Inscription: Thwarting ActionScript Web Attacks From Within

Meera Sridhar*, Abhinav Mohanty*, Fadi Yilmaz*, Vasant Tendulkar*, and Kevin W. Hamlen†
*Department of Software and Information Systems, University of North Carolina at Charlotte, NC, USA
†Department of Computer Science, University of Texas at Dallas, TX, USA
{msridhar, amohant1, fyilmaz}@uncc.edu, tendulkar.vasant@gmail.com[1], hamlen@utdallas.edu

***Abstract*—The design and implementation of Inscription, the first fully automated Adobe Flash binary code transformation system that can guard major Flash vulnerability categories without modifying vulnerable Flash VMs, is presented and evaluated. Inscription affords a means of mitigating the significant class of web attacks that target unpatched, legacy Flash VMs and their apps. Such legacy VMs, and the new and legacy Flash apps that they run, continue to abound in a staggering number of web clients and hosts today; their security issues routinely star in major annual threat reports and exploit kits worldwide. Through two complementary binary transformation approaches based on *in-lined reference monitoring*, it is shown that many of these exploits can be thwarted by a third-party principal (e.g., web page publisher, ad network, network firewall, or web browser) lacking the ability to universally patch all end-user VMs—write-access to the untrusted Flash *apps* (prior to execution) suffices. Detailed case-studies describing proof-of-concept exploits and mitigations for five major vulnerability categories are reported.**

I. Introduction

Exploits of legacy Flash VMs constitute one of the largest and highest impact attack surfaces of today's web. They are the primary vehicle for web-based ransomware and banking trojans, accounting for $\sim$80% of successful Nuclear exploits [12] and six of the top ten exploit kit vulnerabilities in 2016 [56]. More than 90% of malicious web pages abuse Flash, making Flash the #1 attack medium for malicious pages [44]. And the threat is growing: market proportion of Flash Player exploits grew more than 150% in 2016 relative to the previous year [18].

The prevalence of attacks that exploit known, patchable vulnerabilities in legacy Flash VMs can be traced to a perfect storm of at least three major trends: First, Flash's seamless integration of almost every major web media format (images, sounds, videos, etc.) into a highly portable bytecode binary with strong DRM capabilities [48], makes it extremely compelling for the highly dynamic web content desired by today's developers and end-users. Many popular content streaming websites such as Vudu [67] and HBO Go [28] rely on Flash to deliver content to their user-base. The widely used job-search platform, Glassdoor [20], and the internet performance measurement website, Ookla SpeedTest [50], use Flash for critical functionality. Miniclip [47], a Flash-based games website, receives over a million visitors daily.

Second, this power and flexibility has led to an extremely complex VM implementation that must support live streaming of all these different media formats, leading to a risk of implementation vulnerabilities associated with each format. As a result, the Flash Player regularly has among the top web vulnerability disclosures per year (e.g., it claimed the most CVEs of any application in 2016 [13]), and a rapidly evolving version history.

This rapid version churn inevitably means that hundreds of distinct Flash Player versions are currently deployed by end-users worldwide, each with its own vulnerabilities and idiosyncrasies [35, 33]. Studies estimate that nearly 62% of Internet Explorer users, 37% of Edge users, and 32% of Safari and Firefox users are running outdated Flash Player versions that leave them unprotected against well-known attacks [54].

Third, defense research on Flash has been impeded by the fact that the most widely deployed Flash VMs are closed-source, and content for them is purveyed in binary-only form without sources. The Flash ActionScript (AS) bytecode language has many features that make apps difficult to statically analyze at the binary level, including gradual typing, runtime code generation, dynamic class loading, and direct access to security-relevant system resources via a variety of runtime APIs [2]. Consequently, Flash defense has been significantly less studied in the scholarly literature relative to non-binary scripting languages, such as JavaScript [59].

The most common third-party Flash exploit protections deployed today therefore take the form of relatively weak network-level filters that scan transmitted Flash binaries for structural malformities known to trigger bugs in unpatched, legacy VMs [69, 63, 42]. However, many Flash exploits cannot be reliably detected via such static analyses. For example, many of the highest impact Flash attacks (e.g., Angler EK, detailed in §III-B) exploit use-after-free (UAF) vulnerabilities to achieve arbitrary remote code execution. Since AS is a Turing-complete language, it is impossible to statically predict whether any given call site in the binary might receive a freed object argument when the code is ultimately executed by an arbitrary receiving VM. Accurate static filtering of such attacks is therefore provably infeasible in general.

In this paper we propose and evaluate Inscription, the first Flash defense that automatically transforms and secures untrusted AS binaries in-flight against major Flash Player VM exploits without requiring any updates or patches of VMs or web browsers. Inscription works by modifying incoming Flash binaries with extra security programming that self-checks

[1]The work reported herein was performed while at UNC Charlotte.

against known VM exploits as the modified binary executes. Flash apps modified by Inscription are therefore self-securing. This hybrid static-dynamic approach affords Inscription significantly greater enforcement power and precision relative to static filters.

Inscription conservatively assumes that untrusted Flash binaries might be completely malicious. The extra security programming it adds therefore resides within potentially hostile scripts. Inscription must therefore carefully protect itself against tampering or circumvention by the surrounding script code. Moreover, we assume that all implementation details of Inscription might be known by adversaries in advance of preparing their attacks. Inscription therefore modifies and replaces all potentially dangerous script operations in each binary to ensure that its security checks cannot be bypassed even by knowledgable adversaries who are aware of the defense.

Our binary transformation algorithm is implemented as a web script. This allows web page publishers and ad networks to protect their end-users from malicious third-party scripts (e.g., malvertisements) that may get dynamically loaded and embedded into served pages on the client side, even when end-users are potentially running legacy, unpatched browsers and VMs. To do so, page publishers simply include Inscription's binary rewriting script on their served pages, or ad networks make the script part of their ad-loading stubs. When the page is viewed, the included script dynamically analyzes and secures all incoming Flash scripts on the client side before rendering them. We consider this deployment model to be a compelling one, since publishers and ad networks are often strongly motivated to protect their end-users from attacks (to avoid reputation loss, and therefore loss of visitors), but are rarely willing to go so far as to withhold potentially dangerous services (e.g., third-party ads) from clients running outdated software. Inscription affords publishers the former without sacrificing the latter.

Our approach expands upon prior works that have leveraged Flash app binary modification to customize apps [45] or enforce custom security policies [55, 38]. Inscription is the first work to innovate code transformations that can secure apps against exploits of major, real-world VM vulnerability classes. That is, it is the first such work to consider the underlying VM as not fully trusted. By introspectively determining which VM version is running and limiting its security guard implementation to operations known to be reliable for that version, it can secure known unsafe operations with safe replacements.

In particular, our main contributions are as follows:

- We present the design and implementation of Inscription, the first fully automated Flash code binary transformation system that can guard major Flash vulnerability categories without modifying vulnerable Flash VMs.
- Our experiences reveal that many Flash vulnerabilities can be addressed via two complementary binary transformation approaches: (a) direct *monitor in-lining* as bytecode instructions, and (b) binary *class-wrapping*.
- Inscription is the first bytecode transformation approach that assumes that the underlying VM might not be fully trustworthy; it thus strategically chooses security guards that avoid known, unpatched VM vulnerabilities.
- We discuss detailed case-studies and mitigate five major vulnerability categories of Flash exploits currently being observed in the wild.[1]
- We develop a novel memory management layer that prevents major classes of ActionScript use-after-free and double-free vulnerabilities. Inscription is able to defend against the recent (February 2, 2018) zero-day attack campaign targeting South Korean citizens [25] (§III-B).

The rest of the paper is organized as follows. Section II describes our technical approach, including an overview and implementation details of Inscription. Section III presents case studies of three vulnerability classes, including proof-of-concept malvertisement apps with full exploits and corresponding defense solutions. Section IV outlines experimental results, and Section V discusses the security analysis and approach limitations. Sections VI and VII outline related and future work, respectively.

II. Technical Approach

A. Overview

At a high level, Inscription automatically (1) disassembles and analyzes binary Flash programs prior to execution, (2) *instruments* them by augmenting them with extra binary operations that implement runtime security checks, and (3) re-assembles and packages the modified code as a new, security-hardened Shockwave Flash (SWF) binary. This secured binary is self-monitoring, and can therefore be safely executed on older versions of Flash Player that lack the latest security patches.

Inscription conservatively assumes that Flash programs and their authors have full knowledge of the IRM implementation, and may therefore implement malicious SWF code that attempts to resist or circumvent the IRM instrumentation process. Inscription thwarts such attacks via a *last writer wins* principle: Any potentially unsafe binary code that might circumvent the IRM enforcement at runtime is automatically replaced with safe (but otherwise behaviorally equivalent) code during the instrumentation. Thus, since the binary rewriter is the last to write to the code before it executes, its security controls dominate and constrain all untrusted control-flows.

Thwarting many VM exploits requires enforcement of *stateful* policies, which constrain program operations based on the history of operations that have come before. For example, thwarting UAF attacks entails suppressing method invocations on objects that have previously been freed. To enforce such policies, Inscription injects, maintains, and protects new program variables, called *reified security state variables* [60], which explicitly track the security state of security-relevant objects, values, and the overall program at runtime. *Guard code* in-lined by Inscription consults and updates these variables at security-relevant operations to check for impending policy

[1]For brevity we discuss four vulnerability categories in the paper. Please consult the companion tech report for the fifth category, viz., *Heap Spray* [61]

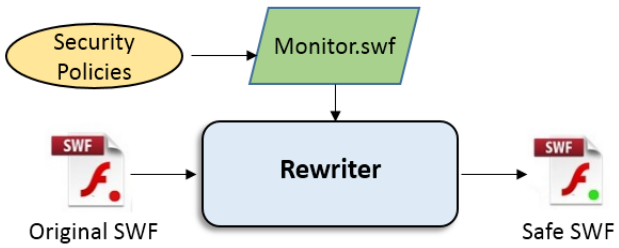

Fig. 1: Inscription IRM instrumentation architecture

violations and take corrective action if necessary. Corrective actions include premature termination, event suppression, and logging of event information.

Inscription's guard code insertions come in two major forms: (a) direct insertion or replacement of bytecode instructions at sites of potentially security-relevant program operations, and (b) substitution of potentially abusable classes with *wrapper* classes through a `package`. The latter is particularly useful when Inscription cannot statically predict where a security-relevant object may flow at runtime—a common limitation of purely static analyses. By preemptively substituting such objects with more secure variants at their creation points, Inscription can effectively track the object's flow dynamically, and intervene if it is later used improperly.

Our threat model includes exploits of known vulnerabilities in AVM2 and Flash-based libraries, and undiscovered (*zero-day*) UAF vulnerabilities.

While Inscription can protect against simple static attacks that use syntactically malformed SWF files to exploit VM parser bugs (e.g., [42, 10, 11]), we here restrict our attention to more sophisticated attacks consisting of syntactically legal SWFs that dynamically exploit VM vulnerabilities, since these are the ones not reliably detectable by standard network-level filters.

B. Implementation

Fig. 1 illustrates Inscription's SWF modification process as an in-lined reference monitoring architecture. Security policies consisting of code that evades or blocks known VM vulnerabilities are first pre-compiled to a binary `Monitor.swf` library. Inscription's rewriter statically analyzes incoming SWF files for potentially unsafe operations, and in-lines code from the `Monitor.swf` library to dynamically secure these operations at runtime. The pre-compilation step occurs once, prior to deployment; thus, Inscription can be deployed to client-side environments that lack a full AS compiler.

We use the AS Bytecode (ABC) Extractor from the Robust ABC [Dis]-Assembler (RABCDAsm) tool kit [52] to extract bytecode components [4] from the original, untrusted SWF. A Java ABC parser converts the untrusted bytecode into Java structures, according to the AS 3.0 bytecode file format specification [2]. The rewriter core, also written in Java, performs a linear search of the untrusted code to locate potentially security-relevant code points (defined by the policy). Typically such code points constitute a small percentage of the entire untrusted code. The rewriter subsequently rewrites the untrusted bytecode, inserting reified state variables, state updates, and other guard code directly as ABC instructions into the Java structures. Post-rewriting, a Java code-generator converts the instrumented Java structures back into ABC format. Finally, the RABCDAsm ABC Injector [52] re-packages the modified bytecode with the original SWF data to produce a new, safe SWF file.

When static analysis cannot reliably predict all potentially dangerous instructions where a security-relevant object might flow at runtime, Inscription replaces the instructions that create the object with instructions that instead create a corresponding wrapper object implemented by the `Monitor.swf` library. The wrapper object implements all the original object's methods, but with extra security guard code that allows Inscription to retake control if the object is subsequently abused. This affords the enforcement a means of uncircumventable, complete mediation over the wrapped object without the need to statically predict all its flows.

Inscription's rewriter ensures that all invocations of the vulnerable class (including object instantiations and method calls) in the original SWF are replaced by our new safe wrapper for the class. This is achieved by maintaining a hash-map that maps the package name of the vulnerable class to the package name of our wrapper class. When merging the monitor package with the untrusted SWF, our rewriter scans the untrusted SWF's bytecode for all occurrences of the vulnerable class' package name and replaces them with the mapped package name of our wrapper class. Please see §V for a detailed security analysis of this rewriting technique.

III. Case Studies

To more explicitly demonstrate our technical approach, we here present an in-depth analysis of five vulnerability classes, exemplified by six case studies[2] with proof-of-concept exploits and Inscription's IRM defense for each. These six detailed case studies evidence the generality of our approach by showcasing reference monitor in-lining strategies that suffice to close many other difficult, real-world, dangerous vulnerabilities. For example, our class-wrapping approach for mitigating `valueof()` exploits (see §III-B) directly mitigates almost 20 other reported vulnerabilities in the literature.

A. Double-Free Attacks and Defense

An important class of web attacks exploit double-free (DF) vulnerabilities in various legacy versions of the Flash Player. DF exploits corrupt the VM's memory management data structures, giving the exploiting Flash app a pair of corrupt object references that the VM believes are distinct at the bytecode level, but that actually refer to the same storage location. By giving the objects different types, the app can write to data fields of one object that reside atop method pointers of the other, affording the malicious app arbitrary remote code execution capabilities when it calls the corrupted method pointers.

[2]Finer details about the case studies and Inscription's implementation have been documented in a technical report [61].

```
1  bShared = new ByteArray();
2  bgWorker = WorkerDomain.current.createWorker(swfBytes);
3  bgWorker.setSharedProperty("byteArray", bShared);
4  var ib:uint = 0;
5  var b:ByteArray = null;
6  var a:Array = new Array();
7  for (k=4; k<0x3000; k+=4) {
8    bShared.writeBytes(tempBytes);
9    bShared.length = 0x400;
10   b = new ByteArray();
11   b.length = 0x30;
12   b[8] = ib;
13   a.push(b);
14   ib++;
15   b = new ByteArray();
16   b.length = 0x30;
17   b[8] = ib;
18   a.push(b);
19   ib++;
20 }
21 for (k=0; k<a.length; k++) {
22   b = a[k];
23   if (b[8] != (k%0x100)) {
24     a[k+1].length = 0x1000;
25     v.length = vLength;
26     b.position = 0;
27     b.writeUnsignedInt(0x41414141);
28     a[k-1].length = 0x1000;
29     var l:uint = 0x40000000-1;
30   }
31 }
```

Listing 1: Primary `Worker` writing to `ByteArray bShared`

```
1  function playWithWorker(){
2    for (j=0; j<0x1000; j++) {
3      bShared.writeObject(tempBytes);
4      bShared.clear();
5      bShared.length = 0x30;
6    }
7    mutex.unlock();
8    Worker.current.terminate();
9  }
```

Listing 2: Background `Worker` writing to and clearing `ByteArray bShared`

Inscription blocks DF exploit attempts by adding an extra layer of memory management at the bytecode layer to detect and suppress free operations that target already-freed objects. This extra layer is implemented as an in-lined addition to the app's bytecode, making the app self-securing, and shielding the underlying VM from potentially dangerous double-free requests that vulnerable VMs fail to catch. The bytecode implementation leverages synchronization primitives that are known to be reliable on all versions of the Flash Player, making it version-independent.

Case Study #1: To demonstrate our DF defense, we here show how it protects against exploits of CVE-2015-0359, which is listed among Kaspersky's Devil's Dozen highest impact exploits [19]. CVE-2015-0359 features in numerous exploit kits, including Flash EK, Sweet Orange, Fiesta, Angler, and Neutrino. Adobe gives it a *critical* classification, warning that it affects all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [5]. IBM X-Force Exchange rates it 9.3 out of 10 on their base score, marking its impact on confidentiality, integrity, and availability as *complete* [29].

Listings 1 and 2 show code for a primary worker and background worker thread (respectively), that together implement the attack by concurrently operating on a shared `ByteArray` object, `bShared`. The vulnerability is the result of a race condition in Flash workers, triggered by abusing the `length` property and the `writeObject()` and `clear()` methods of `ByteArray`.

```
1  package Monitor {
2   public class ByteArray extends flash.utils.ByteArray{
3    public static const hashtable:flash.utils.Dictionary;
4    public var orig_byteArray:flash.utils.ByteArray;
5    public function ByteArray() {
6      super();
7      hashtable[this] = 0;
8      orig_byteArray = this;
9    }
10   public function clear():void {
11     if(Monitor.ByteArray.hashtable[this] == 0) {
12       Monitor.ByteArray.hashtable[this] = null;
13       super.clear();
14     }
15   }
16   public function valueOf():flash.utils.ByteArray {
17    return this.orig_byteArray;
18   }
19  }
20 }
```

Listing 3: `ByteArray` safe wrapper class

Each modification to the shared array's `length` field (line 9 of Listing 1, and line 5 of Listing 2) prompts the VM to reallocate (and possibly move) the storage space assigned to the array. The two concurrent loops create a race condition in which the background worker frees `bShared` (line 4 of Listing 2) between the events of freeing and reallocating `bShared` (line 9 of Listing 1) inside the primary worker. This race causes `bShared` to be freed twice.

To determine whether the double-free was triggered, each loop iteration allocates a new `ByteArray` twice to the same variable `b` (lines 10 and 15 of Listing 1).

The attacker then assigns an index at the ninth element of `b` and pushes them one by one onto an array `a` (lines 13 and 18 of Listing 1). The attacker keeps track of the index to be assigned to the next allocation of `b` using a sequential counter `ib` (lines 12 and 17 of Listing 1). If the attack succeeds, the second allocation of `b` overwrites the first allocation.

To determine the iteration of the loop where the exploit succeeded, the attacker scans the index of every `ByteArray b` allocated inside `a` (lines 23–30 of Listing 1). If two allocations of `b` have the same index, it implies that the missing index was overwritten by the instance of `b` that allocated to the same memory chunk. This gives the attacker access to a pointer to control the heap and inject shellcode via `b`.

Inscription's defense against DF-attacks in-lines bytecode that independently double-checks that each `ByteArray` object is cleared at most once. It does so by introducing the wrapper class defined in Listing 3, which augments the app with a global, thread-safe hash table that explicitly tracks object-frees. In particular, Inscription's pre-compilation phase first creates a wrapper for the `ByteArray` class, extending it, and thereby inheriting all functionality of the original class. The wrapper class adds a static `Dictionary` object that uses objects as keys and non-null integers as values. Declaring the dictionary to be static makes it a fixed, global object shared across all workers. Locating the hash table within our private `Monitor`

package namespace prevents any hostile, surrounding app code from accessing it to corrupt its data.

To make our implementation thread-safe, we introduce a lock for our dictionary in the form of a 1-integer, shareable `ByteArray`. Inscription-instrumented threads always acquire the lock to make updates on the dictionary and subsequently release the lock. For brevity and simplicity of the presentation, we only show single-threaded code listings in this paper; however, our actual implementation maintains thread-safe synchronization.

We override the `ByteArray` constructor inside the wrapper class, so that whenever a new `ByteArray` object is created, an entry for it is added to the global hash table (lines 5–9). Our overridden `clear()` method (lines 10–15) only allows a `ByteArray` to be freed if its value in the hash table is non-zero (implying it has not been freed already). Our monitor then sets it to null before safely calling the free property of the `ByteArray` class. However, if the value stored in the hash-table is zero or null, then our monitor suppresses the free operation, which prevents the DF. Additional synchronization code (not shown) prevents these methods from executing concurrently.

Inscription's rewriter then merges our monitor containing the wrapper class with the untrusted SWF so that every call to `ByteArray`() and `ByteArray.clear()` is replaced by our overridden methods. After instrumentation of this IRM code, the rewritten safe SWF is produced.

Case Study #2: In 2014, FireEye and Adobe identified a targeted attack campaign, Operation GreedyWonk [9], exploiting a zero-day DF Flash vulnerability that was later recorded as CVE-2014-0502. The vulnerability permits the attacker to overwrite a Flash object pointer to alter the flow of code execution on Windows XP and 7 machines. In this section, we present the analysis of this vulnerability, a proof-of-concept attack, and our IRM defense strategy. Our discussion closely follows Ben Hayak's vulnerability description in the SpiderLabs blog [27].

CVE-2014-0502 is a DF vulnerability caused by the AVM's mis-handling of `SharedObjects`. While `SharedObjects` can be explicitly flushed to disk using the `flush()` method, all `SharedObjects` belonging to a `Worker` thread are also implicitly flushed when a `Worker` terminates.

Before flushing, each `SharedObject`'s destructor performs two checks: (1) check the object's *pending flush* flag, which indicates whether there is data in the `SharedObject` that must be flushed to disk, and (2) check the maximum allowed storage settings for the domain. If the flag is set and flushing would not exceed the domain's storage allowance, then it is flushed to disk and its flag is reset. If there is insufficient storage, the flush operation does not succeed and the flag is not reset.

Unfortunately, older versions of the AVM fail to properly synchronize these operations, allowing multiple flushes to proceed concurrently, exposing a DF exploit opportunity.

The attacker first creates a `SharedObject` that would exceed the storage limit if flushed (Listing4 lines 3–9). Just before the destructor called by `Worker.terminate()` frees the object (line 10), the AVM sees that the object is available

```
1  public class WorkerClass extends Sprite {
2    public static var G:Worker = new Worker();
3    public function increaseSize():void {
4      var exp:String ="AAAA";
5      while ((exp.length<102400))
6        exp=(exp + exp);
7      var sobj:SharedObject= SharedObject.getLocal("record")
           ;
8      sobj.data.logs=exp;
9    }
10   Worker.current.terminate();
11 }
```

Listing 4: Triggering a `SharedObject` double-free

```
1 var sobj : SharedObject = SharedObject.getLocal("record");
2 if (current_size + o.length < max_size) {
3   current_size += o.length;
4   current_size.flush;
5   sobj.data.logs = o;
6 }
```

Listing 5: IRM guard code for SharedObject writes

for garbage collection, overlooks the ongoing destruct, and calls the destructor again. Both destructors hit the size limit, leave the flag set, and free the object, resulting in a DF.

To thwart such attacks, Inscription enforces the storage limit preemptively—before the objects undergo flush attempts. This ensures that the AVM's pending flush flag always gets reset during flushes, ensuring proper synchronization of concurrent flushes on legacy VMs. Specifically, Inscription in-lines bytecode that tracks a running sum (the IRM's reified security state) of all writes to `SharedObjects` in each domain. Writes that would exceed the storage limit are suppressed.

To enforce this policy, the bytecode rewriter injects a thread-safe, global, static variable of type `SharedObject`, which counts the total size of all `SharedObjects` belonging to the web domain. Using a `SharedObject` as the counter allows it to access all other `SharedObjects` across the domain even if there are multiple SWFs.

The rewriter scans the application's bytecode to identify all operations where a `SharedObject` is created or updated, and inserts guard code that tests and updates the counter, as shown in Listing 5. In particular, before each write to object *o* (line 5), the current size for the domain is synchronized, tested, and updated (lines 1–3). Since the counter is also a `SharedObject`, we explicitly flush it to the disk (line 4) so that it remains updated across SWFs. Additional synchronization bytecode (not shown) ensures that these operations are atomic. Listing 5 shows the bytecode instrumentation at the source level for clarity, but the actual instrumentation is done directly at the bytecode level.

CVE-2014-0574, CVE-2015-0312 and CVE-2015-0346 are similar vulnerabilities that Inscription can prevent from being exploited.

B. Use-After-Free Attacks and Defense

Another large class of web attacks exploit UAF vulnerabilities in various legacy versions of the Flash Player [15, 8, 46, **?**]. UAFs afford attackers similar hijacking opportunities to DFs (see §III-A). By retaining a dangling pointer to an object that has been freed by the AVM's memory manager, a malicious app can contrive to allocate a new object of different type atop

TABLE I: AS classes, methods, and properties used in ApplicationDomain UAF example

Name (Type)	Description
SecurityDomain (class)	Represents the security sandbox for the web domain from which the SWF application was loaded.
ApplicationDomain (class)	Allows for partitioning of AS classes within same security domain into containers (smaller sandboxes). AS allows loading an external SWF into an existing SWF's source. ApplicationDomain is used to create a separate container for classes of the external loaded SWF.
currentDomain (property)	Read-only property of ApplicationDomain, the class that gives the current application domain in which the code is executing.
ByteArray (class)	Allows for reading/writing of raw binary data.
domainMemory (property)	A property of the ApplicationDomain class that can be set to a ByteArray object for faster read/write access to memory [14].
Worker (class)	Allows creation of virtual instances of the Flash Runtime; this is how AS implements concurrency.

the vacated storage space. As with DFs, this allows writes to the data fields of one object to corrupt method pointers of the other object stored at the same location, resulting in arbitrary remote code execution by malicious apps.

Although UAFs and DFs offer similar attack opportunities, mitigation of UAFs requires a substantially different IRM enforcement approach relative to DFs. This is because the security-violating operation that facilitates the attack is retention of a dangling pointer, which is not detectable merely by monitoring object allocations and deallocations. To thwart UAF exploits, Inscription therefore extends its bytecode-level memory management layer presented in §III-A with an additional object alias tracking capability. This allows the memory manager to suppress attempted frees of objects to which other threads retain references, shielding vulnerable VMs from dangerous frees that could result in a UAF.

Most Flash UAF vulnerabilities arise due to the AVM's mismanagement of objects that are mutable, can change size at run time, have explicit `clear` or `flush` operations, which ideally should free all assignments of the object from the memory and prevent dangling pointers, or are nullified (`null`) while being subscribed by some other object.

To secure such operations on potentially vulnerable AVMs, Inscription implements an extra layer of memory management at the bytecode level within the app code. The extra layer consists of wrappers that interrupt the creation of such objects, remembering them in a secure `Dictionary` (`HashTable`). The rewriter exhaustively scans the code to find all assignments that might reference these objects, and maintains an explicit reference count for each in the `Dictionary`. Explicit `clear`, `flush` and `null` operations in the app that reference these objects are replaced with bytecode that checks the `Dictionary` for dangling references before clearing the object from memory. This blocks many AVM-level UAF exploits.

To cover as many potentially vulnerable operations as possible with our defense (even operations for which no specific AVM vulnerability is yet known by defenders), we built a crawler that parses all the AS3 API documentation (web pages) to find all APIs that expose explicit free operations to apps, and wrapped all such operations in untrusted apps. This potentially generalizes Inscription's defense to zero-day exploits. For example, our crawler helped in introducing protections for year-2015 CVEs 0313, 5119, 0311, 3128, 5122, 5561, 7652, 8044, 8046, 8049, 8050, 8140, and 8413, without any explicit prior knowledge of any of these CVEs.

For example, Inscription is able to prevent the zero-day exploit of CVE-2018-4878 discovered on Feb 2, 2018 [25], even though our defense implementation predates the discovery of that vulnerability. The vulnerability is triggered by nullifying the `DRMOperationCompleteListener` [7] event listener object subscribed by a `MediaPlayer` instance, causing the AVM to prematurely free the object. Our IRM defense automatically delays such nullifications until the `MediaPlayer` has been notified, blocking the UAF.

Case Study #3: The Angler EK exploit mentioned in §I is a prominent UAF example. The exploit targets two UAF vulnerabilities (CVE-2015-0311 and CVE-2015-0313) in Flash's `ApplicationDomain` class. These two vulnerabilities were extremely popular among exploit kit writers in 2015, and were a part of Kaspersky's Devil's Dozen [19]. They allow remote attackers to execute arbitrary code via multiple attack vectors on Windows, OS X, and Linux machines. We created a proof-of-concept SWF ad that exploits CVE-2015-0313 to conduct the attack and defense.

We first outline the exploit [70], and then discuss Inscription's mitigation. Table I describes the AS3 [1] classes, methods and properties used in this example.

The Angler EK exploit implements a malicious SWF file containing one primary worker and one background worker thread that share a `ByteArray` object set to the `domainMemory` property. In the attack, the background worker frees the shared `ByteArray` object; however, the primary worker can still reference it. This inconsistency results in a UAF vulnerability, and gives the attacker a pointer to control the heap memory of the SWF application.

The attack is performed in three stages. First, the primary worker sets a `ByteArray` object to the `domainMemory` property and instructs the background worker to free the object. Second, the background worker frees the object upon receiving the instruction from the primary worker, despite the primary worker retaining a pointer to the freed object in memory. Third, the malicious SWF uses the dangling pointer in `domainMemory` to inject a `Vector` (an AS array of changeable size), containing shellcode corresponding to the *return-oriented programming* (ROP) [58] gadget chain of its malicious payload. Finally, the malicious SWF scans the heap for the `Vector` of the same length and writes the ROP chain and shellcode to the buffer, which then allows it to execute ROP attacks.

To mitigate this attack, Inscription implements a `SafeApplicationDomain` wrapper class that replaces `ApplicationDomain` on vulnerable VMs. The wrapper ensures that a `ByteArray` shared amongst multiple workers is never inconsistently freed. To to so, the memory manager hash table

```
1 if (hashtable[byteArray1] > hashtable[byteArray1]+1)
2   integer_overflow_error();
3 else{
4   hashtable[byteArray1]++;
5   ApplicationDomain.currentDomain.domainMemory=byteArray1;
6 }
```

Listing 6: IRM guard-code for `ByteArray` object assignment to shared `domainMemory`

described in §III-A counts the number of *subscribers* (i.e., referencing workers) for every `ByteArray` object in the untrusted SWF, instead of merely tracking frees. Our rewriter then instruments operations that assign `ByteArray` objects to the `domainMemory` property with guard code that updates the subscriber count.

Enforcing this policy leverages a combination of direct bytecode in-lining and class-wrapping. The pre-compilation phase first creates the wrapper in Listing 3 with subscriber counts as values.

The overridden `clear()` method (lines 10–15) only allows a `ByteArray` to be freed when its subscriber count reaches 0. Inscription's rewriter merges this monitor `package` into the untrusted SWF so that every call to `ByteArray()` and `ByteArray.clear()` is intercepted by our overridden methods.

To track and update subscriber counts, the rewriter must update the table whenever a `ByteArray` is assigned to a shared `domainMemory` property. This cannot be achieved by class-wrapping since the wrapper class does not have access to assignment operations outside its class. Inscription therefore applies direct bytecode instruction modification to secure such operations.

Listing 6 expresses the modified bytecode as source code (although the actual transformation is performed at the binary level). Before each security-relevant assignment (line 5), Inscription in-lines bytecode that increments its subscriber count (line 4). To thwart arithmetic overflow attacks against the counter, both operations are additionally guarded by an overflow check (lines 1–3). When the `domainMemory` shared object stops subscribing to the `byteArray1`, the IRM decrements the subscriber count (not shown here). When the subscriber count becomes 0, `byteArray1` becomes clearable again (see line 11 of Listing 3).

Case Study #4: Many UAF exploits against legacy Flash Players are rooted in a logical flaw wherein numerous AVM implementations fail to consider that the semantics of assignment operations in AS implicitly invoke the `valueOf` method of the assigned object when a type coersion is needed, and `valueOf` may be overridden by the app to perform unexpected side-effects. AVM implementations with this vulnerability fall prey to UAF attacks when the AVM fails to recheck its object pointers after the assignment completes, erroneously assuming that their referants cannot have been freed during the assignment.

Inscription blocks these attacks by introducing bytecode at sites of assignment-solicited type-coersions in order to force the coersion (and the resulting call to `valueOf`) to occur strictly before the VM begins processing the assignment. Forcing

```
1  public class malClass extends Sprite {
2   public function malClass() {
3     var b1 = new ByteArray();
4     b1.length = 12;
5     var mal = new hClass(b1);
6     b1[0] = mal;
7   }
8  }
9  public class hClass {
10  private var b2 = 0;
11  public function hClass(var b3) {
12    b2 = b3;
13  }
14  public function valueOf() {
15    b2.length = 13;
16    return 15;
17  }
18 }
```

Listing 7: Classes used in `ByteArray` UAF

the coersion early ensures that legacy VMs never attempt the coersions amidst assignments, thereby evading the vulnerability.

We illustrate this attack and defense using CVE-2015-5119, another popular vulnerability from Kaspersky's Devil's Dozen [19]. It abuses indexed assignments involving the `ByteArray` "`[]`" operator. This UAF attack was added to Angler EK, Neutrino, Hanjuan, Nuclear Pack, and Magnitude exploit kits in 2015, after it was leaked from the Hacking Team [37]. Adobe categorizes it *critical*, and warns that it affects all Flash Player versions up to 18.0.0.194 for Windows, Macintosh, and Linux [6]. IBM X-Force Exchange rates it an 8.8 out of 10 on their base score, marking its impact on confidentiality, integrity, and availability as *high* [30].

Listing 7 demonstrates the attack. A `ByteArray` object `b1` of length 12 is first created by `malclass` (lines 3–4). Next, an `hclass` object is instantiated and `b1` is assigned to its `b3` field (lines 5, 11–13). Back in `malclass`, `mal` is assigned to index 0 of `b1` using operator "`[]`" (line 6). The assignment must coerce the type of `mal` to a numeric type, so control flows to the `valueOf` function of `hclass` (line 14). As a side-effect of this function, the attacker increases the length of `ByteArray b2` (line 15), causing the AVM to free and reallocate it to a new address. However, in `malclass`, `b1[0]` still references the freed memory chunk, allowing a malicious app to substitute a differently typed object in its place and subsequently misuse its method pointers as data.

Inscription thwarts this attack by replacing line 6 with bytecode equivalent[3] to the following:

```
6 b1[0] = Number(mal);
```

This forces the numeric coersion to be explicitly processed as part of the right-hand-side expression evaluation, which the AVM performs separately before the assignment executes. The ensuing assignment then requires no additional coersion, avoiding evaluation of `valueOf` inside the AVM's assignment operator implementation.

C. Buffer Overflow Attack and Defense

Case Study #5: Another Kaspersky *Devil's Dozen* vulnerability, CVE-2015-3090, was spotted in May 2015, and has

[3]Our actual implementation is at the bytecode level, not the source level, to avoid compiler optimizations that unsafely eliminate the source-level coersion.

```
1  public function ShaderJobTOCTOU():void
2  {
3    var ba:ByteArray = new ByteArray();
4    ba.writeByte(0xa1);  // Define parameter
5    ba.writeByte(0x00);  // Empty string
6    ba.writeUnsignedInt(0x00000010);
7    ba.writeUnsignedInt(0);
8    ba.position = 0;
9    var bd:BitmapData = new BitmapData(1024, 1024);
10   var job:ShaderJob = new ShaderJob();
11   var shader:Shader = new Shader();
12   shader.byteCode = ba;
13   job.target = bd;
14   job.shader = shader;
15   job.start(false); // false means asynchronous job
16   job.height = 1025;
17 }
```

Listing 8: Triggering Buffer Overflow in `ShaderJob`

been exploited in the wild [22]. The vulnerability occurs due to the lack of a buffer overflow check in a specific part of the AVM code. As seen in Listing 8, the AS3 APIs involved in triggering the vulnerability are in the `Shader` class (line 11), used to represent a Pixel Bender shader kernel in ActionScript. Pixel Bender is an image and video processing toolkit, that has been developed by Adobe, and employs the Pixel Bender kernel language [3]. `Shader` operations can be performed in standalone mode on a target image using a `ShaderJob` instance (line 10). The target image is represented by a `BitmapData` object, as seen in line 9.

Attackers can exploit this vulnerability by creating a race condition in `ShaderJob`—increasing the width/height of a target `BitmapData` object, while performing the `Shader` operation in asynchronous mode using the `ShaderJob` object, will result in a buffer overflow. Lines 3 to 8 depict the creation of the Pixel Bender shader kernel, which is assigned to the `Shader` object in line 12. The `BitmapData` constructor takes two integer parameters, which represents and fixes its height and width as seen in line 9. The `ShaderJob` is started in asynchronous mode in line 15; subsequently, increasing the height of the target `BitmapData` object results in a buffer overflow due to a missing check in the AVM implementation of `ShaderJob`.

Inscription intercepts the assignments of the height and width properties of a `ShaderJob` object when the object is started in asynchronous mode. If the `ShaderJob` object is running, and its new height/width is going to exceed the predefined `BitmapData`'s height or width (as seen in line 16), Inscription restricts this assignment, hence, thwarting the buffer overflow.

D. Out-of-Bounds Read

Case Study #6: CVE-2015-0310, another Kaspersky's Devil's Dozen, is a *out-of-bounds read* vulnerability that was part of numerous attack campaigns during early 2015. The vulnerability resides in the AVM level implementation of the `exec` function of the AS3 regular expression API, `RegExp`. The `exec` function has a fixed-sized array of size 99 in the call stack that stores starting and ending indexes of matched strings. This array can store up to 49 matches. After completing execution, the `exec` method returns an integer as result. A positive return value indicates success and the returned value is the number of matches, a 0 indicates that the array offset is not

```
sh(?!e|((((((((((((((((((((((((((((((((((((((((((((((((((((((((((?!e|
())37)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)
|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)
|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)(?P<test>)
```

Fig. 2: Regular expression used to trigger CVE-2015-0310

```
1 if (RegExpTest.Safe_RegExp(⟨pattern⟩, ⟨flag⟩))
2   var re:RegExp = new RegExp(⟨pattern⟩, ⟨flag⟩);
```

Listing 9: Replacing `RegExp.exec` function with `Safe_RegExp` at binary level

```
1  package Monitor {
2    public class RegExpWrapper {
3      public static function
4          Safe_RegExp(pattern:String, flag:String) {
5        var left_parenthesis_counter = 0;
6        for (var i:int = 0; i < pattern.length; i++) {
7          if (pattern.charAt(i) == "(") {
8            if (++left_parenthesis_counter > 49)
9              return false;
10         }
11       }
12       return true;
13     }
14   }
15 }
```

Listing 10: Wrapper for the `RegExp` class

big enough to hold the matches, a -1 indicates that no match was found, and any lower value indicates an unexpected error. The attacker exploits this by supplying the regular expression in Fig. 2, which consists of multiple opening and closing parentheses that exceed the array's bound (49) to store matches. Therefore, the `exec` method returns 0, and the array gets filled with garbage values, allowing the attacker to access out of bounds memory offsets to execute malicious code.

Inscription provides a wrapper for the `RegExp` AS3 class which can be seen in Listing 10. At the binary level, Inscription replaces all calls to the `exec` method of the `RegExp` class with the `Safe_RegExp` function provided by the wrapper (shown in Listing 9), to investigate the pattern of the regular expression. The `Safe_RegExp` function restricts the number of open parentheses to 49, and returns returns a boolean value indicating whether the regular expression is safe to be created.

IV. Experimental Results

We created proof-of-concept exploits for each vulnerability class presented in §III in order to fully test our solution. Our proof-of-concept exploits are modeled after real-world exploit analyses and vulnerability descriptions found in popular exploit and security research archives such as Google Security Research Database [24], ExploitDB [49], KernelMode.info [34], and security blogs by research companies such as TrendMicro [65], FireEye [16] and TrustWave [66]. All ads were created using Adobe Flash Builder v4.7.

Table II summarizes our experimental results for the proof-of-concept exploits and the corresponding policies. (See companion technical report for the heap spray policy.) All experiments were conducted on a machine with a 2.5 GHz Intel Core i5 processor with 8GB RAM. The parser, rewriter, and code-generator for AS3 bytecode were written in Java using JDK v1.8.0_144. For computing the total rewriting time for each policy, we ran each policy rewriter ten times and computed the

TABLE II: Performance Benchmarks for Proof-of-Concepts Exploit Code

Case Study	Vulnerability Class	Stateful	Rewriter Type	Rewriting Time (ms)	SWF Size (bytes) Before	SWF Size (bytes) After	Execution Time (ms) Before	Execution Time (ms) After
#1 (§III-A)	DF	✓	(2)	154	3893	4266 (+9.6%)	198.9	217.4 (+9.3%)
#2 (§III-A)	DF	✓	(1)	115	1281	1374 (+7.3%)	9.0	10.4 (+15.6%)
#3 (§III-B)	UAF	✓	(1) & (2)	100	1656	1737 (+4.9%)	211.3	231.5 (+9.6%)
#4 (§III-B)	UAF		(1) & (2)	146	936	1359 (+45.2%)	30.3	32.7 (+7.9%)
#5 (§III-C)	Buffer Overflow		(1)	56	482	488 (+1.24%)	1	1 (+0.0%)
#6 (§III-D)	Out-of-Bounds Read		(1) & (2)	71	330	558 (+69.09%)	1.0	1.1 (+10.0%)
	Heap Spray	✓	(2)	133	1283	1901 (+48.2%)	1.0	1.2 (+20.0%)
	Average			**110**	**1408**	**1669 (+18.5%)**	**64.6**	**70.7 (+9.07%)**

(1) direct bytecode instrumentation, (2) wrapper class instrumentation

TABLE III: Performance Benchmarks for Benign SWFs

Filename	Size (bytes) of ABC (before)	Size (bytes) of ABC (after)	Rewriting Time (ms)
atmosenergy	708	708 (+0.0%)	50
att	21,532	22,332 (+3.71%)	141
beetle	80,707	81,534 (+1.02%)	412
CookieSetter	598	598 (+0.0%)	53
ecls	2,007	2,007 (+0.0%)	55
eco	2,007	2,007 (+0.0%)	57
expandall	2,778	2,778 (+0.0%)	58
flash_animation	2,980	2,980 (+0.0%)	59
freechat_313	2,273	2,273 (+0.0%)	55
fxcm	1,738	1,738 (+0.0%)	52
gen_live	21,784	22,622 (+3.84%)	176
gm	22,037	22,897 (+3.90%)	146
gucci	1,079	1,079 (+0.0%)	54
hma	2,364	2,364 (+0.0%)	57
iphone	1,152	1,152 (+0.0%)	65
IPLad	1,655	1,655 (+0.0%)	52
jlopez	16,655	16,655 (+0.0%)	113
men1	33,771	34,714 (+2.79%)	219
men2	40,300	41,291 (+2.45%)	256
reliant	4,731	4,731 (+0.0%)	67
t2	919	919 (+0.0%)	49
thehappening	107,548	107,548 (+0.0%)	81
utv	20,635	21,475 (+4.07%)	140
verizon_orig	2,799	2,799 (+0.0%)	80
verizon	3,305	3,305 (+0.0%)	60
verizonm2m	2,245	2,245 (+0.0%)	54
weightwatchers	3,454	3,454 (+0.0%)	58
Average	**14,954**	**15,108 (+1.015%)**	**100.7**

average. Rewriting times include the linear search performed to locate code fragments requiring instrumentation, and the inlining of security guard code and reified security state variables. However, these instrumentation times are typically negligible since only a tiny portion of most SWF files are comprised of code; the majority of the content is comprised of images, sounds, and video. Therefore, we believe that even though our experiments are on proof-of-concept exploits, rewriting times are representative of real-world apps.

Size overhead of each rewritten SWF was measured using the uncompressed size of the application bytecode before and after rewriting. Wrapper class and binary instrumentation contributes additional bytes to SWF files. These percentage size overheads will be much smaller for real-world, non-malicious SWF files (see III), since our proof-of-concept exploits are far more densely packed with dangerous code sites than typical SWFs.

Table III summarizes performance benchmarks of evaluating Inscription with benign SWFs, using the `[]`-operator rewriter. We chose to use this policy rewriter since `[]` is the most frequently occurring policy-relevant instruction (out of our five policies), and therefore represents the worst case scenario in terms of number of instrumentations needed and rewriting time.

V. Discussion

A. Security Analysis

As explained in §II-A, Inscription IRMs maintain self-integrity and complete mediation within potentially hostile script environments based on a "last writer wins" principle: By modifying the untrusted bytecode before it executes, Inscription can automatically replace any potentially unsafe binary code that might circumvent the IRM enforcement with safe code during the instrumentation. Thus, since Inscription's rewriter is the last to write to the file before it executes, its security controls dominate and constrain all untrusted control-flows.

Our approach can be applied both to protect against many attacks falling within a general attack class (e.g., large classes of UAF and DF attacks), and also to protect against specific attacks that do not fall within a generalizable class (e.g., the regexp vulnerability discussed in §III-D).

All wrapper classes are implemented as `final` classes in a dedicated namespace (i.e., `Monitor`), allowing AS's object encapsulation and type-safety to prevent untrusted code from directly accessing the private members of wrapper classes. The bytecode rewriter then modifies the metadata of the untrusted SWF to change all references to wrapped classes to instead reference the corresponding wrapper classes. This ensures that the untrusted SWF uses the safe functions provided by our `Monitor` class instead of using unsafe functions in the untrusted class, thereby providing complete mediation.

Flash apps cannot directly self-modify (except by first exploiting a VM bug, which we prevent), but they can dynamically generate and execute new bytecode via a select collection of system API methods (e.g., `Loader.loadBytes`). Inscription wraps these methods with bytecode that recursively applies the code rewriting algorithm to dynamically generated code before it executes. Likewise, system API methods that allow AS code to dynamically generate class references from strings (e.g., `flash.utils.getDefinitionByName`) are wrapped with bytecode that substitutes the resulting reference with one to a wrapper class if the class is wrapped. This prevents untrusted AS code from acquiring unmediated access to vulnerable classes even by reflective programming.

In direct bytecode rewriting, Inscription's bytecode rewriter scans the untrusted code for every occurrence of the vulnerable method and injects guard-code surrounding it. AS type-safety guarantees that checks in the guard-code are not circumvented. For policies that use wrapper classes, Inscription's `SWF` merge tool replaces every binary occurrence of the vulnerable method call in the untrusted `SWF` file with the corresponding overridden method of the wrapper class instead.

B. Attack and Defense Design Challenges

While our overall approach is general enough to mitigate many different VM vulnerabilities and vulnerability classes (specifically, any computable safety policy [26] and some non-safety policies [40]), formulating sound and efficient policy implementations can sometimes require a detailed understanding of VM internals, including known bugs. All vulnerabilities described in this paper were results of subtle inconsistencies in the complex AS language semantics or obscure security flaws deep inside the AVM. To formulate appropriate policies, we therefore performed extensive background research and experiments, since the AVM2 is not open source. Additionally, a thorough knowledge of all AS 3.0 classes and their properties involved in the vulnerabilities and exploits was required to create policies to mitigate further attacks.

Testing the resulting defenses can also be challenging. Some vulnerabilities require a very specific environment in order to be triggered; for example, the `ByteArray` DF studied in §III-A targets SWF version 25 specifically. Many synchronization vulnerabilities abuse `Worker`s, but neither of Adobe's Creative Suite tools for Flash development (Animate CC or Flash Builder 4.7) have tracing or debugging for background `Worker`s. To test our policies, we therefore manually created proof-of-concept ads with full exploits by stitching the exploits from code snippets and relevant information dispersed among a broad array of threat reporting sources.

To the best of our knowledge, there are currently no commercially available libraries or tools for AS bytecode manipulation. Complicating this problem, the SWF binary format specification is open-ended in the sense that SWFs may include binary sections with proprietary or otherwise undocumented content tags; Flash players simply ignore sections with tags they do not recognize. This unfortunately tasks security tools with the daunting challenge of recognizing and analyzing all possible tags (even undocumented ones) recognized by all players in order to secure all malicious content. To develop Inscription, we therefore pieced together scattered information about many different players, AS compilers, and AS parsers, to support as many SWFs as possible. While we cannot ensure that our efforts are fully comprehensive, we successfully tested our prototype on a large number of ads currently distributed by major ad networks to assess its completeness.

C. Deployment

We conservatively assume that most users update their web-browsers and Flash Players only sporadically, which allows their systems to be compromised by exploits targeting vulnerabilities that were recently patched.

Our work targets malicious SWFs delivered by exploit kits and malicious third-party content (e.g., malvertisements) loaded by second-party content (e.g., web pages). Second-parties do not serve the malicious content directly, so cannot rewrite the Flash files on their servers. But the loader scripts that they serve to end-users do see and have the opportunity to rewrite all dynamically loaded content, including content loaded through redirections to malicious servers. Our work therefore provides a means for trustworthy second-parties to protect their end-users from malicious third-party content by embedding Flash rewriting logic into their loader scripts. This does not entail updating the end-user's client, which second-parties generally cannot do. Third-party malicious content dynamically embedded into otherwise trustworthy second-party content is one of the most common web attack patterns highlighted in major threat reports today, motivating this as a potentially high-impact deployment model.

D. Limitations

While our high-level approach can apply to AVM1 vulnerabilities, our current prototype implementation does not yet support them. AVM1 runs AS 1.0 and 2.0 which are very different from AS 3.0, requiring a different parser and rewriter.

Inscription cannot stop malicious events generated within externally loaded files. For example, in CVE-2016-0967, loading an external `.flv` file corrupts the stack [23]. However, we do not analyze or instrument the external file before loading; therefore our IRM cannot protect against it. In SWF binaries, externally loaded files can be written in languages other than AS (e.g., JS). Protecting against such attacks should therefore combine Inscription with appropriate defenses for those other languages.

VI. Related Work

In-lined Reference Monitoring for ActionScript Bytecode: Prior work has established theoretical foundations for secure in-lined reference monitoring of type-safe bytecode languages like Flash/ActionScript [60, 62]. These works propose certification algorithms for proving soundness (instrumented code satisfies a given security policy) and transparency (instrumentation process does not alter the behavior of safe programs) properties of IRMs. Inscription adopts an IRM approach to protect unpatched Flash VMs from real-world exploits. Future work should therefore consider applying machine-certification based on these prior works to achieve formal guarantees for the policies enforced by Inscription.

FlashJaX [55] is an IRM solution for cross-platform web content spanning Flash and JavaScript. It primarily targets attacks that abuse inconsistencies between the security models of the two languages (e.g., differences between the same-origin policies enforced by the Flash and JS VMs). In contrast, Inscription targets direct exploits of legacy Flash VMs, which are currently the highest-impact targets of in-the-wild web exploit kits (see §I).

FIRM [38] presents an IRM approach for mediating the interaction between Flash and the DOM using *capability tokens*. Each SWF is assigned a unique capability token which is associated with a set of policies to be enforced on the SWF. FIRM instruments the SWF with wrappers that guard functions that interact with DOM objects; additionally, FIRM wraps certain security-sensitive DOM objects' getters and setters. The SWF wrappers work in sync with the DOM wrappers to allow or deny function calls based on the capability tokens. Inscription targets vulnerabilities arising out of security flaws inside the AVM, which FIRM cannot enforce.

Mitigations for Specific Flash Security Issues: The *extended same-origin policy* (eSOP) [31] mitigates Flash-based DNS rebinding attacks by adding a `server-origin` component to the browser's same-origin policy. The `server-origin` is explicit information provided by the server concerning its trust boundaries; any mismatch between `domain` and `server-origin` stops the attack.

Copious benign usage of URL redirection in Flash ads misleads security tools to produce false negatives for truly malicious URL redirects in Flash plug-ins. Related work monitors plug-ins instead of SWFs to reduce this false negative rate [64]. Spiders also identify malicious Flash URL redirects [36].

HadROP [53] utilizes machine learning to mitigate (Flash) ROP attacks. Differences in micro-architectural events between conventional and malicious programs are used for detection. In another related work, static and dynamic analyses are used to extract features of a SWF for feeding into a *deep learning* [57] tool for anomaly-based Flash malware detection [32].

GORDON [68] uses structural and control-flow analyses of SWFs and machine-learning to detect the presence of malware. However, GORDON has been implemented on Flash's open source implementations, Gnash [21] and LightSpark [41]. FlashDetect [51] extends OdoSwiff [17] to ActionScript 3.0. It dynamically analyzes SWF files using an instrumented version of Lightspark [41] Flash player to save traces of security relevant events. It then performs static analysis on AS3 bytecode to identify common vulnerabilities and exploitation techniques.

Ad Network Infrastructure and Malvertising: Web advertising is a billion-dollar business that attracts attackers due to its unsolicited purveyance of executable script code to many users. Significant effort is therefore invested by both academia and industry to identify and remove malicious advertisements from ad networks. Unfortunately, malvertisers often use sophisticated techniques to evade being detected by traditional malicious content inspectors. For example, the ubiquitous practice of *ad syndication* is often abused by malvertisers to flow their malicious creations through many levels of indirection to end-user browsers without ever submitting them to a reputable ad provider for vetting [39].

A recent study of over 600,000 ads [71] concludes that malvertising attacks typically come in three forms: *drive-by downloads* exploit browser vulnerabilities to do damage, *deceptive downloads* lure human victims into installing malware, and *link hijackings* redirect users to malicious web sites. Inscription primarily defends against the first of these categories. While server-side filters employed by ad networks try to defend against such attacks, the study concludes that drive-by downloads nevertheless regularly reach end-users. This is in part because modern ad delivery systems transmit ads directly from advertiser servers to end-user browsers, preventing the ad networks from verifying that the ads received by users are the same ones that passed their filtering process. Inscription avoids this difficulty by securing the received ads on the client-side.

A recent evaluation of commercial malvertising detection tools based on blacklisting and signature-based detection concluded that such tools detect only about 73% of malicious content in practice [43]. These approaches consult and maintain public databases where reputation of websites and domain blacklists are stored, and compute a trustworthiness score for redirected URLs and executables. If the trustworthiness score is below a threshold, the tools raise alarm indicating the content is malicious. In contrast, Inscription directly secures untrustworthy script binaries without relying on reputations. It therefore does not need to determine whether the original content was malicious in order to be effective.

VII. Conclusion

We have presented the design and implementation of Inscription, a fully automated Flash code binary transformation system that can guard major Flash vulnerability categories without modifying vulnerable Flash VMs. We demonstrated two complementary binary transformation approaches, direct monitor in-lining as bytecode instructions and binary class-wrapping, for flexible and precise instrumentation. In detailed case-studies, we describe proof-of-concept exploits and mitigation strategies for five major Flash vulnerability categories.

In future work, we plan to fit Inscription into certification systems for IRM soundness and transparency [60, 62]. We also plan to extend the developed technology and security mechanisms to other similar ECMAScript languages and platforms, and to extend Inscription to handle malicious events generated in externally loaded files inside a SWF as well.

Acknowledgements

This research was supported by NSF CRII award #1566321, NSF CAREER Award #1054629, NSF Collaborative Research Award #1513704, AFOSR Award FA9550-14-1-0173, and ONR Awards N00014-14-1-0030 and N00014-17-1-2995.

References

[1] Adobe. ActionScript® 3.0 reference for the Adobe® Flash® platform. http://tinyurl.com/hx5w647.
[2] Adobe. ActionScript virtual machine 2 (AVM2) overview. https://tinyurl.com/yaynm9u4, May 2007.
[3] Adobe. Pixel Bender reference. https://tinyurl.com/yakzqfux, 2009.
[4] Adobe. SWF file format specification version 19. https://tinyurl.com/yca8fw8e, 2012.
[5] Adobe. Security updates available for Adobe Flash Player: APSB15-06. http://tinyurl.com/lphfr6v, April 2015.
[6] Adobe. Security updates available for Adobe Flash Player: APSB15-16. http://tinyurl.com/ofdwo9c, July 2015.
[7] Adobe. Adobe Primetime TVSDK 1.4 for desktop HLS programmer's guide. https://tinyurl.com/ycr7tq8m, 2018.
[8] N. Cano. Adobe Flash zero day vulnerability exposed to public. *Bromium*, July 2015. https://tinyurl.com/ya8nlpww.

[9] D. Caselden, J. Weedon, X. Chen, M. Scott, and N. Moran. Operation GreedyWonk: Multiple economic and foreign policy sites compromised, serving up Flash zero-day exploit. *FireEye Threat Research*, February 2014. http://tinyurl.com/mps9ltp.

[10] Check Point Software Technologies. Adobe Flash Player memory corruption (APSB15-04: CVE-2015-0318). https://tinyurl.com/ybkfj967.

[11] Check Point Software Technologies. Adobe Flash Player memory corruption (APSB16-39: CVE-2016-7874). https://tinyurl.com/y9saxfbe.

[12] Cisco. 2016 midyear security report. https://tinyurl.com/y7kupmkr, 2016.

[13] CVE. Top 50 Products By Total Number Of "Distinct" Vulnerabilities in 2016. https://www.cvedetails.com/top-50-products.php?year=2016.

[14] P. Dolla. Faster byte array operations with ASC2. http://tinyurl.com/j6kccdl, August 2013.

[15] F. Falcón. Exploiting CVE-2015-0311: A use-after-free in Adobe Flash Player. *CoreLabs Security*, March 2015. https://tinyurl.com/ycx67vvw.

[16] FireEye. Threat research and analysis. https://www.fireeye.com/blog.html.

[17] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Analyzing and detecting malicious Flash advertisements. In *Proc. of the 25th Annual Computer Security Applications Conf. (ACSAC)*, pages 363–372, 2009.

[18] M. Garnaeva, F. Sinitsyn, Y. Namestnikov, D. Makrushin, and A. Liskin. Overall statistics for 2016. https://tinyurl.com/yasmnff6, December 2016.

[19] M. Garnaeva, J. van der Wiel, D. Makrushin, A. Ivanov, and Y. Namestnikov. Overall statistics for 2015. http://tinyurl.com/zgkkdbj, Dec 2015.

[20] Glassdor, Inc. Glassdoor. http://www.glassdoor.com.

[21] Gnash. GNU Gnash. https://www.gnu.org/software/gnash, 2016.

[22] Google Project Zero. Issue 318: Flash: Memory corruption with ShaderJob width and height TOCTOU condition. https://tinyurl.com/y7sn8qw7, April 2015.

[23] Google Project Zero. Issue 633: Adobe Flash: H264 file causes stack corruption. http://tinyurl.com/jp2o5xs, November 2015.

[24] Google Security Research Database. Monorail: Project-Zero: Issues. http://tinyurl.com/jpkjl6p.

[25] M. Gorelik. CVE-2018-4878: An analysis of the Flash Player hack. *MorphiSec Moving Target Defense*, February 2018. https://tinyurl.com/ya3lyfjz.

[26] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. On Programming Languages And Systems (TOPLAS)*, 28(1):175–205, 2006.

[27] B. Hayak. Deep analysis of CVE-2014-0502 — a double free story. *SpiderLabs*, March 2014. http://tinyurl.com/h9d9v9f.

[28] Home Box Office. HBO Go. http://www.hbogo.com.

[29] IBM X-Force Exchange. Adobe Flash Player code execution CVE-2015-0359. http://tinyurl.com/glxgfzh, 2015.

[30] IBM X-Force Exchange. Adobe Flash Player code execution vulnerability report CVE-2015-5119. http://tinyurl.com/zmz2jqu, 2015.

[31] M. Johns, S. Lekies, and B. Stock. Eradicating DNS rebinding with the extended same-origin policy. In *Proc. of the 22nd USENIX Security Symp.*, pages 621–636, 2013.

[32] W. Jung, S. Kim, and S. Choi. Poster: Deep learning for zero-day Flash malware detection. In *Proc. of the 36th IEEE Symp. on Security & Privacy (S&P)*, 2015.

[33] Kenna Security. How the rise in non-targeted attacks has widened the remediation gap. https://tinyurl.com/yc5kujtg, September 2015.

[34] KernelMode. KernelMode.info. http://www.kernelmode.info/forum.

[35] M. Korolov. Despite recent moves against Adobe, 80% of PCs run expired Flash. *CSO*, October 2015. http://tinyurl.com/zhzcuv9.

[36] K. Levchenko, A. Pitsillidis, N. Chachra, B. Enright, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage. Click trajectories: End-to-end analysis of the spam value chain. In *Proc. of the 32nd IEEE Symp. on Security & Privacy (S&P)*, pages 431–446, 2011.

[37] B. Li. Hacking Team Flash zero-day integrated into exploit kits. *TrendLabs Security Intelligence*, 2015. http://tinyurl.com/jh3tvy3.

[38] Z. Li and X. Wang. FIRM: Capability-based inline mediation of Flash behaviors. In *Proc. of the 26th Annual Computer Security Applications Conf. (ACSAC)*, pages 181–190, 2010.

[39] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proc. of the 19th ACM Conf. on Computer and Communications Security (CCS)*, pages 674–686, 2012.

[40] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *Proc. of the 10th European Symp. on Research in Computer Security (ESORICS)*, pages 355–373, 2005.

[41] Lightspark Developers. Lightspark. http://lightspark.github.io, 2016.

[42] F. Lindner. Preventing Adobe Flash exploitation: Blitzableiter — a signature-less protection tool. Whitepaper, Recurity Labs, July 2010. https://tinyurl.com/ybc2cud7.

[43] R. Masri and M. Aldwairi. Automated malicious advertisement detection using VirusTotal, URLVoid, and TrendMicro. In *Proc. of the 19th Int. Conf. on Information and Communications Security (ICICS)*, pages 336–341, 2017.

[44] Microsoft. 2016 trends in cybersecurity: A quick guide to the most important insights in security. https://tinyurl.com/jb9jspk, June 2016.

[45] A. Middelkoop, A. B. Elyasov, and W. Prasetya. Functional instrumentation of ActionScript programs with Asil. In *Proc. of the 16th Int. Symp. on Implementation and Application of Functional Languages (IFL)*, pages 1–16, 2011.

[46] M. Mimoso. Adobe patches Flash zero day under attack. *ThreatPost*, October 2016. https://tinyurl.com/ybwlgqnm.

[47] Miniclip SA. Miniclip. https://www.miniclip.com, 2018.

[48] NeuroGadget. 5 reasons why Adobe Flash is still important. http://tinyurl.com/zfcbrb9, March 2016.

[49] Offensive Security. Exploits database by Offensive Security. https://www.exploit-db.com.

[50] Ookla. SpeedTest. http://www.speedtest.net.

[51] T. V. Overveldt, C. Kruegel, and G. Vigna. FlashDetect: ActionScript 3 malware detection. In *Proc. of the 15th Int. Symp. on Recent Advances in Intrusion Detection (RAID)*, pages 274–293, 2012.

[52] V. Panteleev. RABCDAsm: Robust ABC [Dis-]Assembler. https://github.com/CyberShadow/RABCDAsm, 2010.

[53] D. Pfaff, S. Hack, and C. Hammer. Learning how to prevent return-oriented programming efficiently. In *Proc. of the 7th Int. Symp. on Engineering Secure Software and Systems (ESSoS)*, pages 68–85, 2015.

[54] T. T. Pham. Trusted access report Microsoft edition: The current state of device security health. https://tinyurl.com/y75jmrbs, September 2016.

[55] P. H. Phung, M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. Venkatakrishnan. Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. *IEEE Trans. on Dependable and Secure Computing (TDSC)*, 12(4):443–457, 2015.

[56] RFSID. New kit, same player: Top 10 vulnerabilities used by exploit kits in 2016. *Recorded Future*, December 2016. https://tinyurl.com/glplb54.

[57] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[58] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[59] M. Sridhar, B. Ferrell, D. V. Karamchandani, and K. W. Hamlen. Flash in the dark: Surveying the landscape of ActionScript security trends and threats. *Journal Information System Security (JISSEC)*, 13(2):59–95, 2017.

[60] M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *Proc. of the 11th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 312–327, 2010.

[61] M. Sridhar, A. Mohanty, F. Yilmaz, V. Tendulkar, and K. W. Hamlen. Inscription: Thwarting ActionScript web attacks from within. Technical report, University of North Carolina Charlotte, 2016.

[62] M. Sridhar, R. Wartell, and K. W. Hamlen. Hippocratic binary instrumentation: First do no harm. *Science Computer Programming (SCP), Special Issue on Invariant Generation*, 93(B):110–124, 2014.

[63] N. Šrndić and P. Laskov. Hidost: A static machine-learning-based detector of malicious files. *EURASIP Journal on Information Security*, 2016(22).

[64] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time URL spam filtering service. In *Proc. of the 32nd IEEE Symp. on Security & Privacy (S&P)*, pages 447–462, 2011.

[65] TrendMicro Research. Research and analysis, TrendMicro, USA. http://tinyurl.com/j49zh7t, 2018.

[66] TrustWave. SpiderLabs. https://tinyurl.com/y9qy8abm, 2018.

[67] VUDU, Inc. Vudu. http://www.vudu.com.

[68] C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck. Analyzing and detecting Flash-based malware using lightweight multi-path exploration. Technical report, University of Göttingen, Germany, December 2015.

[69] C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck. Comprehensive analysis and detection of Flash-based malware. In *Proc. of the 13th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 101–121, 2016.

[70] T. Yan. The latest Flash UAF vulnerabilities in exploit kits. http://tinyurl.com/h4d7omg, May 2015.

[71] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna. The dark alleys of Madison avenue: Understanding malicious advertisements. In *Proc. of the Internet Measurement Conf. (IMC)*, pages 373–380, 2014.