i\}^i_{\{d\}}}\ (\text{V-Loc}) \qquad \frac{\sigma_1 = (\sigma \to \sigma')_{\{d\}}}{\vdash_H \lambda_{S,d} x^{\sigma}.\ t^{\sigma'} :: \sigma_1}\ (\text{V-Fun}) \qquad \frac{x_i^{\kappa_i,\sigma_i} \in \mathtt{dom}(H(\mathcal{L}_d))}{\vdash_H \mathcal{L}_d :: [\![x_i^{\kappa_i} : \sigma_i]\!]^i_{\{d\}}}\ (\text{V-SwfLoc})$$

Fig. 5. Value Typing

5.1 Soundness

We now describe the soundness properties of our type system. We first define the typing for values, written as $\vdash_H v :: \sigma$. The judgments for this relation are shown in Figure 5. We also define consistency of the type environment with the runtime environment as follows:

Definition 5 (Consistency of Type Environment with Loading Function). *Γ_s is said to be consistent with the swf loading function δ_{ld}, $\Gamma_s \sim \delta_{\mathsf{ld}}$, if whenever $\delta_{\mathsf{ld}}(a) = [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i\in[m]}$ and a is loaded under domain d, then $(\Gamma_s, \phi) \vdash^d_\rho [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i\in[m]} :: [\![x_i^{\kappa_i} : \sigma_i]\!]^{i\in[m]}_{\{d\}}$ and $[\![x_i^{\kappa_i} : \sigma_i]\!]^{i\in[m]}_{\{d\}} \preceq \Gamma_s(a)$.*

Definition 6 (Consistency of Type Environment with Stack and Heap). *$\Gamma \sim_{\rho,H} S$ if whenever $\Gamma_v(x) = \sigma$ and $H[x]_S$ resolves to $x^{\kappa,\sigma'}$, we have $\sigma' \preceq \sigma$. Also, if $H[x_{\mathsf{parent}}]_S = \mathcal{L}_d$, $\vdash_H \mathcal{L}_d :: \sigma''$, then $\sigma'' \preceq \Gamma_s(x_{\mathsf{parent}})$.*

Our type system supports a stronger notion of trusted locations that subsumes Definition 2.

Definition 7 (Stronger Definition of Trusted Location). *In a heap H, a location ℓ^d is trusted if $\forall x^{\kappa,\sigma} \in dom(H(\ell^d))$, if $v = H(\ell^d)(x^{\kappa,\sigma})$ then $\vdash_H v :: \sigma'$ s.t. $\sigma' \preceq \sigma$.*

The definitions of trusted heap and data integrity under code execution remain same except for the use of the new definition of trusted locations.

Our main soundness theorem says that for an execution of a well-typed term, starting from a trusted heap, either the execution results in an error or in a new heap that is also trusted. Moreover, the type of the value is statically consistent with the type of the term.

Theorem 1 (Soundness of Type System). *If $\Gamma \vdash^d_\rho t :: \sigma$, $S \vdash^d_\rho (H, t) \Downarrow (H', v)$, $\Gamma_s \sim \delta_{\mathsf{ld}}$, $\Gamma \sim_{\rho,H} S$, and $H\surd$, then either $v = \mathsf{abort}$ or $\vdash_{H'} v :: \sigma'$ s.t. $\sigma' \preceq \sigma$ and $H'\surd$.*

A More Powerful Attacker The interesting property of our type system is that it allows an attacker to by-pass the integrity labels type checking completely. We place only two restrictions on the attacker's code: (a) it must be *structurally*

well-typed, and (b) it must use the other top level swf types correctly. The restriction (b) does not limit the attacker's capability since it can always type check the other swfs' properties accesses via (C-SwfR') and (C-SwfW').

When type checking its own swf, the victim must make the worst possible assumptions about the type labels in such an attacker's code. In particular, it must assume that all the values that come from such an attacker's swf could have originated anywhere, and therefore it must use the integrity label $\mathbb{D}$ in the type assumptions about the attacker's code. However, it need not change the labels inside the top level swf types used in the attacker's swf. We define $\Gamma_s \downarrow$ as: $\Gamma_s \downarrow (a) = \Gamma_s(a)$ if a is a victim's swf, or $[\![x_i^{\kappa_i} : \sigma_i \downarrow]\!]_{\{d\}}^{i\in[m]}$ if a is an attacker's swf s.t. $\Gamma_s(a) = [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d\}}^{i\in[m]}$. The victim should use $\Gamma_s \downarrow$ to type check its code.

However, the type soundness theorem requires the Γ_s to be consistent with the loading function. We prove that if an attacker's swf follows the above restrictions (a) and (b) *only*, then it can be labeled to be well-typed under $\Gamma_s \downarrow$.

Let us define a typing judgment $\Gamma \vdash'^d_\rho t :: \sigma$ which is same as Figure 4, except that the rule (S-Type) does not perform the $\mathcal{D}_1 \subseteq \mathcal{D}_2$ check. Note that, however, the rule (S-Swf) still forces the attacker to use the top level swf types consistently. We define $t \downarrow$ and $\Gamma_v \downarrow$ as the obvious extensions of $\sigma \downarrow$. Then:

Theorem 2 (Attacker SWF Typing). *Let* $t = [\![x_i^{\kappa_i} : \sigma_i = t_i]\!]^{i\in[m]}$ *be an attacker swf code s.t.* $(\Gamma_s \downarrow, \phi) \vdash'^d_\rho t :: [\![x_i^{\kappa_i} : \sigma_i]\!]_{\{d\}}^{i\in[m]}$. *Then,* $(\Gamma_s \downarrow, \phi) \vdash^d_\rho [\![x_i^{\kappa_i} : \sigma_i \downarrow = t_i \downarrow]\!]^{i\in[m]} :: [\![x_i^{\kappa_i} : \sigma_i \downarrow]\!]_{\{d\}}^{i\in[m]}$.

This theorem is a direct corollary of the following lemma:

Lemma 1 (Attacker Term Typing). *Let* t *be a subterm in the attacker's code s.t.* $(\Gamma_s \downarrow, \Gamma_v) \vdash'^d_\rho t :: \sigma$. *Then,* $(\Gamma_s \downarrow, \Gamma_v \downarrow) \vdash^d_\rho t \downarrow :: \sigma'$ *s.t.* $\sigma' \preceq \sigma \downarrow$.

So now, $\Gamma_s \downarrow$ is consistent with δ_{ld} for the victim's swfs as well as for any attacker's swf that is structurally well-typed but is not subjected to the static integrity labels checking. Hence $\Gamma_s \downarrow \sim \delta_{\mathsf{ld}}$. The type soundness theorem now ensures that the data integrity property is maintained at run time for the swfs typed under $\Gamma_s \downarrow$. Thus, the victim can enforce data integrity all by itself.

We now see how the type system works for our running examples.

Example 2 When type checking its code, the victim would use the following type for the attacker's swf:

$\Gamma(\mathtt{a}) = \{\mathtt{f'} : ((\mathtt{int}_{\mathbb{D}} \to \mathtt{int}_{\mathbb{D}})_{\mathbb{D}} \to \mathtt{int}_{\mathbb{D}})_{\mathbb{D}}\ \}_{\{d\}}$

The sensitive function has the type:

$\mathtt{s\_fn} : (\mathtt{int}_{\{d'\}} \to \mathtt{int}_{\{d'\}})_{\{d'\}}$

As expected, the victim's code fails to type check under these assumptions. For the function call `a_swf.f'(s_fn)`, the type checker checks that the type of `s_fn` is statically consistent with the argument type of `a_swf.f'`, which fails as shown in the following derivation:

$$\dfrac{\dfrac{\dfrac{\texttt{int} \preceq \texttt{int} \quad \boxed{\mathbb{D} \subseteq \{d'\}}}{\texttt{int}_{\mathbb{D}} \preceq \texttt{int}_{\{d'\}} \qquad \texttt{int}_{\{d'\}} \preceq \texttt{int}_{\mathbb{D}}}}{\texttt{int}_{\{d'\}} \to \texttt{int}_{\{d'\}} \preceq \texttt{int}_{\mathbb{D}} \to \texttt{int}_{\mathbb{D}} \qquad \{d'\} \subseteq \mathbb{D}}}{(\texttt{int}_{\{d'\}} \to \texttt{int}_{\{d'\}})_{\{d'\}} \preceq (\texttt{int}_{\mathbb{D}} \to \texttt{int}_{\mathbb{D}})_{\mathbb{D}}}$$

Since $\mathbb{D} \not\subseteq \{d'\}$, the typing derivation fails, and thus, the data integrity violation is prevented at compile time at the victim's end.

Example 3 In this case the non-sensitive function has the type:
$\texttt{ns\_fn} : (\texttt{int}_{\mathbb{D}} \to \texttt{int}_{\mathbb{D}})_{\{d'\}}$
and we can now see that the victim's code will successfully typecheck.

Cross Site Scripting Our type system can also prevent Cross Site Scripting attacks. FlashVars can be annotated with the integrity label $\mathbb{D}$ and the parameter of the URL navigation function can be assigned the label $\{d'\}$, where d' is the victim's domain. Once this is done, the type checker ensures that the FlashVars cannot be passed as arguments to the URL navigation function.

6 Related Work

Flash Security Model Unlike JavaScript's binary trust model, where it's either no trust or full trust between the principals, Flash allows controlled communication among otherwise isolated clients. The existing model when combined with our type system, provides stronger data integrity guarantees to the programmers. Thus, it could be worthwhile to explore Flash as the platform for client mashups investigated in [6].

Flash Security Analysis DeVries et. al. [7] developed an inline reference monitoring system for enforcing security policies in malicious ActionScript code. They first inject runtime security guards in untrusted swf bytecode and then verify that it obeys the desired security policies.

However, they do not present a formal study of the Flash security model, as we do here. Moreover, their problem statement is fundamentally different from ours. They seek to sanitize untrusted swfs according to certain security policies whereas the goal of our type system is to help the victim ensure that its code maintains data integrity invariants, even in the presence of untrusted swfs.

Jang et. al. [8] study the server based aspect of Flash security – the crossdomain policy files. They present an empirical study of crossdomain policy files for Alexa top 50,000 websites. In this paper, we have mainly focused on the client side aspect of Flash security.

Security Type Systems A characteristic feature of our system is the combination of static and dynamic checks to enforce data integrity. In that sense, our system is similar to hybrid type checking [9]. Chaudhuri et. al. [10] present a

similar type system, that in conjunction with the dynamic checks, enforces data integrity in Windows Vista.

Our type system maintains *comes from* invariants, where the static type of a term is an over-approximation of the origin of the run-time value that the term evaluates to. There is a long history of security type systems that guarantee various properties for well-typed programs ([11–13]). Sabelfeld et. al. [14] present a detailed survey of language based information flow security.

7 Conclusion

In this paper, we have presented the first formal model of the security features in the Flash platform. We find that in its current form, the model is vulnerable to attacks violating data integrity property in victim's programs when they execute alongside untrusted programs. We presented a static type system, that in conjunction with existing runtime checks, prevents such attacks. We stated the soundness theorem of our type system and proved that a well-typed program maintains its data integrity invariants, even when co-executing with untrusted programs.

References

1. Adobe: Creating more secure SWF web applications , `http://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html`.
2. OWASP: Example Vulnerabilities, `https://www.owasp.org/index.php/Category:OWASP_Flash_Security_Project#Example_Vulnerabilities`.
3. Adobe: Adobe Flash Platform, `http://www.adobe.com/flashplatform/`.
4. Siek, J.G., Taha, W.: Gradual Typing for Functional Languages. In: Scheme and Functional Programming Workshop. (2006)
5. Rastogi, A., Chaudhuri, A., Hosmer, B.: The ins and outs of gradual type inference. In: POPL, ACM (2012)
6. Howell, J., Jackson, C., Wang, H.J., Fan, X.: Mashupos: operating system abstractions for client mashups. In: HotOS, USENIX Association (2007)
7. DeVries, B.W., Gupta, G., Hamlen, K.W., Moore, S., Sridhar, M.: Actionscript bytecode verification with co-logic programming. In: PLAS, ACM (2009)
8. Jang, D., Venkataraman, A., Sawka, G.M., Shacham, H.: Analyzing the cross-domain policies of flash applications. In: W2SP. (2011)
9. Flanagan, C.: Hybrid Type Checking. In: POPL, ACM (2006) 245–256
10. Chaudhuri, A., Naldurg, P., Rajamani, S.K.: A type system for data-flow integrity on windows vista. In: PLAS, ACM (2008)
11. Heintze, N., Riecke, J.G.: The slam calculus: programming with secrecy and integrity. In: POPL, ACM (1998)
12. Myers, A.C.: Jflow: practical mostly-static information flow control. In: POPL, ACM (1999)
13. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a java-like language. In: CSF, IEEE Computer Society (2002)
14. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications (2003)