PAD: Programming Third-Party Web Advertisement Censorship

Weihang Wang*, Yonghwi Kwon*, Yunhui Zheng†, Yousra Aafer*, I-Luk Kim*, Wen-Chuan Lee*, Yingqi Liu*, Weijie Meng*, Xiangyu Zhang*, and Patrick Eugster*‡
*Department of Computer Science, Purdue University, West Lafayette, Indiana, USA
{wang1315, kwon58, yaafer, kim1634, lee1938, liu1751, mengw, xyzhang, p}@cs.purdue.edu
†IBM T.J. Watson Research Center, Yorktown Height, New York, USA
zhengyu@us.ibm.com
‡Technische Universität Darmstadt, Darmstadt, Germany
peugster@dsp.tu-darmstadt.de

***Abstract*—In the current online advertisement delivery, an ad slot on a publisher's website may go through multiple layers of bidding and reselling until the final ad content is delivered. The publishers have little control on the ads being displayed on their web pages. As a result, website visitors may suffer from unwanted ads such as malvertising, intrusive ads, and information disclosure ads. Unfortunately, the visitors often blame the publisher for their unpleasant experience and switch to competitor websites. In this paper, we propose a novel programming support system for ad delivery, called PAD, for publisher programmers, who specify their policies on regulating third-party ads shown on their websites. PAD features an expressive specification language and a novel persistent policy enforcement runtime that can self-install and self-protect throughout the entire ad delegation chain. It also provides an ad-specific memory protection scheme that prevents malvertising by corrupting malicious payloads. Our experiments show that PAD has negligible runtime overhead. It effectively suppresses a set of malvertising cases and unwanted ad behaviors reported in the real world, without affecting normal functionalities and regular ads.**

I. Introduction

Web advertisements (ads) are probably the most ubiquitous mashups nowadays and are still growing substantially. The annual revenue of US online advertising increased from 12.5 billion (2005) to 59.6 billion (2015), demonstrating a stunning 17% compound annual growth [44]. They are the main source of income for many Internet companies (e.g. Google and Facebook) and nourish various online services (e.g., news and social networks). Web ads have been widely deployed on most high-traffic websites. In 2017, more than 1.97 million popular websites participate in advertising campaigns [45].

Under the hood, a gigantic digital advertising system connects various parties and fulfills the complicated transactions among them. Websites join the ecosystem as publishers. They offer pre-allocated slots to ad networks and deliver bootstrapping JavaScript libraries hosted by the network to visitors. These ad loading snippets are executed in visitor's browser. They collect and send the visitor/website profiles to the ad exchange. This information is further shared with participating advertisers/networks before the ad exchange conducts a pay-per-impression auction. Advertisers evaluate its value based on the profiles and bid for a particular impression. Finally, additional JavaScript snippets are returned to the visitor and eventually load the actual ad content (e.g. Adobe Flashes, videos, images or texts) from the auction winner.

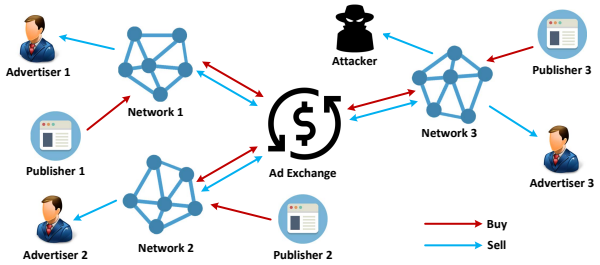

Fig. 1. Ad network system.

However, the *impedance mismatch* between mashup software devOps and software engineering [59] has already resulted in several challenging issues. The nature of dynamically including source-code from all over the world makes the traditional software engineering (e.g. modularity and security guidelines) less applicable. In the context of web ads, although the ad exchange and impression bidding approach greatly improves the efficiency of the advertising business, it brings risks to publishers because websites don't know the auction winners. They have to blindly trust and deliver any contents from the ad networks to their visitors without any control.

Undesired Ads. Even though reputable ad networks filter malformed ads, visitors still receive undesired ads that cause negative user experience or even endanger visitors' systems. Such obnoxious practices easily alienate visitors and irreparably damage websites' reputations [41]. Besides, a recent study [52] shows that while undesired ads can bring in 0.10-0.80 USD, they cost publishers 1.53 USD per thousand impressions, meaning that websites are actually losing money by running such bad ads. To make things even worse, attackers are gaming the system and using this channel to spread malware. As shown in Fig. 1, attackers join the network and act as normal advertisers. However, instead of sending ads, they trick the network and attempt to deliver malicious code to the website visitors. Exploiting the open nature of ad network, attackers can even bid for high-value targets based on their

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

operating systems, browser versions, country locations and other identifying features. By paying a small amount of money, attackers can easily "select" the right type of targets from a huge number of Internet users all over the world.

Malvertising campaigns are becoming a prominent threat. According to Cyphort Labs, "*by 2013 malvertising increased to over* $209,000$ *incidents and generated over* 12.4 *billion malicious ad impressions, which is more than four per each person using the Internet*". During 2014, a 325% increase in malvertising attacks has been observed [39]. In addition, it's believed that leveraging zero-day exploits has made malvertising more effective. In March 2016, visitors of *the NY Times*, *the BBC*, *Newsweek*, *The Hill*, *MSN.com*, *Aol.com*, *the Weather Network*, *the HNL*, and *Realtor.com* have seen ads trying to install malicious ransomware and Trojans. These popular websites have billions of visitors every month [43]. "*The websites themselves weren't compromised. The problem was that the ad networks these sites use - Google, AppNexus, AOL, Rubicon - were tricked into serving the malicious ads, which would lead users to sites hosting an exploit kit*" [46]. The exploit kit then attempts to leverage browser vulnerabilities, penetrate the sandbox and get into the target's computer.

In response to the chaos, Ad-block software are widely used by website visitors. According to a survey by PageFair, an anti-adblocking authority, over 600 million devices run Ad-blocking software globally in 2016 [47]. Although Ad-blockers can mitigate some of the problems caused by third party ads, in the long run simply blocking ads will devastate the economic structure underlying the Internet. Without the monetary revenue gained from ads, various web services would go out of business, which is to no one's benefit. As such, ad-blocking is not a desirable solution to the problem.

Our approach. In this paper, we propose PAD, a novel technique that allows publishers to program their policies to regulate ads rendered on their websites. Such regulations aim to protect website visitors from undesired ad related behaviors such as malvertising or intrusive ads, and protect publishers from detrimental client-side behaviors such as ad blocking. The protection is enforced during the entire life-cycle of the ad delivery, disregarding layers of reselling and delegation.

PAD allows publisher programmers to compose their regulation logic using an expressive language. It further compiles policies to JavaScript (JS) code that executes on a novel persistent runtime, which can self-install in each delegation layer and execute preemptively before any third-party script. The runtime also features a novel memory randomization mechanism that protects memory accesses to block malware at the entry points. PAD can be configured to disable certain regular JS functionalities (e.g., pop-ups) based on the runtime context (e.g., the advertiser's domain). PAD integrates the recent advances in web page randomization [62] to allow the publishers to randomize specified ad content, instead of the entire web page as in [62], to circumvent ad-blocking.

In summary, we make the following contributions.

- We propose the novel idea of allowing publishers to program their regulation of ads displayed on their websites.
- We develop a simple yet expressive language to program regulation policies. Policies are transparently compiled to JS code that enforces the regulation.
- We develop a novel subversion-resilient runtime that ensures persistent regulations throughout the life-cycle of ad delivery and features a novel memory protection technique to prevent malvertising.
- Our evaluation on Alexa Top 200 Global Sites shows that PAD has reasonable overhead and does not affect normal contents. It successfully prevents a large set of real incidents of malvertising and undesired ads we have reproduced.

II. Motivation

In this section, we illustrate undesired ads by examples and motivate our approach.

A. Malvertising

Like normal advertisers, attackers can also participate in the impression bidding. However, instead of delivering ads, they distribute malware. When a website visitor satisfies the conditions (geographic location, browser versions, etc.), she will receive the malicious code via the ad network. To fly under the radar, such code usually runs various checks on the execution context (e.g. existences of debuggers and vulnerable components) before attempting to infect the victim.

Recently, a sophisticated campaign named *AdGholas* was reported [40]. According to a cybersecurity company Proofpoint [42], the attackers used 22 ad networks to distribute malicious code. It affected 113 legitimate sites including *The New York Times, Le Figaro, The Verge, PCMag, IBTimes, ArsTechnica, Daily Mail, Telegraaf, La Gazetta dello Sport, CBS Sports, Top Gear, Urban Dictionary, Playboy, Answers.com, Sky.com, etc*. It was highly effective and infected thousands of victims per day. Depending on locations, different malwares were delivered and installed on victim's computer. *Gozi* (targeting at Internet Solutions for Business) was dropped in Canada, *Terdot.A (aka DELoader)* in Australia, *Godzilla loaded Terdot.A* in UK, and *Gootkit* in Spain.

These malware have caused massive damages. For example, the banking Trojan *Gozi* steals banking information to automate the procedure of robbing online banking customers. According to the indictment by the US Department of Justice [38], *Gozi* alone infected more than 1 million computers and caused tens of millions of dollars in losses. When an infected visitor logs into the online banking system, the Trojan injects form fields (e.g., Fig. 2) that request additional information. These sensitive financial data will be sent to the attackers. *Gozi* also allows attackers to spoof the victim's bank balance. After transferring all available money , *Gozi* forces the victim's browser to display the original balance before the robbery [42].

Now, let's inspect how the *AdGholas* campaign can effectively infect so many victims without being caught for such a long time. Fig. 3 shows the key steps, which involve multiple parties, redirections and anti-monitoring checks. The standard ad network bidding details are omitted.

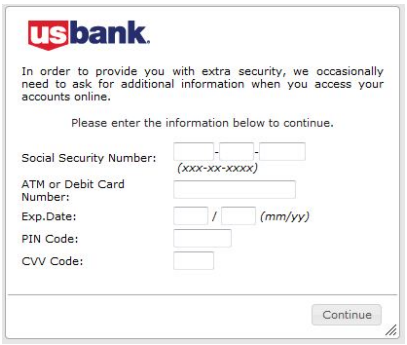

Fig. 2. *Gozi* injects forms to steal sensitive financial information [42].

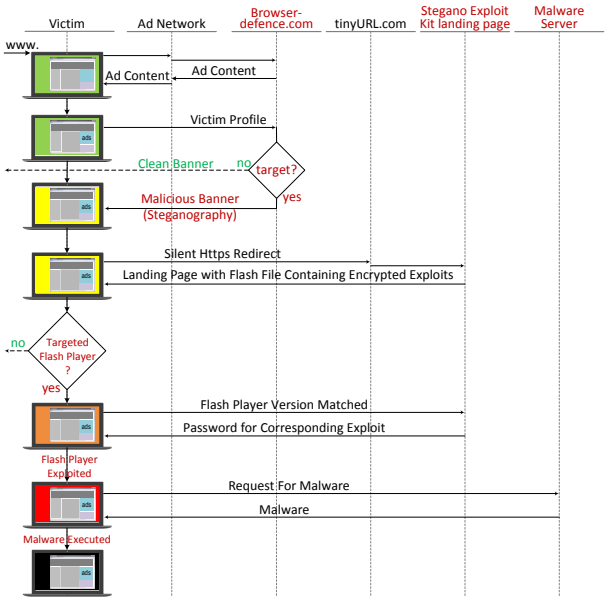

Fig. 3. AdGholas campaign (Dec. 2016) [40]

After receiving the "sniffering" scripts from the ad network, visitors are profiled and filtered by a third-party website *browser-defence.com*. If they are from the targeted regions, they will get a banner with malicious code.

The next step is critical for evasion. *AdGholas* checks the client-side environment and makes sure it's not being monitored. This is done by executing a JS snippet using the Internet Explorer vulnerability CVE-2016-0162. However, *AdGholas* follows a special and innovative design. Instead of retrieving the code explicitly, it hides this malicious script using *steganography* techniques. In particular, this JS snippet is encoded in the alpha channel of the banner image, which defines the transparency of each pixel. Then, The JS snippet is decoded from the image and reconstructed on the fly.

If monitoring utilities such as anti-virus software are not detected, the visitor will be silently redirected to the landing page of the real exploit kit. The kit is designed to target specific versions of the Flash players. If present, the exploited Flash player will download binary executables of backdoors, spyware or trojans and execute them without user consents.

B. Annoying Ads

Besides malvertising, some ads are not deliberately harmful. However, they may blink, float around, pop open a new window or automatically play sounds, which are irritating. HubSpot Research conducted a survey on annoying ads [41]. They interviewed $1,055$ Internet users in the US, UK, Germany and France in 2016, and found that pop-ups and video ads are the most undesired ads. 83% agree that some ads should be disallowed. 85% say obnoxious ads damage publisher's reputation and should be filtered out.

There are already efforts [50], [56], [61], [60] on detecting undesired ads based on the observed behaviors. Ad-blockers powered by predefined blacklists are widely used. But they are not sufficient as they cannot suppress malvertising until their patterns are discovered and modeled. Instead, we argue that allowing publishers to persistently regulate third-party contents is a better solution. Particularly, publishers may specify regulations for an ad slot, which will be enforced for all transactions and delegations introduced by the ad slot.

Fig. 4(a) shows a sample policy supported by PAD for the ad tag `ad_btf`. It disallows any ad with sound after the delegation chain becomes longer than 3 steps (we assume the top-level ads vendors deliver multimedia ads properly). Fig. 4(b) shows the ads without PAD, where sound related resources are highlighted by ①. Fig. 4(c) shows the ads loaded with PAD enabled, where sound related tags are removed.

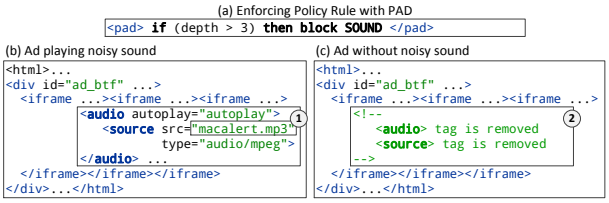

Fig. 4. Preventing Ads Playing Sound

III. Problem Statement

The current ads system has a very open policy: anyone can bid for (and win) an ad slot and can resell the slot to other parties. The current system is fundamentally vulnerable to malicious/undesired ads that are intentionally or unintentionally delivered by bidders. We study and classify the commonly seen undesired ad behaviors in Table I.

TABLE I
Commonly Seen Undesired Ad Behaviors

Category (Type)	Description/Examples
Malvertising (U1)	Delivering malicious software via ads
Intrusive Ads (U2)	Disrupt navigation behavior (e.g., popup)
Information Disclosure (U3)	Access private information for tracking
	Fingerprinting browser/system versions
Inappropriate ads (U4)	Ads with inappropriate contents (e.g., violence)
Nontransparent ads (U5)	Ads with encrypted/encoded flashes
	Ads delivered through HTTP
Ad-blocking (U6)	Blocking legitimate ads

Malvertising uses ad networks to deliver malicious software. *Intrusive ads* distract visitors and cause unpleasant experience. For example, some ads intentionally divert users' attention

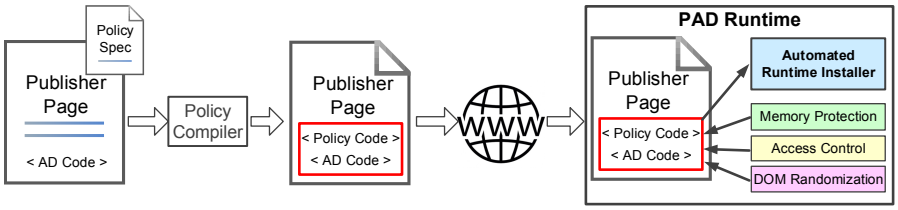

Fig. 5. Overview

with audio or video. *Information disclosure* in ad delivery leaks sensitive user information. Third party ads often contain JS code that collects information on the client side and sends them back to the advertisers. In many cases, sensitive information is collected, causing privacy concerns. For example, user browsing history is often collected to retarget ads. However, exposing such history to nonreputable advertisers should be regulated. Examples of *inappropriate ads* include videos of violence for kids and ads from the publisher's competitors (which should not be displayed on the publisher's web page). Although ad networks have rules to prevent inappropriate ad contents, these rules are usually too general and context-insensitive. *Nontransparent ads* use stealthy techniques for disguising themselves or preventing inspection. For example, a malicious ad may dynamically construct a (malicious) flash to avoid being detected. *Ad-blocking* removes ads from web pages. While undesired ads are suppressed, ad-blockers also block legitimate ads, eventually damaging the ecosystem.

While ad networks strive to suppress undesired ads, they usually lack the necessary context to perform the needed fine-grained regulation. For example, they may not have enough information about the publisher site to determine what ads are deemed inappropriate. They also lack the resources and mechanism to regulate the huge volume of ads. Note that after bidding, ads are often loaded directly from the advertisers' servers to the client, without going through the ad networks. We consider publishers to be the ideal party to regulate unwanted ads and third-party contents because they have the comprehensive contexts. They also have a strong incentive to deliver relevant ads to the right consumers. In addition, comparing to having centralized censorship on ad networks, it is more scalable to do so on the individual client machines.

Therefore, our problem statement is to *allow publisher developers to program their regulation of third party ads shown on their pages in a reliable and persistent fashion.*

IV. Overview

We propose a novel programming support system for third party ad censorship called PAD. As shown in Fig. 5, PAD consists of three main components: the policy specification language, the policy compiler that compiles specification to JS code, and a runtime that facilitates regulations.

Policy Specification Language. PAD provides a language for publisher developers to program their ad censorship. The developers engineer their regulation logic within an ad tag in the publisher's content page using our language. Such tags indicate the locations of ads, which are populated when the content page is loaded.

Policy Compiler. The policy compiler compiles censorship policies in a content page to the corresponding JS code. Only compiled pages are deployed. Each ad tag has its independent policy code. In other words, censorship is ad tag specific. This is because individual tags independently load ads from different sources at runtime, requiring different regulations. Programming global censorship (for an entire page or publisher domain) is possible but we leave it to our future work.

PAD Runtime. PAD runtime consists of 4 modules: Automated Runtime Installer (ARI) (§V-C1), Memory Protection (§V-C2), Access Control (§V-C3) and Randomization (§V-C4). They are essentially JS code that is shipped to the client.

- **Automated Runtime Installer (ARI).** ARI is the core engine enforcing the regulating code in all delegation layers. For multi-layer ad networks, PAD monitors dynamically generated contents and propagates itself into the next layer.
- **Memory Protection.** Based on our extensive study on malvertising campaigns in the wild, we observe that the manipulation of consecutive memory regions in JS or ActionScript is the root cause of payload injection. To mitigate malvertising attacks, PAD prevents payload injection by perturbing values in consecutive memory regions.
- **Access Control.** This module allows publishers to restrict undesired ads such as information leak ads and intrusive ads. The module essentially intercepts and/or masks some JS APIs that are critical for undesired ads.
- **Randomization.** To circumvent client-side Ad-blockers, our runtime leverages an existing web page randomization technique WebRanz [62] to bypass Ad-blockers' blacklist.

V. Design

In this section, we present the design of PAD. We describe each component in detail and reason about our design choices.

A. Policy Specifications

Program	P	::=	s
Stmt	s	::=	s_1; s_2 | `protect` sbj | `block` ca | `randomize` a | `if` (e) `then` s
Expr	e	::=	$curUrl$ `op` u | $depth$ `op` c | $history$ `contains` $\{u_1, u2, u3, ...\}$
Operator	op	::=	!= | == | < | = | >
Subject	sbj	::=	`MEMORY` | `FLASH`
Capability	ca	::=	`GEO-DATA` | `COOKIE` | `BROWSER-VERSION` | `FLASH-PLAYER-VERSION` | `EMBEDDED-FLASH` | `LOADING` | `MIMETYPE` | `POPUP` | `SOUND` | `SRCLESS-IFRAME` | `REDIRECTION`
Attribute	a	::=	`CANVAS-DATA` | `DOM`
Const	c	::=	$\{0, 1, 2, ...\}$
Url	$curUrl, u$		

Fig. 6. Language.

The policy specification language is presented in Fig. 6. The language supports a few statements that specify the actions

```
1  <pad>
2     if (curUrl != "doubleClick.com") then
3        block BROWSER-VERSION
4  </pad>
5  ...
6  <div id = "google_ads"></div>
```

Fig. 7. Publisher Page Code with Policy Specification

```
1  <script>
2     ARI({"google_ads"});
3  </script>
4  <div id = "google_ads">
5     <script>
6     if (curUrl != "doubleClick.com") {
7        Object.defineProperty(window, "navigator", {
8           get: function() {
9              return null;
10          }
11       });
12    }
13    </script>
14     ...
15 </div>
```

Fig. 8. Publisher Page with Compiled Policy Code

for undesired ad behaviors, for example, **protect** memory, **block** clients' cookies or geo-location data from being accessed by ad scripts, and **randomize** DOM to circumvent ad blockers. The language also supports conditional branches to allow publishers to specify the conditions in which the above-mentioned actions should be taken. For example, a publisher may wish to apply memory protection for certain ad networks by comparing the current domain (*curUrl*) with some domain u specified by the publisher. We also support comparison of delegation history, which records the entire ad delegation/reselling history. Such history is quite useful. For example, a cyclic delegation chain (i.e., an ad dealer sells an ad slot and later buys it back) may suggest abnormal behavior. Therefore, the current domain, the delegation depth, and the delegation history are of special interest and explicitly modeled in the language. Publisher developers can use regular JS together with our language to achieve maximum feasibility.

B. Policy Compiler

Given a content page with policy specs, PAD parses the page to a DOM tree using `htmlparser2`. It first adds the automated runtime installer at the beginning of the page to make sure it gets executed before any other elements are loaded. It then traverses the tree to identify all ad tags and the corresponding policy specs. The JS statements in the specs are simply copied to the output page whereas the statements in our language are translated to the corresponding JS code to fulfill the specific actions, including updating critical state variables (e.g., delegation depth) and invoking API functions provided by our runtime. The details are elided.

Fig. 7 and Fig. 8 illustrate how the policy compiler compiles a program with access control specs. In Fig. 7, the publisher includes an ad (with `id` of "*google_ads*") at line 6 and inserts a spec at lines 2-3 to block browser versions from being fingerprinted. In Fig. 8, the compiled policy code is inserted at lines 6-12. The compiled code redefines `window.navigator` and changes its getter to returning `null` when a read attempt is executed. The code at lines 6-12 is only effective at the current layer of ad delegation. To propagate the policy code when inner frames are created, the function **ARI** at line 2 invokes the automated runtime installer (ARI) to self-install the policy code along the entire propagation chain. We will discuss the design of ARI in Sec. V-C1.

C. PAD Runtime

Algorithm 1 Automated Runtime Installer

INPUT : A compiled web app with instrumentation guarding the 1st layer of each Ad
OUTPUT: Enforcing policy specifications over the entire delegation chain during runtime

```
1  procedure ARI(ads)
2     for ad in ads do
3        depth ← 0
4        history ← ∅
5        globalVar ← {depth, history}
6        policyCode ← getPolicyCode(ad)
7        if innerIframeCreated() then
8           propagate(policyCode, globalVar)
9  procedure PROPAGATE(policyCode,globalVar)
10    globalVar.depth ← globalVar.depth + 1
11    globalVar.history ← globalVar.history ∪ curUrl
12    inject(globalVar)
13    inject(policyCode)
14    if innerIframeCreated() then
15       propagate(policyCode, globalVar)
```

1) Automated Runtime Installer (ARI): ARI provides persistent protection along the ad delegation chain. At each delegation layer, new content (such as HTML/JS/Flash/inner frame) can be dynamically generated via JS functions such as `document.write()`. ARI propagates PAD's protection logic along the delegation chain.

Alg. 1 describes the design of ARI. It takes a compiled publisher page with ads guarded by instrumentation code. For each ad within the page, ARI initializes global variables such as the delegation `depth` (line 3), domain `history` (line 4) and extracts the policy code generated from compiler for each specification (line 6). When an inner `<iframe>` is created, ARI invokes a recursive function **propagate()** to propagate the policy code and global variables to the next layer of delegation (lines 7-8). The recursive function **propagate()** increments the delegation `depth` (line 10) and updates the domain `history` to include the current domain (line 11). Then it injects the updated global variables and policy code into the current layer (lines 12-13). The recursive procedure continues to propagate if a next layer of `<iframe>` is dynamically generated.

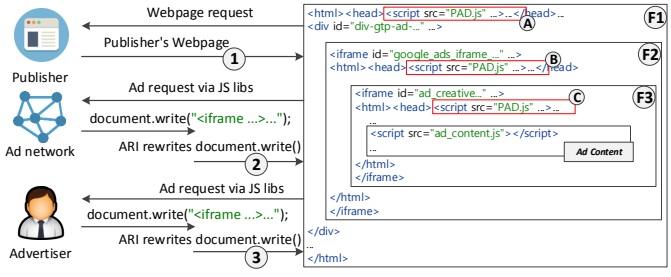

Fig. 9. Recursively Applied *ARI* on Multiple Delegations

Fig. 9 depicts the code snippet of how ARI enforces PAD in the delegation layers. The right side of Fig. 9 shows a

webpage including dynamically generated ad content during the ad delivery process. Note that in the example, the publisher already installed PAD on their webpages by including `<script src="PAD.js" ...>` (Ⓐ in Fig. 9).

When a user visits the website, the web page with PAD is loaded (①). During the page load, an ad request is initiated to ad networks to retrieve the ad contents (e.g., HTML/JS/Flash). The ad contents are dynamically inserted into the webpage (②) by creating an inner frame (F2). Note that the `document.write()` at ② is executed within frame (F1), where PAD is already installed. Hence, PAD intercepts it at ② and inserts `<script src="PAD.js" ...>` (Ⓑ) to install itself within the new inner frame. Similarly, another layer of delegation is dynamically inserted into the inner page (F2). At ③, PAD inserts `<script src="PAD.js" ...>` (Ⓒ) to install itself into the new inner frame (F3), where the actual ad is finally delivered. Since PAD is propagated through delegations, it can enforce the protections persistently.

2) Memory Protection: In this section, we first analyze memory management related attack vectors in ads. Then, we introduce our memory protection runtime support which mitigates malvertising by corrupting malicious payloads.

Memory Management in Web Ads. As ads include JS/Flash contents, they can allocate and access memory via JS or ActionScript (AS) interfaces. We first explain different ways of accessing memory and how malicious ads exploit them.

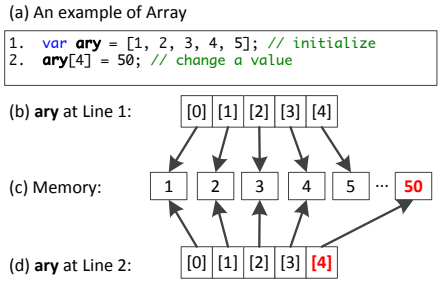

Fig. 10. Immutable Object

- **Immutable Objects.** Immutable Objects are objects whose previously stored values on the memory are not overwritten. Fig. 10(a) illustrates a simple JS/AS program that initializes an array and updates its fifth element. Fig. 10(b-c) show how the array is allocated in the memory. Note that `Array` in JS is actually implemented as a hash table and a set of pointers to elements. More importantly, it is *immutable*, meaning that changing a value in the array does not change the previously stored value. Instead, as shown in Fig. 10(c), a new memory buffer is allocated for the new value.
- **Mutable Objects (MO).** Some JS and AS objects can access memory directly. We call such objects *mutable objects*. Unlike immutable objects, mutable objects allow overwriting previously stored values in memory. More importantly, the buffer allocated for a mutable object is consecutive. `TypedArray` (e.g., `Uint8Array`, `Uint16Array` and `Uint32Array`) and `DataView` in JS and `ByteArray`/`Vector` in AS are popular classes that can create mutable objects. Besides, `Canvas` in JS and `ImageData` are used to access image data (e.g., pixel colors). They can create mutable objects too. Indeed, many exploits leverage them to directly write consecutive memory by changing image data (e.g., pixel colors in RGBA format).
- **Consecutive Immutable Objects (CIO).** We call multiple immutable objects that may be allocated in consecutive memory *consecutive immutable object*. For example, in modern browsers (e.g., Firefox and Chrome), `Array` objects in JS are allocated in consecutive buffers when they hold a primitive type (e.g., integer). `String` objects are also allocated in consecutive buffers.

Attacks on Memory in Malvertising. We analyze popular malvertising campaigns and exploit kits used in attacks and find that most attacks exploit vulnerabilities in JS/AS memory management. In particular, they use mutable objects to inject and trigger malicious payloads. In the following paragraphs, we explain details of such attacks and their root causes.

- **Payload Injection and Triggering via Mutable Objects.** Mutable objects are widely used to inject and trigger malicious payloads by overwriting critical values in memory (e.g., return addresses). Specifically, upon the allocation of a mutable object, a new buffer is allocated and the corresponding memory address and length are stored in memory. When the mutable object is used, the address pointing to the start of the buffer is retrieved in order to compute the address for the access. If the address or the length of the allocated buffer is compromised, an attacker can access arbitrary memory just like accessing this mutable object. Such compromise is often achieved through vulnerabilities such as type confusion [25].
- **Payload Injection via CIO.** As consecutive memory buffers are allocated for CIO, they can be leveraged to inject payloads like heap spraying. However, immutable objects cannot be directly used to write arbitrary memory and heap spraying itself does not trigger the injected payload. Therefore, to launch an attack, the attacker must rely on another vulnerability to trigger the injected code.

Algorithm 2 Memory Protection via Value Randomization

```
    INPUT  : A JS statement such as new Uint32Array([21, 3, 15])
    OUTPUT: A Uint32Array object with value encoded
1   procedure OVERLOADED_UINT32ARRAY(values)
2       valuesInMemory ← Uint32Array (values.length)
3       for i = 0; i < values.length ; i++ do
4           valuesInMemory[i] ← encode (values[i])
5       this.values ← Uint32Array (valuesInMemory)
6       return this
    INPUT  : A JS statement such as u32arr.indexOf(15)
    OUTPUT: Return the index of "15" from [21, 3, 15]
7   procedure OVERLOADED_INDEXOF(elem)
8       valuesInMemory ←this.values
9       for i = 0; i < valuesInMemory.length ; i++ do
10          values[i] ← decode (valuesInMemory[i])
11      return this.indexOf (values)
```

Mitigating Malvertising via Memory Protection. We mitigate malvertising via memory protection, which is driven by the procedure of attacks being launched using mutable objects

and consecutive immutable objects. Malvertising attacks are conducted in two critical steps. First, a malicious payload is injected into the victim's memory via legitimate channels (e.g., loading an image with a payload prepared by an exploit kit), and heap spraying is often used due to address space randomization techniques. Later, the payload is triggered and executed by exploiting vulnerabilities. As these two steps are essential to its success, malvertising can be suppressed by breaking either step. We describe how we achieve this via value and data layout randomization.

- **Value Randomization (Encoding/Decoding Values).** PAD breaks a malicious payload by perturbing values in memory. In particular, it encodes values written to memory via MO and CIO, and decodes them upon retrieval using the same key employed in encoding. In this scenario, malicious payload is encoded before being injected to memory. However, since a malicious payload is always executed natively, without going through the JS/AS interfaces, it is not properly decoded before execution and hence broken. Alg. 2 illustrates how value randomization is realized for `Uint32Array` objects. Specifically, when a new `Uint32Array` object is created, instead of storing the arguments ([21, 3, 15]) as array values, the overloaded constructor (lines 1-6) **`encodes`** each element and the encoded values are stored in memory. When the statement invokes `indexOf(15)` on the previously created object, the `Uint32Array.indexOf` is overloaded (lines 7-11). To return the index of element "15" (i.e., 2), the values in memory are **`decoded`** and the native implementation of **`indexOf`** is invoked to return (lines 9-11).
- **Data Layout Randomization.** PAD also randomizes the underlying memory layout for JS/AS objects to break a malicious payload, which is quite sensitive to memory structures. In fact, attackers leverage CIOs to inject payloads mainly because they provide a convenient (and reliable) way to hold the carefully crafted payload in a consecutive sequence of buffers. To randomize memory layout, when a new CIO is created, PAD creates new dummy objects and inserts them in between the original buffers. For instance, when a program creates an array [1,2,3], PAD constructs [1,`R`,2,`R`,3] instead, where `R` represents a random value. As a result, if an attacker tries to inject malicious ROP payloads with CIO, the random values placed between ROP gadgets will break the exploit. Note that the semantics of the array is not broken as when the array is accessed, we translate the index of the real values, skipping the inserted `R`.

3) Access Control: As discussed in Sec. V-B, the policy compiler compiles the specification program and redefines objects such as `window.navigator` to block the access to particular objects and achieve access control. During runtime, the redefined getters will be invoked rather than the native functions. For example, as illustrated in Fig. 8 (line 6-12), any read access to the browser information from domains other than *doubleclick.com* via `window.navigator` is restricted. Security sensitive objects and objects related with intrusive ads are identified and redefined during runtime. We omit the discussion of other objects as they are similar to above-mentioned case.

4) Randomization: WebRanz [62] is integrated into PAD. WebRanz is a web page randomization technique which protects advertisements from being blocked by ad-blockers. Specifically, it randomizes URLs and DOM element properties (e.g., `id` and `name`), which ad-blockers rely on to locate and remove ads. However, WebRanz is not sensitive to *ad tags*. Instead, its randomization technique is applied to the entire page, resulting in a high overhead. PAD only performs randomization on ad contents within *ad tags* specified by publishers. By doing so, publishers can deliver ads without being blocked by ad-blockers with less overhead. As the randomization technique is not our focus, details are omitted.

D. Threat Model

PAD assumes publishers and website visitors are benign, while the ad-networks and advertisers may be malicious. Note that, in most malvertising campaigns, the malicious party is the advertiser. While the ad-networks might also be compromised, protecting compromised publishers is beyond the scope of our paper. In fact, if a publisher is compromised, any operation in the compromised page cannot be trusted. Similarly, protecting visitors with compromised systems (e.g., OS/Kernel/Browser) is out of our scope.

Accessing Original Definitions. An attacker may try to bypass PAD by locating and directly using the original functions or classes. A straightforward way is scanning all variables to find the ones holding the original versions. For example, `window` in JS can be used to enumerate all defined global variables. By recursively resolving the global variables, one may gain access to any variable including the variables holding the original definitions. To prevent this, we disable variable enumerations via `Object.defineProperty()` as a part of our runtime installer. Note that `Object.defineProperty()` is protected so an attacker cannot abuse this interface. Moreover, even though it is not enumerable, if the variable names are statically defined, they can be directly accessed by names. Hence, we generate random names for the variables that contain original definitions. The name is randomized on each load to make it a moving target.

Removing Our Protection. An attacker may try to remove our protection by replacing classes and methods overwritten by PAD with their own version. To prevent this, we use `defineProperty` to make classes and methods we modified read-only. Specifically, we set `writable` property to `false` and override `defineProperty` to prevent further changes. As we do it at the very beginning of each delegation, no JS code including malicious one can disable our definition.

Disabling Automated Runtime Installer. As PAD's protection depends on ARI which installs itself and PAD, an attacker may try to disable ARI. However, to achieve this, the attacker needs to bypass our methods by calling the original ones which we made impossible as discussed above. In brief, one cannot remove our methods as they are read-only (e.g., `writable: false`). To prevent any attempts

TABLE II
CHARACTERISTICS OF MALICIOUS AD CAMPAIGNS

Ad	Date	Publisher (Ad Networks)	First Ad Redirect
1*	17-01-14 [25]	N/A (PropellerAds, ClickAdu etc.)	N/A
2*	16-12-06 [33]	Yahoo, MSN (N/A)	broxu.com
3	16-07-25 [10]	N/A (RevenueHits)	foundationarcet.org
4	16-07-05 [11]	N/A (RevenueHits)	top4download.org
5*	16-05-25 [32]	answers.com (DoubleClick, Zedo etc.)	rfihub.com
6	16-03-04 [34]	nalsee.com (N/A)	49.238.137.2
7*	15-09-30 [12]	Japanese News Sites (N/A)	N/A
8	14-11-11 [9]	wira-ku.com (N/A)	a.horsered.com
9	14-10-06 [8]	network-tools.com (N/A)	ox.help.org
10	14-09-29 [7]	hindustantimes.com (N/A)	static.rcs7.org
11	14-09-11 [6]	marica.bg (N/A)	serve.intelelink.net
12	14-09-05 [5]	centralpark.com (N/A)	saw.piroscia.com
13	14-08-25 [3]	urbantoronto.ca (N/A)	reserve.sandystrong.org
14	14-08-22 [4]	onlinenewspapers.com (N/A)	c.ic.com.au
15	14-06-28 [2]	N/A (N/A)	zamcheck.org
16	14-06-19 [1]	N/A (N/A)	stat.litecsys.com

*: Proof-of-Concept (PoC) sample.
N/A: Publisher/Ad traffic is redacted or not included in sample.

to change the `writable` property of object, we override `defineProperty` to disallow change requests.

VI. EVALUATION

PAD is implemented in JavaScript and ActionScript, leveraging a set of *Node.js* utilities. We investigated the following research questions to evaluate the effectiveness of PAD to regulate third-party ads over the complex ads delegation process.

RQ1 How effective is PAD on preventing malvertising?
RQ2 How effective is PAD on preventing information leak ads, non-transparent ads, inappropriate and intrusive ads?
RQ3 How much runtime overhead does PAD incur?

TABLE III
MALICIOUS AD CAMPAIGNS' ATTACK VECTORS

Ad	Exploit Kit	Payload	CVE Report	Attack Vector
1	Neutrino	Trojans	[26], [27]	`DataView`
2	Stegano	Spyware	[24], [23], [21], [22]	`ByteArray`
3	Magnitude	Cerber*	[24]	`ByteArray`
4	Magnitude	Cerber*	UNREPORTED	`ByteArray`
5	Angler	Ransomware	[24]	`ByteArray`
6	KaiXin	PeCompact2, Trojan	[18]	`Vector`
7	Angler	Spyware	[19], [20]	`Uint32Array, Vector`
8	Angler	Poweliks	[17], [16], [15]	`ByteArray`
9	Sweet Orange	Zemot#	[17]	`Vector`
10	Nuclear	Cryptowall*	[17]	`Vector`
11	Sweet Orange	Gimemo Trojan	[17]	`Vector`
12	Sweet Orange	Zusy Trojan	[17]	`Vector`
13	Nuclear	Zemot#	[17]	`Vector`
14	Nuclear	Zusy Trojan	[17]	`Vector`
15	Sweet Orange	Trojan Dropper	[17]	`Vector`
16	Nuclear	Zbot, Spyware	[17]	`Vector`

*: Ransomware. #: Downloader.

A. Experimental Methodology

To answer the research questions, we run PAD on real world ad examples. Specifically, we collect 16 malvertising ad samples (for **RQ1**) and 15 undesired ads (for **RQ2**). We gather the source code of publisher web pages and host them on our own web server. For the cases in which the publisher is not reported or intentionally redacted, we set up a test page to include the undesired ad(s) on our web server.

B. Experimental Results

RQ1: Malvertising Attacks. Table. II and III show the malicious ad samples and the memory protection enforced. Column Publisher lists the websites redirecting to malware through ad networks. First Ad Redirect and Ad Networks show the first malicious redirect and the ad networks involved in each malicious campaign.

We collect 12 samples of malvertising in the wild and 4 proof-of-concept (PoC) samples from online malvertising reports. 15 of them are exploiting Adobe Flash Player, 3 samples for Internet Explorer and 1 for Microsoft Edge. Note that it is very hard to collect malvertising samples from the wild, as most malicious campaigns simply change their IPs or ad networks once they are caught. Moreover, malvertising attacks that exploit zero-day vulnerabilities make them very difficult to be detected. We have validated that all attacks can be suppressed by one line of memory protection specification. Note that PAD provides a general protection hence it prevents an **unreported** case (Ad 4) too.

RQ2: Undesired Ads. We investigate how PAD benefits the protection against intrusive ads (U2), information disclosure ads (U3), inappropriate ads (U4), and nontransparent ads (U5). Table. IV summarizes our findings. Column Ad and Date are the reference and date of the undesired ad. Note that entries with + are ads that are active as the time of paper submission. The table also shows the Publisher involved in each ad campaign and the access control specification publishers can leverage to suppress each undesired ad.

For example, ad [37] on *nytimes.com* redirects users to a malicious page. Developers can leverage our specification language `block REDIRECTION` to disable redirections. [36] on *youtube.com* directs users to a flash ad which fingerprints users' browser. To prevent users' browsers being fingerprinted, Youtube developers can specify `block BROWSER-VERSION` with PAD. Preschool sites *funology.com* and *twistynoodle.com* serve inappropriate ads to children via ad retargeting service. Publishers can use PAD to block ads from particular domains. Ad [28] serves from ad network domain *ero-advertising.com* embedded a Flash ad on *stopmalvertising.com* to perform DDoS attacks. `block EMBEDDED-FLASH` can be used to block such ads.

As shown in the result, all undesired ad behaviors are prevented by enforcing our access control policies. A detailed case study is discussed in VI-C3.

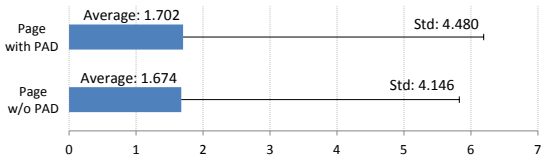

Fig. 11. Page Load Latency

RQ3: Runtime Overhead. Finally, to understand the runtime overhead induced by PAD, we study the average latency of rendering a web page with and without PAD. We choose Alexa top 200 websites, load each page 10 times and calculate the average load time. As shown in Fig. 11, a web browser takes a slightly longer time to render a web page when it is enforced with our protection. This is because (1) PAD introduces additional JavaScript and the browser needs extra

time to parse the code; (2) the self-propagating enforcement intercepts the multiple layers of ad delivery on the fly. As the size of additional JavaScript is small and the interception is lightweight, the overhead of rendering a page with PAD protection is negligible (1.67%). To avoid different ads being loaded across runs (of the same website), we save all the contents, including ads, and host them on our own proxy. We also used a plugin to check each page load event. The plugin is designed in such a way that any unsuccessful load causes the browser to hang forever. In our experiment, we have not encountered any such cases. This suggests that PAD does not affect normal functionalities (including ads).

TABLE IV
ACCESS CONTROL ON UNDESIRED AD BEHAVIORS

Type	Ad	Date	Publisher	Access Control Spec
U2*	+	+	watchfomny.tv torrentz.eu	`block POPUP`
	[37]	09-09-13	nytimes.com	`block REDIRECTION`
U3*	[31]	16-04-12	N/A	`block MIMETYPE`
	[31]	16-02-06	N/A	
	[36]	14-02-20	youtube.com	`block BROWSER-VERSION`
	[29]	12-01-21	fileserve.com	`block FLASH-PLAYER-VERSION`
U4*	+	+	funology.com twistynoodle.com	`if curUrl == "xxx.com"` `then block LOADING`
	+	+	realstreamunited.tv	
	+	+	stream2watch.cc	
U5*	[40]	16-12-06	N/A	`randomize CANVAS-DATA`
	[14]	15-05-12	N/A	`block LOADING`
	[35]	15-05-07	N/A	`block SRCLESS-IFRAME`
	[30]	15-02-04	gopego.com	`block EMBEDDED-FLASH`
	[13]	15-01-24	N/A	
	[28]	11-07-22	stopmalvertising.com	

+: The ad is active now. N/A: Publisher is redacted or not included in report.

C. Case Study

In this section, we show three case studies to demonstrate how PAD prevents undesired ad behavior in practice.

1) Type Confusion Vulnerability in Microsoft Edge: Type confusion vulnerabilities have been reported lately in the JS Engine (Chakra) of Microsoft Edge. When an ad is delivered, JS can be injected for logging and dynamic ad delivery. Unfortunately, a malicious advertiser can also inject a malicious JS file to exploit the vulnerabilities to execute malicious code.

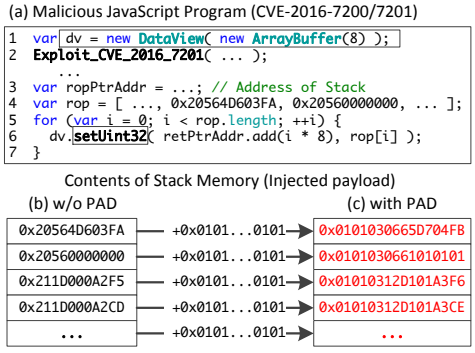

Fig. 12. Malicious JS Program Exploiting `DataView`

Fig. 12(a) shows a malicious JS program which exploits two vulnerabilities (CVE-2016-7200 and CVE-2016-7201). In particular, it first creates a DataView object which will be used to inject a malicious payload at Line 1. Then, at Line 2, it leverages the vulnerabilities to obtain a corrupted DataView object (`dv`). Specifically, it exploits CVE-2016-7201 which is a type confusion vulnerability to obtain the ability to modify a target address of DataView when it reads and writes. As we discussed in Sec. V-C2, a DataView object internally stores a pointer to a buffer which holds its value and the length of the buffer. CVE-2016-7201 essentially changes the pointer of the buffer via a type confusion vulnerability. As a result, after Line 2, the JS program can read and write arbitrary memory through the corrupted object `dv`. Specifically, by changing the first argument of `setUint32` which specifies an offset of the DataView's buffer for writing a value, an attacker can overwrite values on any address he/she wants to. Then, at Line 3, it obtains a stack address containing a return address by exploiting address information leak. We omit the detail due to the space limit. At Line 4, it constructs a malicious ROP payload (`rop[]`) within an array. Finally, the program injects the malicious payload at Lines 5-7 through the `setUint32()` method. Note that `dv` is already compromised so that it can point to arbitrary buffer. Therefore, by providing addresses pointing to the program's stack, it injects the malicious payload.

PAD protects the attack by mutating input values passed to `DataView`. In particular, PAD overrides the `DataView.setUint32()` to mutate any input parameter. In this example, to make the discussion easier, we mutate the input by adding 1 to every byte of a value passed to `setUint32()`. For example, a 0x64D603FA input value will be mutated to 0x65D704FB. Note that in our implementation, we use a one time pad encoding scheme. Hence, we apply different mutations every time. Fig. 12(c) shows an example of a corrupted stack with a simple addition (+1 for each byte). Observe that such a small addition completely breaks the functionality of the payload and leads the exploit to a crash as the execution jumps to an unmapped memory.

In addition to overriding setters (e.g., `setUint32()`), we do also override getters (e.g., `getUint32()`) to decode retrieved values in order to enable seamless execution of benign program operations via `DataView`. The decoding performs the exact reverse operation of the encoding process. For example, if we encode by adding 0x1 to the values, we decode by subtracting 0x1 at getters invocation.

2) Pixel Bender Parser Vulnerability in Adobe Flash Player: Similar to the type confusion attack in Microsoft Edge, `Vector` in AS can be corrupted by a vulnerability on Pixel Bender Parser which is a high-performance graphic programming interface for image processing.

```
1 var vt = new Vector.<int>( ... );
2 Exploit_CVE-2014-0515( ... );
    ...
3 var ropPtrAddr = ...; // Address of Stack
4 var rop = [ ..., 0x70ff016a, 0x70fff870, ... ];
5 for ( ... ) {
6   vt[retPtrAddr + i * 4] = rop[i];
7 }
```

Fig. 13. Malicious AS Program Exploiting `Vector`

Fig. 13 shows a malicious flash file which contains AS code exploiting CVE-2014-0515 vulnerability. At Line 1, the code

first allocates a `Vector`. The function at line 2 represents exploiting the vulnerability in order to overwrite the `length` of `Vector`. Note that `Vector` in AS and `DataView` in JS have similar internal representations and thus share the same problems of the vulnerable `length` information. By overwriting the `length` of `Vector`, an attacker can gain arbitrary memory access. Similar to the Edge case, it prepares ROP gadgets and stack addresses, then injects the ROP payload at Line 6. Note that it uses the [] operator to inject the payload, instead of invoking a function. To handle this, PAD also overrides the operator by using `defineProperty` with index (e.g., 1, 2, and so on). Note that in JS/AS `obj[name]` is equivalent to `obj.name`. Hence, we override `obj.1()`, `obj.2()`, and so on. As in JS, PAD prevents the attack by encoding the values passed through [] operator.

3) Preventing Geo-location Information Access: Personalized ads are chosen based on the profile of targeted customers. While it is valuable to advertisers, collecting sensitive information to deliver targeted ads is quite controversial. When multiple ad networks are involved, it is hard to know and control who collects what information. Hence, it is important for publishers to protect their customers' sensitive information from being collected by ad networks or advertisers.

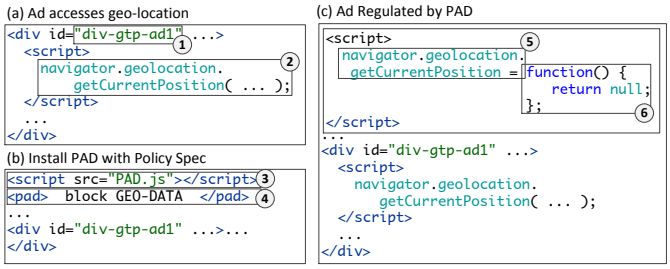

Fig. 14. Preventing Geo-Location Accesses

In this case study, we show how PAD enables a publisher to protect its customer's geo-location from being disclosed to ad networks and advertisers. Fig. 14(a) shows the publisher web page without PAD. The tag `div` with id `div-gpt-ad1` (①) serves as an *ad slot* on the page. ② shows the JS code loaded as a part of the ad delivery. It accesses the geo-location of the customer via the JS interface `navigator.geolocation.getCurrentPosition`.

Fig. 14(b) shows how PAD can be employed to enforce the geo-location access control. First, the publisher developer includes `<script src="PAD.js"></script>` at the beginning of the page (Fig. 14-③). Then, she can add policy spec `block GEO-DATA` as shown in Fig. 14-④ to enforce that "blocks geo-data on any delegation (any url)".

Fig. 14(c) shows how PAD enforces the policy at runtime. Specifically, PAD first overrides `navigator.geolocation.getCurrentPosition` (⑤) to intercepts the access to geo-location. When `getCurrentPosition` is invoked, PAD checks the current specifications and applies actions accordingly. In this case, the access control for blocking geo-location takes effect. PAD returns empty geo-location information instead of returning the real location(⑥). Besides blocking, flexible fuzzing methods such as reducing location accuracy can be easily supported in PAD.

VII. Related Work

Malvertising detection. Zarras et al. [64] investigated the source of malvertising and showed that every publisher without an exclusive agreement with the advertiser is likely to serve malicious ads. Xing et al. [63] performed a large scale study on malvertising in ad-injecting browser extensions. OdoSwiff [50] detects malicious flash files based on known patterns in the code and execution traces. In particular, it statically looks for invalid images, suspicious jumps and APIs used in known exploits. MadTracer [56] detects malvertising based on rules learned from the ad delivery paths. Stringhini et al. [61] and Mekky et al. [58] identify malicious content based on HTTP redirections. Poornachandran et al. [60] proposed a classifier-based malvertising detection approach. As learning based detections rely on patterns found in known attacks, they may not catch unknown attacks. In contrast, we break the payloads that are carefully crafted for specific defects, we can stop the attacks even targeting at zero-day vulnerabilities.

Ads and security policies. Alt [48] presents an advertising platform based on adaptive profiles, where visitors can setup profiles to obtain favorable ads. AdJail [57] is a framework for publishers to specify confidentiality and integrity policies on ads. AdSentry [49] allows publishers and end users to specify access control policies for ads. Gui et al. [53] surveyed 21 Android apps and identified several undesired consequences of ads including hidden costs and complaints. They further studied 400 mobile ad reviews to identify complaint topics [54]. Hussein et al. [55] use a runtime verification tool to enforce security specification for Java. Ghotbi and Fischer [51] presents a fine-grained role- and attribute-based access control model. We focus on developing a policy language for publishers so that they can easily specify how PAD operates.

VIII. Conclusion

We propose a novel programming support system PAD to regulate third party ads. PAD allows publisher programmers to specify and enforce their ad regulation policies throughout the entire ad delegation chain. This is enabled by a self-installing and self-protecting runtime. PAD also features an ad-specific memory protection scheme that prevents malvertising. Our experiments show that PAD has negligible overhead and can effectively prevent real world malvertising and undesired ads.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, 1320306, 1618923, and 1421910, ONR under contracts N000141410468 and N000141712947, ERC grant FP7-617805, and BMBF project CRISP. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] 2014-06-19 - NUCLEAR EK FROM 5.135.28.118. http://www.malware-traffic-analysis.net/2014/06/19/index.html.

[2] 2014-06-28 - SWEET ORANGE EK FROM 94.185.80.43 PORT 8590. http://www.malware-traffic-analysis.net/2014/06/28/index.html.

[3] 2014-08-15 - NUCLEAR EK FROM 178.32.92.105. http://www.malware-traffic-analysis.net/2014/08/25/index2.html.

[4] 2014-08-22 - NUCLEAR EK FROM 87.117.255.66. http://www.malware-traffic-analysis.net/2014/08/22/index.html.

[5] 2014-09-05 - SWEET ORANGE EK FROM 8.28.175.69. http://www.malware-traffic-analysis.net/2014/09/05/index.html.

[6] 2014-09-11 - SWEET ORANGE EK FROM 87.118.126.94. http://malware-traffic-analysis.net/2014/09/11/index.html.

[7] 2014-09-29 - NUCLEAR EK Delivers Digitally-signed Cryptowall Malware. http://www.malware-traffic-analysis.net/2014/09/29/index.html.

[8] 2014-10-06 - Sweet Orange EK. http://www.malware-traffic-analysis.net/2014/10/06/index2.html.

[9] 2014-11-11 - ANGLER EK USES DIFFERENT OBFUSCATION FOR THE MALWARE PAYLOAD. http://www.malware-traffic-analysis.net/2014/11/11/index.html.

[10] 2016-07-25 - MAGNITUDE EK FROM 51.254.181.39 SENDS CERBER RANSOMWARE. http://www.malware-traffic-analysis.net/2016/07/25/index3.html.

[11] 2016-07-25 - MAGNITUDE EK FROM 51.254.181.39 SENDS CERBER RANSOMWARE. http://www.malware-traffic-analysis.net/2016/07/05/index.html.

[12] 3,000 High-Profile Japanese Sites Hit By Massive Malvertising Campaign. https://blog.trendmicro.com/trendlabs-security-intelligence/3000-high-profile-japanese-sites-hit-by-massive-malvertising-campaign/.

[13] A Different Exploit Angle on Adobe's Recent Zero-Day. https://www.fireeye.com/blog/threat-research/2015/01/a_different_exploit.html.

[14] Angler Exploit kit breaks Referer chain using HTTPS to HTTP redirection. https://hiddencodes.wordpress.com/2015/05/29/angler-exploit-kit-breaks-referer-chain-using-https-to-http-redirection/.

[15] CVE-2013-2551: Buffer overflow in the Pixel Bender parser in Flash Player. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2551.

[16] CVE-2014-0322: An use-after-free vulnerability in IE. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0322.

[17] CVE-2014-0515: Use-after-free involving the CMarkup object in IE. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0515.

[18] CVE-2014-0569: Interger Overflow in Sound.toString() in Flash Player. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0569.

[19] CVE-2015-2419: Double-free in jscript9's `JSON` APIs. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2419.

[20] CVE-2015-5560: Integer overflow in mp3 ID3 tag in Flash Player. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5560.

[21] CVE-2015-8651: Integer overflow in domainMemory in Flash Player. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8651.

[22] CVE-2016-0162: An information-disclosure vulnerability in IE. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0162.

[23] CVE-2016-1019: Type confusion in FileReference class's type checking in Flash. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1019.

[24] CVE-2016-4117: Out-of-bound read in DeleteRangeTimelineOperation in Flash. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4117.

[25] CVE-2016-7200, CVE-2016-7201 (Edge) and Exploit Kits. http://malware.dontneedcoffee.com/2017/01/CVE-2016-7200-7201.html.

[26] CVE-2016-7200: Info leak in Array.filter in Chakra JavaScript engine. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7200.

[27] CVE-2016-7201: Type confusion in FillFromPrototypes in Chakra. https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7201.

[28] DDoS Attacks - A new twist in Malvertisements. http://stopmalvertising.com/malvertisements/ddos-attacks-a-new-twist-in-malvertisements.html.

[29] Malvertisement on FileServe delivers Password Stealer via Exploits. http://stopmalvertising.com/malvertisements/malvertisement-on-fileserve-delivers-password-stealer-via-exploits.html.

[30] Malvertising on Indonesian portal gopego.com delivers Cryptowall 3.0. https://www.cyphort.com/gopego-malvertising-cryptowall/.

[31] Microsoft Patches CVE-2016-3351 Zero-Day, Exploited By AdGholas and GooNky Malvertising Groups. https://www.proofpoint.com/us/threat-insight/post/Microsoft-Patches-Zero-Day-Exploited-By-AdGholas-GooNky-Malvertising.

[32] New wave of malvertising leverages latest Flash exploit. https://blog.malwarebytes.com/cybercrime/2016/05/new-wave-of-malvertising-leverages-latest-flash-exploit/.

[33] Readers of popular websites targeted by stealthy Stegano exploit kit hiding in pixels of malicious ads. http://www.welivesecurity.com/2016/12/06/readers-popular-websites-targeted-stealthy-stegano-exploit-kit-hiding-pixels-malicious-ads/.

[34] Recent example of KaiXin exploit kit. https://isc.sans.edu/forums/diary/Recent+example+of+KaiXin+exploit+kit/20827/.

[35] Security: Referer chain breaking by abusing src-less iframe body context. https://bugs.chromium.org/p/chromium/issues/detail?id=485722.

[36] The Wild Wild Web: YouTube ads serving malware. https://labs.bromium.com/2014/02/21/the-wild-wild-web-youtube-ads-serving-malware/.

[37] Anatomy of a Malware Ad on NYTimes.com. http://troy.yort.com/anatomy-of-a-malware-ad-on-nytimes-com/, 2009.

[38] Three Alleged International Cyber Criminals Responsible For Creating And Distributing Virus. https://www.justice.gov/usao-sdny/pr/three-alleged-international-cyber-criminals-responsible-creating-and-distributing-virus, 2013.

[39] Cyphort Lab, The Rise of Malvertising. http://go.cyphort.com/rs/181-NTN-682/images/Malvertising-Report-15-RP.pdf, 2015.

[40] Eset Research, Readers of popular websites targeted by stealthy Stegano exploit kit hiding in pixels of malicious ads. http://www.welivesecurity.com/2016/12/06/readers-popular-websites-targeted-stealthy-stegano-exploit-kit-hiding-pixels-malicious-ads, 2016.

[41] Mimi An, Why People Block Ads And What It Means for Marketers and Advertisers. https://research.hubspot.com/reports/why-people-block-ads-and-what-it-means-for-marketers-and-advertisers, 2016.

[42] Proofpoint, Massive AdGholas Malvertising Campaigns Use Steganography and File Whitelisting to Hide in Plain Sight. https://www.proofpoint.com/us/threat-insight/post/massive-adgholas-malvertising-campaigns-use-steganography-and-file-whitelisting-to-hide-in-plain-sight, 2016.

[43] The Guardian, Major sites including New York Times and BBC hit by 'ransomware' malvertising. https://www.theguardian.com/technology/2016/mar/16/major-sites-new-york-times-bbc-ransomware-malvertising, 2016.

[44] The Interactive Advertising Bureau (IAB), 2015 Internet Advertising Revenue Full-Year Report. http://www.iab.com/wp-content/uploads/2016/04/IAB_Internet_Advertising_Revenue_Report_FY_2015-final.pdf, 2016.

[45] W3Techs Web Technology Surveys, Usage of advertising networks for websites. https://w3techs.com/technologies/overview/advertising/all, 2016.

[46] Zeljka Zorz, Malvertising campaign hits MSN.com, NY Times, BBC, AOL. https://www.helpnetsecurity.com/2016/03/16/malvertising-campaign/, 2016.

[47] PageFair, 2017 Global Adblock Report. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017.

[48] Florian Alt, Moritz Balz, Stefanie Kristes, Alireza Sahami Shirazi, Julian Mennenöh, Albrecht Schmidt, Hendrik Schröder, and Michael Goedicke. Adaptive User Profiles in Pervasive Advertising Environments. In *Proceedings of the European Conference on Ambient Intelligence*, AmI'09, pages 276–286, Salzburg, Austria, 2009.

[49] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. Adsentry: Comprehensive and Flexible Confinement of Javascript-based Advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC'11, pages 297–306, Orlando, Florida, USA, 2011.

[50] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and Detecting Malicious Flash Advertisements. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC'09, pages 363–372, Honolulu, Hawaii, USA, 2009.

[51] Seyed Hossein Ghotbi and Bernd Fischer. Fine-Grained Role- and Attribute-Based Access Control for Web Applications. In *International Conference on Software and Data Technologies*, ICSOFT'12, pages 171–187, Berlin, Heidelberg, 2012.

[52] Daniel G. Goldstein, Siddharth Suri, R. Preston McAfee, Matthew Ekstrand-Abueg, and Fernando Diaz. The Economic and Cognitive Costs of Annoying Display Advertisements. *Journal of Marketing Research*, 51(6):742–752, 2014.

[53] Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William G. J. Halfond. Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE'15, pages 100–110, Florence, Italy, 2015.

[54] Jiaping Gui, Meiyappan Nagappan, and William G. J. Halfond. What Aspects of Mobile Ads Do Users Care About? An Empirical Study of Mobile In-app Ad Reviews. arXiv:1702.07681, 2017.

[55] Soha Hussein, Patrick Meredith, and Grigore Roşlu. Security-policy Monitoring and Enforcement with JavaMOP. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS'12, pages 3:1–3:11, Beijing, China, 2012.

[56] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS'12, pages 674–686, Raleigh, North Carolina, USA, 2012.

[57] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 24–24, Washington, DC, 2010.

[58] H. Mekky, R. Torres, Z. L. Zhang, S. Saha, and A. Nucci. Detecting Malicious HTTP Redirections Using Trees of User Browsing Activity. In *IEEE Conference on Computer Communications*, INFOCOM'14, pages 1159–1167, Toronto, Canada, 2014.

[59] Tommi Mikkonen and Antero Taivalsaari. The Mashware Challenge: Bridging the Gap Between Web Development and Software Engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER'10, pages 245–250, Santa Fe, New Mexico, USA, 2010.

[60] Prabaharan Poornachandran, N. Balagopal, Soumajit Pal, Aravind Ashok, Prem Sankar, and Manu R. Krishnan. *Demalvertising: A Kernel Approach for Detecting Malwares in Advertising Networks*, pages 215–224. Springer Singapore, Singapore, 2017.

[61] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Shady Paths: Leveraging Surfing Crowds to Detect Malicious Web Pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS'13, pages 133–144, Berlin, Germany, 2013.

[62] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-bot Prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'16, pages 205–216, Seattle, WA, USA, 2016.

[63] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding Malvertising Through Ad-Injecting Browser Extensions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1286–1295, Florence, Italy, 2015.

[64] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC'14, pages 373–380, Vancouver, BC, Canada, 2014.