AdCube: WebVR Ad Fraud and Practical Confinement of Third-Party Ads

Hyunjoo Lee[*], Jiyeon Lee[*], Daejun Kim, Suman Jana[‡], Insik Shin, Sooel Son[†]
School of Computing KAIST, ‡Columbia University

Abstract

Web technology has evolved to offer 360-degree immersive browsing experiences. This new technology, called WebVR, enables virtual reality by rendering a three-dimensional world on an HTML canvas. Unfortunately, there exists no browser-supported way of sharing this canvas between different parties. Assuming an abusive ad service provider who exploits this absence, we present four new ad fraud attack methods. Our user study demonstrates that the success rates of our attacks range from 88.23% to 100%, confirming their effectiveness. To mitigate the presented threats, we propose AdCube, which allows publishers to specify the behaviors of third-party ad code and enforce this specification. We show that AdCube is able to block the presented threats with a small page loading latency of 236 msec and a negligible frame-per-second (FPS) drop for nine WebVR official demo sites.

1 Introduction

WebVR [77] is a JavaScript programming interface that enables virtual reality (VR) presentation in user browsers. It aims to provide an integrated VR environment for different browser platforms and operating systems. WebVR works in tandem with WebGL [31] and leverages a canvas document object model (DOM) to render VR scenes; this canvas becomes a window displaying a VR world.

WebVR websites provide the unique feature of enabling immersive virtual world experiences. Internet surfers who seek diverse and content-rich experiences are attracted to WebVR websites [65, 68]. Considering that advertisers seek opportunities to expose their ads to large audiences, it is natural for them to search for a means to bring promotional content into VR worlds. StateFarm reported a 500% increase in the click-through rate over mobile ads due to their VR ad campaigns [5], demonstrating the potential of VR ads to attract audience attention. Online VR ad service providers, including OmniVirt [52] and Adverty [6], have provided a means for advertisers to expose their products or services in VR worlds.

A standard website often monetizes its content by renting out its screen estates for ads. For this, the website embeds a JavaScript (JS) library from an ad service provider (e.g., Google or Facebook), and this library leverages an iframe element to display ads and confine the execution of their JS scripts. This iframe serves as an execution container such that the hosting website cannot alter ads within the iframe. Unfortunately, in WebVR environments, there are no iframe-like primitives that isolate the execution of an ad-serving JS script; instead, it shares a portion of the displayed VR scene. This WebVR limitation stems from the usage of a canvas DOM to render VR scenes, thus providing no browser-supported method of sharing this canvas between different web origins [51].

Previous research has demonstrated the presence of abusive ad service providers who perpetrate impression or click fraud campaigns [63, 81]. When an ad service provider with ill intent abuses the absence of iframe-like primitives in WebVR environments, there is no practical method for WebVR websites to sandbox the execution of their third-party ad-serving JS scripts. Furthermore, to the best of our knowledge, there is no previous study that investigates security threats imposed by third-party WebVR ads.

Our contributions. Assuming the presence of abusive ad service providers who conduct impression or click fraud, we introduce four new attack variants that leverage unique WebVR features. We present *gaze and controller cursor-jacking* attacks. Gaze and controller cursors are new input channels from head-mounted displays (HMDs) and VR controllers, respectively. These attacks introduce fake gaze and controller cursors into VR scenes and deceive users into clicking promotional VR entities. We then introduce a *blind spot tracking* attack whereby the adversary places promotional objects, images, and videos in the opposite direction of a user's current line of sight. This attack exploits the limited visual awareness of users when they enable 360-degree immersive views. Lastly, we propose an *abuse of an auxiliary display* attack

[*] Both authors contributed equally to the paper
[†] Corresponding author

that exploits the inability of users to view the main display when they enter the immersive mode.

We conducted user studies with 82 participants to measure the efficacy of our attacks. The experimental results show that the gaze and controller cursor-jacking attacks have success rates of 88.23% and 93.75%, respectively, with participants clicking at least two ad entities. The blind spot tracking and abuse of an auxiliary display attacks have success rates of 94.12% and 100%, respectively. These results demonstrate that the adversary is able to readily conduct stealthy ad fraud.

We propose a defense system, AdCube, which is designed to block the four types of attacks presented as well as traditional web threats, including cookie theft [82] and unrestricted private information retrieval [39] by untrustworthy third parties. We define two security requirements to block the presented threats: 1) the visual confinement of three-dimensional (3D) ad entities; and 2) the sandboxing of ad-serving JS scripts according to a given security policy. To address the first requirement, we propose an algorithm confining ad objects as well as 3D models to bounding boxes, called *adcube*. To address the second requirement, we leverage Caja [19], a mature sandboxing technology maintained by Google, to confine the execution of third-party JS code. Specifically, on top of Caja, we design a set of JS APIs that an ad-serving JS script is able to use to create WebVR ads and implement each API. Therefore, a benign WebVR website owner is able to use AdCube to confine the locations and executions of VR ads as the owner specifies. For open science and further research, we have released AdCube at `https://github.com/WSP-LAB/AdCube`.

We evaluated the performance of AdCube in terms of page loading time and frames-per-second (FPS). Compared to the baseline without any defense, AdCube produced a negligible FPS drop when rendering a complex demo site of a virtual art museum and an additional page loading time of 236 msec on average when rendering nine WebVR sites, thus demonstrating the promising efficacy of AdCube in the wild.

2 Background

2.1 WebVR

VR technology offers an immersive user experience that provides users with a virtual 3D world. Rendering a virtual world scene entails heavy usage of matrix computations, high demand for graphics processor unit (GPU) resources, and the frequent loading of large-sized graphic textures and images. These requirements make native applications the only viable means of delivering a VR world. However, the proper installation and frequent software updates, which native applications often require, have hindered their wide adoption.

The advent of WebVR addresses these core limitations. This new technology enables a website to offer a VR environment by means of browser supports. WebVR specifies a set of browser-supported APIs that enables VR in user browsers [77]. It provides interfaces for managing VR peripherals, such as HMDs and VR controllers, thus enabling an immersive 3D world experience. WebVR works in tandem with WebGL [31] to render VR content on an HTML5 canvas DOM element. WebGL provides a set of interfaces that launch shader programs as well as manage viewports, thus rendering sophisticated 3D entities and models via a large volume of matrix computations empowered by GPUs.

In 2018, WebVR was integrated into WebXR [78], which is designed for both augmented reality (AR) [43] and virtual reality (VR) on the Web. However, the original architecture of WebVR remains the same in WebXR, with only keyword changes. In this paper, we focus on addressing new security threats that involve WebVR APIs in WebXR.

WebVR terminology. Here we clarify WebVR terms that we use throughout the paper. A *VR scene* refers to a view of a VR world in WebVR. In this definition, a scene requires a viewer of the VR world. A *camera* refers to this viewer, usually represented by the perspective of a user. The *immersive mode* refers to the mode in which a user sees the scene through an HMD. A *viewport* defines a rectangular area where the VR world is rendered. Most WebVR sites offer two viewports onto their VR world; these viewports correspond to the left and right eyes, respectively. An *entity* refers to a visible or invisible object within a scene. To avoid confusion with DOMs and JS run-time objects, we explicitly use the term *entity* to describe objects placed in a VR world. Therefore, an entity that promotes a commercial product is called either an *ad entity* or a *promotional entity*.

3D library. To facilitate usages of WebGL, many JS 3D libraries, including Three.js [71], babylon.js, and React 360, have been proposed. Several vendors have even promoted new WebVR frameworks (e.g., A-Frame [1], PlayCanvas [53], and Sketchfab [59]), which not only provide intuitive interfaces but also establish their own abstraction layers to ease the implementation of rich VR experiences.

A-Frame [1] is a representative WebVR framework introduced by Mozilla in 2015. Its striking feature is that a scene and all entities rendered within the scene can be defined through a markup language, which is accessible via a DOM [46]. For instance, a developer can create a 3D box entity by defining an `<a-box>` tag in HTML and query this entity via JS DOM APIs. This intuitive approach to encoding diverse 3D entity properties into HTML tag attributes has lowered the technical barriers to developing VR content.

2.2 Online Advertising

There exist three main types of participants in the web ad ecosystem: publishers, ad service providers, and advertisers. Figure 1 depicts how these three participants interact with one another. Publishers are website owners or operators who serve informative, promotional, or intriguing content to their website visitors. Advertisers play a role in planning and bid-

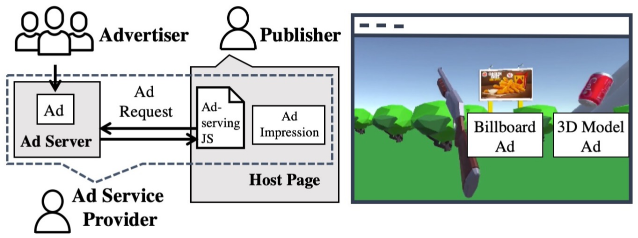


Figure 1: Simplified overview of the web ad ecosystem and examples of OmniVirt VR ads: A billboard ad and a promotional 3D model in a VR scene.

ding on ad campaigns and want to expose those ad campaigns to users who visit publisher websites. Ad service providers connect these publishers and advertisers; they provide publishers with ad-serving JS APIs, which fetch banner, text, and even video ads provided by advertisers.

Web ads have been a prevailing method by which publishers monetize their content. As advertisers seek diverse channels and responsive interactions with their audiences, ad technology has evolved to support not only text banner ads but also various multimedia delivery mechanisms, such as video and native responsive ads integrated into their hosting websites [61, 81]. For instance, news feed ads blended with other non-ad feeds have become a popular ad technology for social media platforms, including Facebook [15, 81].

VR ad market. The VR market was valued at USD 7.3 billion in 2018 and is expected to reach 20.5 billion by 2026 [79]. The total number of active VR users was approximately 171 million as of 2018 [64]. Thus, it is natural for advertisers to seek new opportunities to promote their products or services in a content-rich VR environment, thereby reaching a large number of VR users.

There exist at least 13 VR ad service providers offering VR ad forms; OmniVirt [52] and Wonderleap [80] have supported options to initiate WebVR ad campaigns. OmniVirt reported 100 million and 1 billion delivered VR/AR ad impressions in 2017 and 2018, respectively [73], which demonstrates the surging demand for VR ads.

Ad fraud. Ad fraud refers to an operation that generates unintended ad traffic involving ad impressions or clicks. Previous studies have described various adversarial models that address ill-intentioned publishers committing click fraud [18, 28], malicious extension replacing ads [69], and abusive ad providers generating unwanted ad traffic [63, 81].

In this paper, we assume an abusive ad provider whose objective is to increase ad traffic that fetches ad impressions or generates click events via deceptive techniques. To the best of our knowledge, there have been no previous fraud studies involving WebVR ads.

3 Motive and Threat Model

A typical way of placing web ads is for publishers to copy and paste an ad bootstrapping JS script on their websites. This embedded JS code, which runs with the same origin as its host webpage, creates an iframe [48] of the page that is fetched from a third-party ad service provider. Because the publisher origin differs from the origin of the embedded iframe, the JS script in this iframe can neither alter nor read resources from the hosting page due to browsers enforcing the Same Origin Policy (SOP) [51]. The SOP ensures that the rest of the ad script confined within this iframe is isolated from the hosting page. Thus, publishers only need to check how the embedded bootstrapping JS code performs to prevent potential abuses by advertisers or ad service providers.

Problem. Today's WebVR does not provide an origin separation mechanism that allows a third-party ad script to securely share the same origin as its hosting page to render ad entities, images, or videos within the VR content of the hosting page. This limitation stems from the usage of a canvas element [47] when rendering VR content, which does not provide a way of sharing this element among different origins. This limitation leaves no option for WebVR ad service providers except to run their ad scripts with the origins of hosting pages. Consequently, it is imperative that publishers completely trust these ad service providers.

Unfortunately, previous studies have demonstrated the presence of abusive or malicious ad service providers that victimize visitors to publisher websites [28, 63, 69]. For instance, Springborn *et al.* [63] investigated abusive pay-per-view networks that expose fraudulent impressions via pop-under or invisible ads to increase the number of served ad impressions. Given that an abusive ad service provider is capable of running scripts using its hosting origin, she is able to conduct clickjacking [8, 28], steal cookies [82], and even access the private information of users [39]. However, no previous study has addressed the unique risks entailed in WebVR. Considering that WebVR introduced an immersive mode, in what ways does this paradigm shift favor the attacker?

Sandboxing. Previous studies have investigated how to sandbox the execution of third-party ad scripts within the same origin as the hosting page [2, 7, 22, 30, 42, 45, 57, 72]. Such sandboxing methods are viable as they require low overhead, which is a key requirement in WebVR environments demanding a robust FPS rate. However, it is not clear how to apply these existing techniques to confine WebVR ad scripts.

What are the security properties that the sandbox technique should guarantee? Which API should the sandbox technique provide to support VR features while achieving security requirements? These questions drive our research into providing a practical method of confining ad scripts in WebVR websites.

Threat model. We assume an *abusive ad service provider* who serves 2D/3D ads into WebVR sites. In this scenario, the business imperative is to expose promotional VR entities, images, or videos in the VR worlds of publishers. At the same time, the goal of the adversary is to increase ad traffic by rendering more promotional entities and to generate user clicks via deceptive techniques that increase ad revenue. We emphasize that this adversary model is a real threat. There exist numerous malicious secondary or tertiary ad service providers whose sole motive is to maximize their short-term profit [32, 44]. Adf.ly was a notorious abusive ad service provider that modified the link addresses of publisher pages and tricked users into clicking ads [81].

Considering that there exists no practical way of separating origins that share the same canvas that renders VR scenes, we assume that the adversary places her ad-rendering code at the hosting page, which allows the code to access any resources that belong to the hosting page. The adversary victimizes publishers by abusing their website visitors; her ad-serving JS script generates ad fraud traffic by victimizing visitors. These publishers also lose visitors due to providing bad user experiences with fraudulent ads. An advertiser also becomes a major victim who is obliged to pay for those fraudulent ad impressions and clicks.

4 Attacks

In this section, we present four new ad fraud attacks that exploit blind spots and new VR peripherals.

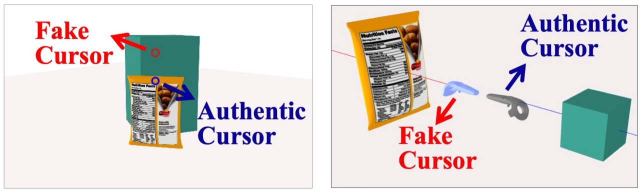
4.1 Cursor-Jacking Attack

Facilitating WebVR experiences requires two representative IO devices: an HMD and a VR controller. These devices introduce two new input channels: a gaze cursor and a controller cursor, which did not exist in a standard web environment.

Unfortunately, both of these input methods can be altered by a JS script, allowing a malicious ad service provider to control them. Thus, the adversary abuses this capability by creating a fake input source to induce actual clicks on other entities. Specifically, we introduce two attack vectors: gaze and controller cursor-jacking attacks.

Gaze cursor-jacking attack. A gaze cursor is a marker that represents the focal point at which a user looks in a VR scene. Usually, a gaze cursor has a circular appearance, which helps users realize what they are looking at. This gaze cursor supports a *fusing* event that fires when a user locates the cursor on a targeted entity. When the gaze cursor stays on this target entity for 1.5 seconds (default), a browser then fires a *click* event. Thus, the gaze cursor provides a unique way of triggering a "click" event on an entity without involving any mouse or controller events.

Gaze cursor-jacking (GCJ) refers to an attack that creates a fake gaze cursor and hides the original cursor in a target VR scene. This GCJ attack leads its victims to believe that a fake cursor is actually an authentic input cursor and to place the "authentic" cursor at a point where the attacker wants it to be.



(a) Gaze cursor-jacking **(b)** Controller cursor-jacking

Figure 2: Illustration of (a) gaze and (b) controller cursor-jacking attacks: (a) When a user clicks a UI button via the gaze cursor made by the attacker, the authentic cursor clicks the ad. (b) Inserting a fake controller cursor by rotating its z-axis by 180 degrees. When a user clicks the green box with an authentic VR controller cursor, the ad placed in the opposite direction is also clicked.

Figure 2a demonstrates the implementation of the attack in an A-Frame environment. The attacker is able to make the authentic gaze cursor invisible and insert a fake gaze cursor that triggers click events on different entities placed near the position where the authentic cursor is located. Thus, she is able to hijack authentic clicks that should be attributed to first-party content.

Controller cursor-jacking attack. A VR controller is another input device that enables a user to trigger various events on entities, such as clicks. Usually, a VR site shows a projection line that points to a target, which varies according to the user's controller direction. A user leverages this projection line in a scene to select a target entity at which the user fires various events.

Controller cursor-jacking (CCJ) is an attack that introduces an additional fake VR controller cursor in a target VR scene. When a victim generates a user event on an entity, the same event is also triggered at the target entity that this fake cursor indicates because this fake cursor shares user events with the original controller cursor. The adversary is able to leverage blind spots to hide fake controller cursors and induce clicks on ads whenever a click occurs (Figure 2b).

In a standard web page, a clickjacking attack [28] performs a similar attack by using another iframe window from a third-party source that actually tricks victims into clicking a target element underneath this iframe window. However, the two attacks presented here differ from the clickjacking attack in that they do not exploit third-party windows due to the WebVR nature of sharing the same scene between first- and third-party scripts. Furthermore, these two attacks abuse new input vectors that only exist in a WebVR environment.

Considering the adversary is already able to fire click events via dispatchEvent API invocations, she might not need to induce genuine user clicks with these two attacks to achieve her goal. However, in Chrome, Edge, and Oculus Android browsers, only event handlers invoked via genuine click events are able to open a new window or cause redirection to a different website. Because the goal of the adversary

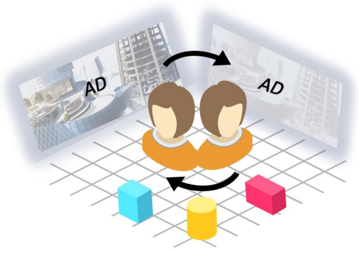


Figure 3: An illustration of blind spot tracking ads.

is eventually to redirect victims to an ad-landing page, the attacker has a clear motive to conduct GCJ and CCJ attacks.

4.2 Blind Spot Attack

A WebVR site offers surrounding 360-degree views through the support of an HMD, thereby enabling a new kind of browsing experience. This results in two types of blind spots that users are unable to see when experiencing the immersive mode with the HMD: 1) one is located in the direction opposite a user's current line of sight, and 2) the other is the main display, such as a desktop monitor or a laptop display, which becomes an auxiliary display when users wear the HMD. The attack is able to place promotional entities in these blind spots, which are invisible to users. We introduce two attack vectors: blind-spot tracking and abuse of an auxiliary display.

Blind spot tracking attack. A blind spot tracking (BST) attack occurs when the adversary hides an ad entity in the opposite direction from the user's current line of sight. She can also move this entity into blind spots whenever the user's gaze changes direction by tracking the camera sight's direction (Figure 3). Thus, the adversary is able to increase the number of rendered ad impressions or entities, later charging the respective advertisers for this inflated number of ad views.

The BST attack is a unique variant of ad impression fraud. Ad impression fraud refers to an operation that (1) hides rendered ads underneath other UI elements, (2) makes ads invisible by making them too small, (3) places ads to appear when a user scrolls down a webpage, or (4) simply renders a vast volume of ad impressions [36, 37, 67]. On the other hand, the proposed BST attack leverages blind spots that are inherent in any VR content.

Abuse of an auxiliary display attack. An attacker can abuse the user's limited awareness of the browser on the auxiliary display by displaying diverse ad impressions or videos to maximize ad view counts. We call this attack an abuse of an auxiliary display (AAD) attack.

Furthermore, the attacker can identify the moment when a victim exits the immersive mode when a `vrdisplaypresentchange` event is fired or when the HMD device is taken off; this is achieved by monitoring abrupt gaze cursor changes or scene change events. When identifying such moments, the attacker can remove all ad impressions and stop video ads on the auxiliary display involved in stealth ad campaigns.

5 User Study

To measure the efficacy of the presented attacks (§4), we recruited 82 participants and investigated their responses to the four attack scenarios. This section describes our user study designs (§5.1) and experimental results (§5.2).

5.1 Experimental Design

From July to October 2019, we recruited a total of 82 university students, consisting of 52 males and 30 females (mean age = 23.69). Among them, 49 had been exposed to VR experiences before. The participants were offered $5 per attack scenario, each of which took approximately 30 minutes to complete. We obtained IRB approvals and consent from every participant. We focused on demonstrating the feasibility of each attack rather than proving its success on general audiences. For participants, we thus targeted primary consumers of VR content, whose ages were between 19 and 30 [66].

Each participant was randomly assigned to one of four attack scenarios. For each attack scenario, we prepared two webpages: one represented the normal case without any attacks (control group); the other was implemented for the corresponding attack. We used a within-subject design; all participants experienced both normal and attack tests in each scenario. To minimize the learning effect, whereby a prior user study experience affects metrics observed during a posterior user study, we shuffled the order of normal and attack cases for each participant, ensuring that the same number of users initially experienced normal and attack cases.

While exploring the two webpages described above, the participants were asked to complete a specific task for each page. At the end of each task, they were asked to complete a survey asking about their awareness of the existence of rendered ads and the differences between the normal webpages and those under attack. The questionnaires are described in detail in Appendix A.1.

We reserved a spacious classroom for the participants to browse the VR websites and prepared a Windows 10 host with an HTC VIVE device. We instructed the participants not to interact with ads and notified them that any clicks on promotional entities would be considered interactions. We gave explicit guidance to the participants that they did not need to interact with any ad entities to finish a given task.

5.2 Experimental Results

5.2.1 Gaze Cursor-Jacking Attack

We used halloVReen [24], a game of finding hidden animation objects, to test the efficacy of a GCJ attack. The participants

Table 1: Experimental results for participants who experienced GCJ and CCJ attacks.

Attack Scenario	Treatment Group	Total	Awareness of Ads			Authentic Clicks			Forged Clicks (Attack Success)		
			at least one	half*	all	at least one	half*	all	at least one	half*	all
GCJ	Normal	17	17	17	5	11	4	0	N/A†	N/A†	N/A†
	Attack	17	17	16	11	6	5	1	17 (100%)	15 (88.23%)	0 (0%)
CCJ	Normal	16	6	2	1	0	0	0	N/A†	N/A†	N/A†
	Attack	16	6	2	2	0	0	0	16 (100%)	15 (93.75%)	6 (37.5%)

*Half indicates at least three out of seven ad entities for GCJ and at least two out of three ad entities for CCJ.
†In the normal case, the attack is not carried out, so the result is shown as N/A.

were expected to find five animated Halloween ghosts scattered across a VR scene via gaze-clicks. The task was to end after five minutes, regardless of the completion of a given task.

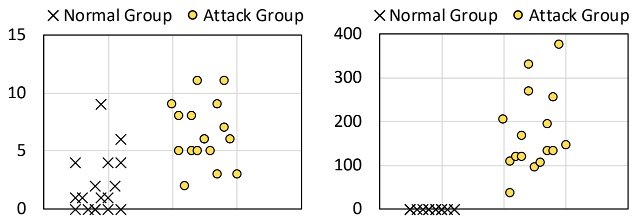
For the normal webpage without the attack, we placed seven ad entities placed near Halloween figures. For the attack page with the GCJ attack, we placed seven different ad entities near Halloween figures. We also created a fake gaze cursor near the actual cursor and made the actual cursor invisible. To minimize the learning effect, we used different Halloween figures and ad entities for each webpage.

When participants gaze-clicked the fake cursor on the Halloween figures, the ad entities were gaze-clicked by the actual cursor. Because a gaze click event is triggered when the cursor stays on a target for at least 1.5 seconds, non-intentional head movements could not have accounted for any of the gaze-clicks. In other words, all counted gaze clicks originate from either users' intentional clicks or the GCJ attack. After a given task, participants were asked on the survey to check which ad objects they had found and whether they had clicked any of them.

Table 1 shows the experimental results. The columns below Awareness of Ads represent the number of participants who noticed the existence of ads. The sub-columns at least one, half, and all represent the number of participants who recognized at least one, half, and all of the promotional entities, respectively. The columns below Authentic Clicks show the number of participants who intentionally gaze-clicked promotional entities. Also, the Forged Clicks columns represent the number of participants who gaze-clicked promotional entities with the real gaze cursor due to this attack.

As the first row in Table 1 shows, all 17 participants who browsed the normal webpage discovered at least three ad entities, which is about half of the seven ad entities that we placed in the scene. Interestingly, whereas the instructions were given to avoid clicking on promotional entities, 11 and 6 people in the normal and attack cases, respectively, intentionally clicked on at least one ad entity.

As the second row in the table shows, every participant gaze-clicked at least one ad entity due to the GCJ attack, which is a significant improvement over the six participants who intentionally gaze-clicked at least one ad entity in the attack case. Also, 15 participants (88.23%) gaze-clicked at



(a) GCJ (b) CCJ

Figure 4: Total number of participants' clicks on all ads in the GCJ and CCJ attack scenarios.

least three ad entities due to the attack; this means that the emplaced attack caused a majority of the participants to gaze-click ad entities.

Figure 4a shows the total number of gaze-clicks on all ad entities for each participant. The normal group represents the number of intentional clicks on ads in the normal case, while the attack group represents how many gaze-clicks were fired due to the attack. The mean of the clicks due to the attack is 6.51, which is three times greater than 2.61, which is the mean of clicks in the normal case. This statistic demonstrates that the adversary can generate more gaze-clicks on ads than in the normal case by exploiting a GCJ attack.

5.2.2 Controller Cursor-Jacking Attack

For the second user study, we used A-Blast [76], a shooting game in which players shoot flying monsters with two blaster guns maneuvered by two VR controllers. Monsters appear randomly within 120 degrees of the front. Each participant played a game for five minutes, or the game ended early when his/her avatar died.

We prepared two webpages. The normal page implemented the A-Blast game, in which three ad entities were placed in positions where the participants would hardly ever look, which was about 180 degrees away from the front. The attack page had the same ads placed in the same location as on the first page. It implemented the CCJ attack, in which a fake controller was inserted that rotated the z-axis 180 degrees. Thus, the participants unwittingly clicked the back of the scene when they shot at monsters in front of their sights, thus clicking ad entities.

The second row of Attack Scenario in Table 1 summarizes the experimental results. It shows that six of 16 participants were aware of at least one ad entity in both cases although they could not see them unless they turned their line of sight around 180 degrees. Also, no participants intentionally clicked ad entities because they were located in the opposite direction of the front area where the game was taking place.

Due to the attack, every participant (100%) clicked at least one ad entity. Also, 15 (93.75%) and 6 (37.5%) participants clicked at least two and all of the ad entities, respectively. This demonstrates that no one intentionally clicked these ad entities but that the attack caused participants to click them.

Figure 4b shows the total number of clicks on all ad entities for each participant. These results show that there were no clicks in the normal case. On the other hand, the mean of ad clicks in the attack case was 174.31. This unbalanced metric indicates the effectiveness of the CCJ attack.

5.2.3 Blind Spot Tracking Attack

We revised two VR game websites to implement a blind spot tracking attack: halloVReen [24] and Whack-a-mole [56]. The Whack-a-mole game is designed such that users attempt to grab moles, which appear in the 360-degree scene, via user gaze-click. We asked the participants to finish the games in one minute.

The attack was implemented by placing a video ad for which z-order was set to the behind camera position, thus rendering the video ad at a blind spot of the participant. Considering that a typical video ad is accompanied by music and sound effects, we also tested the degree to which ad sounds enhanced the participants' awareness of ads in their blind spots. The participants were asked to wear earphones connected to the HMD supporting 3D spatialized sounds. Note that the 3D spatialized sounds only reflect the distance from a sound source and not the direction. Regardless of whether sounds were played in the front of or behind the participants in our VR worlds in A-Frame, the participants heard the identical sounds.

We chose two ad videos that advertise a popular supermarket and drink product. They had been well-received by university students due to a heavy volume of commercial marketing. Thus, the participants were highly likely to recognize these brands by just hearing the sounds of these ad videos.

For the user study, we prepared two treatment groups. One group consisted of 17 participants who experienced two VR websites: Whack-a-mole for the normal case and halloVReen for the attack case with the sound enabled. The other group, consisting of 15 participants, experienced two VR websites: halloVReen for the normal case and Whack-a-mole for the attack case with the sound muted.

Table 2 presents the experimental results for each treatment group. Of the 32 participants who experienced the normal sites that rendered no ads, only three participants (9.375%) claimed that they heard ad sounds, which were actually the sound effects of the underlying websites. Of the 15 participants who experienced the attack site with the sound muted, only two (13.3%) claimed awareness of ad sounds, which were actually noises in the experimental environment, such as desk-dragging sounds. That is, no one heard genuine ad sounds in the normal and attack cases with the sound muted. In contrast, 14 participants (82.35%) who experienced the attack site with ad sounds claimed that they indeed heard ad sounds and became aware of the presence of ongoing ad campaigns.

Table 2: Experimental results for participants who experienced the blind spot tracking (BST) and the abuse of an auxiliary display (AAD) attacks.

Attack Scenario	Treatment Group	Total	Awareness of the Ads	Found ads (Attack Success)
BST	normal	32	3	0 (N/A)
	attack (muted)	15	2	0 (100%)
	attack (w/ sound)	17	14	1 (94.12%)
AAD	normal	32	3	0 (N/A)
	attack (muted)	17	1	0 (100%)
	attack (w/ sound)	15	15	0 (100%)

Note: The *Awareness of the Ads* column indicates that the number of participants who realized the presence of ads. The *Found ads* column shows the number of participants who actually saw the ads.

Note that no one in the attack group saw the ad video in the muted attack, and only one participant claimed that he saw an ad video in the sound attack. Considering that this participant could not specify the ad video he saw, we concluded that he did not see any video ad playing in the opposite direction of his line of sight. We concluded that the BST attack is capable of concealing ad impressions and videos, rendering users unable to recognize whether ads are rendered.

5.2.4 Abuse of an Auxiliary Display Attack

We implemented an AAD attack on the A-Blast website [76]. Each participant played a game for five minutes. The attack created an iframe that rendered a video ad on the A-Blast webpage in the original desktop display when a participant entered the immersive mode. The attack also deleted this iframe when a participant exited the immersive mode. Therefore, it was improbable for participants to find such ads unless they took off the HMD device before finishing the task.

We also measured the effects of ad sounds to measure the participants' awareness of the ads rendered on their auxiliary display, which was the desktop monitor used in this user study. For the user study, we designed two treatment groups. One group consisted of 17 participants, and they experienced the A-Blast website for the normal case and the same website for the attack case with the sound muted. The other group, which consisted of 15 participants, visited the same A-Blast website for the normal and attack cases with enabled sound. For the ad videos rendered, we chose two videos that advertise a popular e-commerce site and a vitamin drink product.

Table 2 presents the experimental results for each treatment group. Only three (9.375%) of the 32 participants who experienced the normal site and one (5.882%) of the 17 participants who experienced the muted attack site claimed hearing ad sounds; however, they were the sound effects of the underlying websites. On the other hand, all (100%) of the 15 participants who experienced the attack site with sound were aware of the presence of ongoing ads due to the video ad sounds. However, note that no one explicitly found the ad video, thus demonstrating the feasibility of abusing this attack in a stealthy manner.

6 AdCube

This section explains two security requirements to mitigate the presented attacks and a defense model of AdCube (§6.1). We then present the architecture of AdCube (§6.2) and its usage in terms of defining security policies (§6.3). Lastly, we explain how AdCube is implemented to enforce the aforementioned security requirements (§6.4 and §6.5).

6.1 Defense Model

We list security requirements that a new defense model should have in order to prevent the four proposed attacks as well as traditional threats [38, 39, 82].

1. Third-party JS code should place ad entities only within the confined area that the first party specifies, and these entities should fit within this area.

2. Third-party JS code should not be able to alter DOM elements and sensitive entities (e.g., camera and controller) if the first party does not permit doing so.

The first requirement aims to block the BST attack and any abusive attempts to place a prohibitive number of ad entities all over the VR scene of a publisher. The second condition is required to block the GCJ, CCJ, and AAD attacks, thereby limiting the adversary's capability of changing gaze cursors, VR controller cursors, and DOM elements belonging to the first party. Note that the defense system in the second requirement also prevents malicious third-party scripts from gaining unrestricted access to sensitive information, such as credential cookies and private information [39, 82].

Previous studies have addressed the second requirement by confining the execution of third-party code [7, 30, 38, 54, 57, 58]. These approaches are categorized according to two objectives: 1) origin-based isolation and 2) code sandboxing. The origin-based isolation refers to a technique that assigns each embedded third-party code with a separate origin (or process) so that SOP (or process isolation by OS) forces the confinement of the third-party code. In contrast, code sandboxing enforces third-party code to interact with its host via specified APIs while sharing the same origin with its host.

Unfortunately, origin-based isolation techniques, including AdJail [38] and AdSplit [58], often demand a heavy volume of cross-origin or process communications, which enable the separate origins of third-party codes to operate as a single app. Such a large volume of communications introduces execution latency, thus impeding a stable frame rate, which undermines rich user WebVR experiences. On the other hand, previous studies of code sandboxing have not explored the confinement of a third-party script in a WebVR environment [7, 54, 57].

To this end, we propose *AdCube*, a client-side defense solution that addresses the aforementioned two security requirements. The defense is designed for benign publishers who wish to prevent third-party scripts from accessing and modifying the host page's DOM elements and VR entities. For the first requirement, AdCube provides a hexahedron, called an adcube, which visually confines the ads. The publishers specify its position to indicate where an ad entity should be rendered. To address the second requirement, AdCube sandboxes a given third-party JS script while providing a limited set of APIs which the third-party codes use to render WebVR ads. Also, it allows the publishers to set a security policy, which defines how specified third-party scripts should interact with host elements. Therefore, the ad service providers should implement their ad-serving scripts in AdCube APIs. To enable AdCube, the publisher embeds an AdCube JS library in their host script.

Publisher's motives. Considering that ad fraud campaigns may not only benefit the adversary but also publishers in the adversary's ad network via inflated numbers of impressions and clicks, the following question arises: *What would motivate publishers to use AdCube?*

Note that an abusive ad service provider may harness the absence of visual confinement of WebVR ads. The adversary emplaces an enormous number of ad entities that visually block the VR content of a publisher, thereby diverting visitors' attention to the invasive ads [27], which conflicts with the publisher's intention. Furthermore, this service can also place eye-grabbing promotional entities that block first-party promotional entities, conducting occlusion attacks [33]. These invasive or spammy ads can eventually contribute to visitors avoiding publisher websites [12].

The FTC states that publishers are responsible for substantiating whether deceptive ads are present [17]. They examine whether publishers have known or participated in serving deceptive and invasive ads. Google penalizes the search rankings of publishers with invasive ads [14]. We believe that these trends necessitate the adoption of AdCube by publishers.

Furthermore, it is known that security vulnerabilities, including cross-site scripting bugs, often arise from third-party JS code [45, 57, 62, 82]. AdCube is able to isolate third-party JS code, thereby preventing the adversary from harming the customer via exploiting security vulnerabilities.

6.2 Architecture

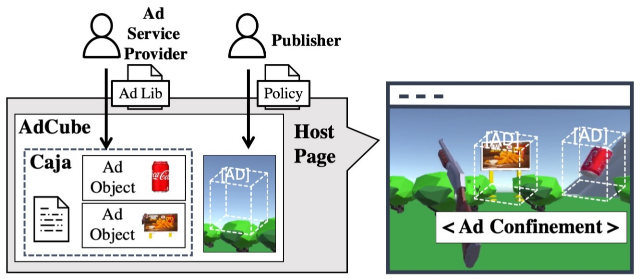


Figure 5: AdCube overview.

The overall architecture of AdCube is demonstrated in Figure 5. AdCube is a JavaScript library, designed to confine the execution of third-party scripts rendering WebVR ads. A publisher furnishes this JS library with a given security policy that specifies how a third-party script should interact with the resources belonging to the first-party origin. The publisher then embeds a third-party JS ad library in Secure ECMAScript [20], which fetches and renders WebVR ads.

To sandbox this embedded JS ad library, AdCube leverages Caja [19], a seminal sandbox framework from Google. We chose Caja from among the previous studies [2, 7, 22, 30, 42, 45, 57, 72] because it is open source software that has been well-managed for over 10 years. Caja sandboxes the execution of guest code from its host page so that the guest code is only accessible to defensive objects that the host page allows. Caja achieves this sandboxing via dynamically monitoring the execution of transformed guest code. Caja conducts *cajoling* of the original guest code into a transformed version, which adds inline checks to enforce invariants that Caja requires.

By design, Caja guarantees no free variables. Thus, the only way for guest code to modify JS objects or DOMs in the host page is to use the references of defensive objects that the host page explicitly offers. Furthermore, the host page is able to enforce customized access control checks on these defensive objects because the host page can revise the APIs that the guest code uses to access defensive objects.

Therefore, the sandboxing of a third-party ad JS script is enforced by Caja. AdCube is a set of wrapper Caja APIs. For publishers, AdCube offers a security policy language. For ad service providers, AdCube offers a set of JS APIs that enable the programming of WebVR ads while interacting with VR entities in host pages.

Listing 1 shows an example of applying AdCube to an A-Frame host page. To enable AdCube, a publisher includes *adcube.js* at Line (Ln) 2. Also, the publisher defines an advertising cube at Ln 9 where a third-party ad-serving script places VR advertising entities. The publisher is also able to specify a security policy that defines which host elements a third-party ad script interacts with. As Lns 5-6 indicate, the third-party is able to *read* the properties of the a-box DOM object and to *write* the properties of the a-sphere object. At

Listing 1 **An example of A-Frame host page with AdCube**

```
 1: <body>
 2:   <script src='adcube.js'></script>
 3:   <a-scene>
 4:     <!-- part of the host app -->
 5:     <a-box can-read></a-box>
 6:     <a-sphere can-write></a-sphere>
 7:     ...
 8:     <!-- a new definition for ad -->
 9:     <a-adcube position='0 0 -2' width='2' height
        ='2' depth='2'></a-adcube>
10:   </a-scene>
11:   <script>
12:     const adcube = AdCube();
13:     adcube.load('https://3rdparty.com/ad.js');
14:   </script>
15: </body>
```

last, the ad script embedded at Ln 13 runs in a Caja-enforced sandbox with limited access to the a-box and a-sphere entities. That is, this *load* invocation specifies third-party scripts that should be sandboxed via AdCube.

6.3 AdCube and Security Policy

AdCube asks a publisher to specify two types of specifications: 1) an `adcube` primitive that specifies a third-party ad rendering space in the VR world of a host page; and 2) a security policy that specifies DOMs that interact with a confined third-party ad script.

AdCube primitive. An `<a-adcube>` tag defines an AdCube primitive for A-Frame enabled web pages. It specifies a hexahedron in which to render WebVR ads. This adcube tag has four properties. The `position` property specifies a hexahedron position in the VR world of a host page. The `width`, `height`, and `depth` define the size of this hexahedron. When this `<a-adcube>` tag is placed as a child of a host element, AdCube internally sets the `parent` of the `adcube` to be this host element, and the location of the `adcube` is relative to this parent element. For instance, when specifying the parent element of an `adcube` primitive to be a camera entity, this `adcube` moves as the camera angle of the scene changes.

Security policy. A publisher with AdCube is able to specify access control policies regarding which host entities and DOMs are *readable* or *writable* by a third-party script that AdCube sandboxes. Specifically, the publisher assigns a `can-read` or `can-write` attribute to an A-Frame entity or a DOM. AdCube stores this labeled entity or DOM in a JS object, called `TamedDOM`, which AdCube lets a third-party script access or revise via the `querySelector` API. That is, `TamedDOM` becomes a bridge between the host and a sandboxed third-party script. AdCube implements this functionality by leveraging the `markfunction` API of Caja.

By default, AdCube prohibits a sandboxed third-party script from accessing any entities or DOMs in the host page. This default policy blocks all the attacks (§4) by preventing

Table 3: An API list for advertising.

Creation
`createElement(`*[tag name|URL]*`)`
Creates a new entity and returns the *entity's interfaces* defined by AdCube
`addElement(`*adcube_id, entity*`)`
Appends an *entity* to the adcube which has the *adcube_id*.
Set
`entity.setAttribute(`*key, value*`)`
Sets an `entity`'s attribute with *key* and *value*
`entity.appendChild(`*child entity*`)`
Appends a *child entity* to the `entity` as its children
`entity.addEventListener(`*event name, function*`)`
Sets an `entity`'s event handler with *event name* and *function*
Get
`entity.getAttribute(`*key*`)`
Returns an `entity`'s attribute corresponding to the *key*
`querySelector(`*tag name|ID*`)`
Returns an *entity* corresponding to the *tag name* or *ID*
`querySelectorAll(`*tag name|ID*`)`
Returns *multiple entities* corresponding to the *tag name* or *ID*

a third-party script from accessing cameras, gaze/controller cursors, and DOMs whose origin is bounded by the host origin. Moreover, this default policy significantly lightens the burden of specifying a proper security policy for publishers.

Furthermore, AdCube attaches an "[AD]" label at the top of a defined `adcube` area, as shown in Figure 6, thus making VR ad content visually distinguishable from host VR entities. In this way, publishers are able to help their visitors easily identify which entities are for ads, which the IAB has been recommending for healthy ad ecosystems [29].

6.4 Ad Service APIs

AdCube sandboxes a third-party script by providing a confined execution environment with predefined objects and APIs. We designed a set of APIs that a third-party ad serving script is able to use to implement VR content. Instead of defining a long list of all possible APIs, we focused on defining essential APIs for the AdCube prototype. Table 3 shows the API list. We designed our APIs similar to JavaScript DOM APIs [46] to make them compatible with common software engineering practices among JS developers.

Caja does not allow any direct access to host DOM elements from Caja's guest context. Thus, AdCube creates a custom JS object called `TamedDOM` that contains the APIs presented in Table 3. Any API invocations other than defined APIs result in an execution error.

Listing 2 is a third-party ad-serving script example that implements VR content. The code creates an ad entity via `createElement()`, which is yet to be added to the scene. By leveraging the returned entity reference, the code sets the attribute that specifies the URL source of a 3D model and attaches a click event handler that causes the model to animate. The invocation of `addElement()` appends this entity to the `adcube` that the host page defines via the `<a-adcube>` tag. Note that this `addElement()` could be an injection channel to insert DOMs and entities furnished with malicious JS code in their event handlers. Thus, AdCube implements filters that allow appending only A-Frame objects (e.g., `<a-gltf-model>` and `<a-obj-model>`) and forbid altering sensitive sink properties (e.g., `Element.innerHTML` and `Element.insertAdjacentHTML`) [45].

Listing 2 An example of ad-serving JS script

```
let e = createElement('a-gltf-model');
e.setAttribute('src', 'product.gltf');
e.addEventListener('click',onClick);
addElement('adcube-id', e);
function onClick(event){
  e.setAttribute('animation-mixer', 'clip:
     animate');
}
```

Host entities with `can-read` and `can-write` attributes are converted into `TamedDOM` objects. Thus, a third-party ad script can obtain the references of these objects via `querySelector()` or `querySelectorAll()`.

6.5 3D Ad Confinement

AdCube uses a bounding helper box, called a BBox [11], for publishers to confine the locations of ad entities, which addresses the first security requirement (§6.1). When loading or creating an ad entity within a specified BBox, AdCube resizes the entity to fit within the BBox. Note that VR axis scales often differ between the VR worlds of the entity and the underlying publisher's website. Therefore, we decided to resize ad entities that do not fit, instead of rejecting them.

This security enforcement requires AdCube to compute whether a specified BBox is able to contain a target entity. It is straightforward to compute whether primitive entities, such as boxes or spheres, fit within the hexahedron. AdCube simply does this by invoking the Box3 API in Three.js, which internally calculates an axis-aligned bounding box in 3D space.

However, checking whether a 3D model fits within a BBox entails a technical challenge; when the model is designed to animate or move around in a scene, it is necessary to compute the maximum size of the model at the time of loading. That is, AdCube should estimate the maximum size of this model and ensure that its estimated size fits within the specified BBox.

We tackle this challenge by playing a target model one-time before attaching this model to a scene. The idea is to sample frames while rendering the target 3D model and compute the maximum boundary of the shapes in these frames.

To this end, we project the model into 2D space and sample one frame out of 17 frames during the animation loop, which runs once. We then find the maximum size of the shape by scanning the pixels in the captured frames. Because only information for two axes is obtained in the 2D projection, we then rotate the camera angle (e.g., from front to side) and repeat the operation to retrieve information for three axes. AdCube projects a model onto the x-, y-, and z-axes and overlays the frames rendered during the animation. AdCube obtains the min/max positions of the pixels that are not the same color as a background and calculates the maximum BBox.

Considering that we only sample one out of 17 frames, our method may not compute an accurate size of a given model. To address this, one can increase the sampling rate, thus capturing more frames in exchange for increasing the latency in model loading.

It is possible to append multiple ad entities to a single `adcube` space. For this, we use a Three.js Group object [21]. The Group object allows the management of multiple entities, including their children, as a single entity. We update the Group object when a new ad is added and adjust the scale of the entire group to prevent it from escaping the `adcube`.

7 Evaluation

This section describes a showcase of WebVR ads enabled by AdCube (§7.1). We then evaluate the security of AdCube (§7.2) and the performance of AdCube (§7.3).

7.1 Ad Showcase

We conducted a preliminary study investigating on-going VR ad campaigns offered by OmniVirt [52], Adverty [6], and Admix [4]. They support three kinds of VR ad campaigns: i) billboard ads, ii) entity ads, and iii) image ads. A billboard ad campaign renders its video or image on a billboard in a VR scene. An entity ad campaign places a 3D ad object in a VR scene. An image ad campaign places an image of which the z-depth is zero in a VR scene.

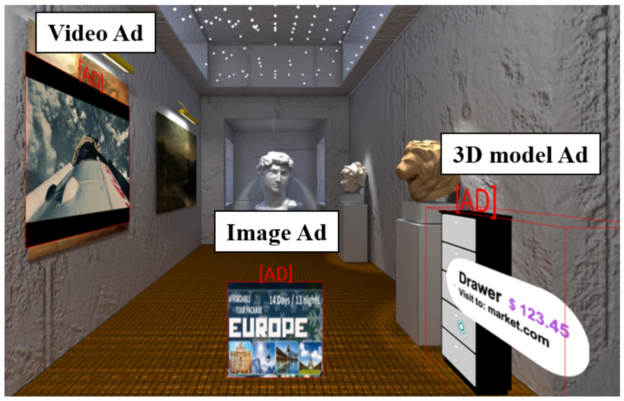


Figure 6: A showcase of WebVR ads with AdCube.

To demonstrate that AdCube supports each VR ad type, we implemented an ad showcase on WebVR, as shown in Figure 6. In the figure, the underlying VR environment is an art museum where users can experience VR art content [60]. All the entities within the red-bordered hexahedrons are from sandboxed ad-serving scripts. The billboard on the left wall renders a video ad campaign, and the one on the floor renders an ad impression promoting hotels in Europe. This image rendering display is attached to the current camera, thus varying in accordance with the user's current line of sight. On the right side, the third-party script draws a 3D drawer model.

7.2 Security

To evaluate the security provided by AdCube, we checked whether it is able to block all four of the presented attacks (§4). We assume that the adversary is an ad service provider that delivers the ad entities in Figure 6. For this experiment, we implemented a host website with the default security policy that specifies no `can-read` and `can-write` properties.

The default policy provides no reference point to a confined third-party script so that the script complied via Caja becomes unable to obtain a current camera position, insert new fake cursors, or modify any DOMs in the host page, thus rendering all the presented attacks ineffective. Note that this default policy blocks third-party scripts from reading or writing any first-party elements, including cookies, thus mitigating traditional web threats [39, 82].

A publisher may grant `can-read` and `can-write` access to their host camera and attach an `<a-adcube>` tag to the current camera, which makes this adcube area to move along with the camera perspective. However, to prevent the BST attack, AdCube prohibits the z position value from being a positive value when the adcube tag has the camera as its parent. Furthermore, all fake gaze and controller cursors that third-party scripts generate will be visually distinguishable from their host scene because these cursors will be confined within a helper box with the "AD" label.

Note that it is feasible to abuse an auxiliary display when the publisher allows a third-party script to revise the host page. However, this can be easily blocked by carefully assigning `can-write` properties to host DOMs. The extension of such a policy can also block a third-party script from accessing private user information and credentials belonging to the host page, which is an original security goal of Caja.

We also emphasize that an `<a-adcube>` tag visually confines VR entities within this adcube area. When AdCube adds or loads VR entities in an adcube area, it ensures that these loaded entities do not escape from this area.

7.3 Performance

We evaluated the performance overhead of AdCube and compared it with two other methods: Baseline and Mirroring. The baseline method is to run a third-party script without any underlying security defense, thus running it with the same origin as its host. For the other method, we chose an origin-based

Table 4: Comparison of the average page loading times for nine WebVR sites and the average FPS for 12 events on the showcase with Baseline, Mirroring, and AdCube.

Performance Evaluation	Baseline	Mirroring	AdCube
Average Loading Time (s)	0.55	0.95	0.78
FPS (drop rate)	56.70 (-)	53.12 (6.32%)	55.79 (1.60%)

execution separation method that runs the third-party script in a separate origin different from its host origin. For this, we referred to AdJail [38], which provides a secure ad service using the origin separation method in a standard web environment. This approach leverages the SOP enforcement of the browser and uses a postMessage API [50] for communications between different origins. The unique feature of this approach is to mirror any entity updates on the host page in a separate third-party iframe to address the scenario in which the first- and third-party contents interact with each other. AdJail only mirrors the static content types of ads that are not necessary to be rendered on the mirror page. However, in a WebVR environment, the mirror page must have a VR scene in order to sync ad behaviors between the two origins; therefore, rendering the scene in both pages is inevitable. We implemented this AdJail approach (denoted by Mirroring) for the comparative study.

Experiment setup. For each defense approach, we measured the page loading time and FPS on a machine with Intel i7 CPU, GeForce GTX 1060, and 32GB of main memory. All experiments were conducted using Firefox 78.0.2.

Loading latency. To understand the overhead of deploying AdCube, we measured the loading time of a WebVR webpage. When a page is requested, all three approaches (i.e., Baseline, Mirroring, and AdCube), request a VR library (e.g., A-Frame) and a third-party ad script and then render the host scene. We assumed that a target website is furnished with a Caja library because this library is a part of AdCube, and AdCube is a defense system for website owners. AdCube establishes an execution environment for Caja. It then parses adcube tags on the host page and renders third-party ad entities into adcube areas after resizing these entities. On the other hand, the Mirroring approach generates a guest page within an iframe and loads the required resources onto both the guest and host pages. It then renders ad entities on the guest page and mirrors these entities on the host page.

For the experiment, we used WebVR showcases on the A-Frame official site [1]. Of 17 showcases, we collected a total of nine open source apps that use the later versions of A-Frame 0.6. These WebVR sites comprise diverse demo purposes (Hello WebVR, Lights, Anime UI), games (A-Blast, Super Says), and utilities (360 Image, 360 Image Gallery, A-Painter, A Saturday Night).

For the guest code to be sandboxed, we created an ad-serving JS script that loads a static 3D chair model and applied three approaches to it. We also specified a security policy for each website that specifies three host entities with the can-write property, with which the ad-serving guest code is able to interact. We measured the page loading time ten times using Firefox with cache enabled and reported the average.

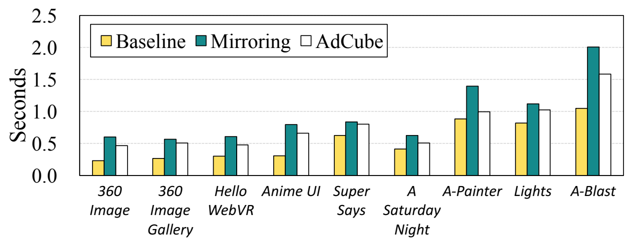


Figure 7: A comparison of page loading times between Baseline, Mirroring, and AdCube for nine WebVR sites.

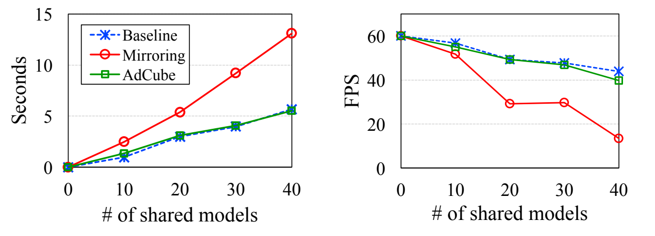
Figure 7 represents the page loading time of the nine WebVR demo websites using three approaches: Baseline, Mirroring, and AdCube. The Saturday Night and A-Blast websites exhibited the smallest and largest overheads for AdCube, reporting an additional 95 and 537 msec, respectively. On average, the page loading time of the nine demo sites with AdCube took an additional 236 msec, compared to an additional 406 msec with Mirroring. Furthermore, the page load time for each website with AdCube was consistently smaller than with the Mirroring approach. As Table 4 shows, the average loading time of the nine WebVR websites was 0.55 sec (Baseline). When applying the Mirroring and AdCube methods, the average loading times were 0.95 and 0.78 sec, respectively.

To understand this observed loading latency by AdCube, we further measured the rendering time by subdividing steps. The rendering time of AdCube includes the execution time of Caja, which can be divided into three steps: 1) requesting caja.js and connecting with the Caja's server; 2) making the host code accessible to the guest code; and 3) loading the guest code and cajoling the code.

Table 5: Overall rendering latencies (msec.) for Lights and A-Blast where having Caja's minimum and maximum execution overhead, respectively.

Website	Caja Execution Time			Total Rendering Time
	Step 1	Step 2	Step 3	
Lights	255.9	1.1	1.6	1016.6
A-Blast	1125.2	1.3	2	1533

Table 5 shows the overall loading time for websites with Caja's minimum and maximum execution overhead from a total of nine websites. Caja's execution resulted in rendering latencies of 25.44% for Lights and 73.61% for A-Blast. This means that more than 25.44% of the rendering latency for all the nine websites using AdCube is due to the Caja setup. That is, the initialization time of Caja dominated the observed page loading times, whereas the latency for cajoling ad-serving JS



(a) Page loading time variations **(b)** FPS variations

Figure 8: Page loading time of a testing page while varying the number of shared 3D models.

scripts was small, which was less than 2 msec.

For the A-painter, Lights, and A-Blast Baseline websites that exhibited relatively longer page loading times, the Mirroring approaches of their corresponding websites also exhibited greater overheads. This is because the Mirroring approach inevitably entails the redundant loading of resources onto the guest page, which means that the more objects rendered, the greater the overhead. On the other hand, the initialization of Caja dominates the page loading time of AdCube, which happens only for the first visit.

Note that we chose an arbitrary number of three host elements that interact with a sandboxed ad-serving JS code for the experiment. Thus, we further measured variations of page loading time as we increased the number of elements shared between a host page and its ad-serving JS code. For the host page, we implemented an empty VR world and added a given number of shared 3D models, which were randomly chosen from 3D static models at Sketchfab [59]. Figure 8a shows the experimental results, which demonstrate that the page loading time in the Mirroring approach increases significantly with the mirroring of many 3D models. This demonstrates that AdCube is more scalable than the Mirroring approach to cope with an arbitrary number of shared entities.

FPS. To assess the overall performance during the exploration of a WebVR world, we measured the FPS change for each approach while the WebVR website was running. We experimented with the museum site presented in Section 7.1, including the various types of advertising campaigns. To show FPS variations, we added entities of museum sculptures to the virtual scene to intentionally lower the FPS rate.

Existing VR advertising services [6, 52] provide interaction behaviors for each type of ad. For example, accordance with user interactions, video ads can be loaded, image ads can be resized, or the animation states of 3D models can be changed. Based on this, to measure the impact of user and publisher interactions on FPS, we also defined 12 different user events and implemented their corresponding event handlers. We then forced them to trigger every 10 seconds using `setTimeout()`. We further describe each event with its handler implementation in Appendix B.1.

Table 4 shows the average FPS of our showcase website when the 12 events were triggered for Baseline, Mirroring, and AdCube, respectively. Note that Firefox caps its FPS at 60. The average FPS for Mirroring is 53.12, which is an additional 6.32% decrease from the Baseline approach. On the other hand, AdCube exhibited a 1.60% decrease (55.79 FPS), which shows a negligible FPS drop from the Baseline approach. This means that AdCube provides stable performance even when various events occur.

We further measured the FPS variations while increasing the number of objects shared between a host page and ad-serving JS scripts. We used the same empty VR webpage used for measuring the page loading time variations as increasing the number of shared objects. We also measured the average FPS for 20 seconds after the page completes loading. As shown in Figure 8b, when reaching 40 shared objects, the average FPSs for Baseline, Mirroring, and AdCube decreased to 43.86, 13.45, and 39.75, respectively. AdCube exhibited an FPS drop similar to that of the Baseline approach. The experimental results indicate that AdCube is a practical solution compared to Mirroring in WebVR, in which FPS drops are critical. Note that an abrupt FPS decrease reduces the user's sense of immersion in the VR mode and may cause a poor user experience [13].

8 Discussion

This section discusses other possible defense methods against the proposed WebVR attacks and their limitations.

Visibility reporting. One may implement a visibility reporting approach that attaches observers [49] to VR ad entities to check their visibility to users, thus mitigating BST attacks. This requires revising existing 3D JS libraries or frameworks (e.g., *Three.js* and *A-Frame*) to compute the intersections between ad entities and users' viewports. However, this type of defense does not block the AAD attack because ad entities are actually rendered in the auxiliary display. Also, the adversary may conduct a GCJ or CCJ attack that induces a victim to trigger clicks on ad entities when the victim watches or clicks non-promotional entities. That is, visibility reporting does not address GCJ, CCJ, or AAD attacks because these attacks stem from no access control when third-party scripts read or revise first-party resources.

HTC supports the eye-tracking API [75], which can be used for visibility reporting. However, this API is unavailable to WebVR, and the current specification [78] does not define interfaces for retrieving eye-tracking information. Furthermore, allowing access to user's eye movements would entail a privacy risk by third parties abusing the information, which necessitates sandboxing third-party scripts.

We also believe that WebXR specification changes cannot address GCJ, CCJ, or AAD attacks. Blocking these attacks requires restricting third-party script behaviors; however, the specification is designed to define interfaces for providing VR worlds and peripherals.

Native browser support. One possible defense is to integrate AdCube with a browser engine, thereby sandboxing third-party scripts. We believe that native browser supports for sandboxing general websites require a long-term development plan with significant engineering effort. Implementing browser-level sandboxing requires identifying the source of a given script to execute; this is because the browser should determine whether to sandbox a given script based on its source. However, the dynamic nature of JS makes it difficult to determine the true sources of dynamically generated JS scripts when the generation involves multiple origins.

Furthermore, it is important to maintain the creator's origin of each DOM element because a browser should determine the accessibility of such DOM elements. However, this requires significant changes to today's modern browsers. Chrome developers discussed implementing a similar functionality of tracking the creators of dynamically generated iframe DOMs and concluded that its implementation would introduce numerous corner cases, providing a false sense of security [16].

We propose a practical sandboxing tool that requires no change to browsers. AdCube addresses the aforementioned two challenges by not allowing dynamically generated scripts and leveraging security policies specified by publishers.

AR support. Recently, WebAR services [40, 55] have been introduced, and several vendors [9, 10, 35, 70] have provided JS libraries that enable AR services in websites. A website owner is able to pop up 3D augmented entities in a user's mobile browser when this user's camera looks at a marked predefined for user recognition. AdCube can be integrated with such a WebAR service; it is able to visually confine augmented entities from untrustworthy third parties and to sandbox their execution when they come with JS scripts.

9 Related Work

Online ad fraud. Ad fraud refers to an operation that generates unintended ad traffic involving ad impressions or clicks. Numerous studies have explored methods of ad fraud [8, 25, 28, 41, 63, 69]. Thomas *et al.* [69] identified ad networks that replace existing ad impressions to swindle advertising income from benign publishers. Huang *et al.* [28] suggested new clickjacking attack methods, such as cursor spoofing and white-a-mole, for click fraud. We introduce four new attacks that harness features unique to WebVR.

Third-party ad sandboxing. There have been extensive studies on sandboxing third-party libraries of publisher sites [2, 23, 30, 38, 42, 45, 54, 58]. AdJail [38] provides an isolation technique that inserts an ad script into another hidden shadow page that has a different origin than that of the publisher, and it adds the ad content to the host page via an inter-origin communication mechanism [50]. AdSentry [23] achieved the same goal using a different technique that modified the JS engine in the browser to prevent third-party code from interfering with the host's context. Politz *et al.* [54] proposed a language-based sandboxing technology, called AdSafety. They defined a subset of JavaScript and provided related safeguards through type-based verification. Instead of devising our own sandboxing system, AdCube is built on top of Caja, a mature open source project.

Security and privacy of AR and VR. Despite significant attention to VR, there have been few studies of its security and privacy aspects [3, 26, 74]. Vilk *et al.* [74] addressed new privacy threats posed in immersive environments. They revealed the privacy risks of using raw camera data or user gesture information, which could expose users' private data, such as room information or people around them. To address these threats, Adams *et al.* [3] established standards for the ethical developments of VR content by carrying out extensive user studies. George *et al.* [26] investigated information leakage that could occur when a bystander observes VR users. Lebeck *et al.* [34] manifested security, privacy, and safety concerns in multi-user AR systems. Because most WebVR sites offer a VR world for each user, we presented the attacks in a single-user scenario. In a multi-user environment, conducting stealthy BST, GCJ, and CCJ attacks would be more difficult because the adversary should compute blind spots and hide cursors from every participant. However, when gaze or controller cursors are invisible to other participants, the multi-user environment will not affect the GCJ and CCJ attacks.

10 Conclusion

Assuming a malicious adversary who abuses the lack of built-in browser support of sharing canvas DOMs, we have devised four new attack variants to conduct VR ad fraud. Our user study showed that the devised attacks are effective in conducting stealthy impression and click fraud. To defend against the presented threats, we proposed AdCube, which allows honest publishers to confine the locations of ad entities as well as to sandbox third-party ad scripts. We advocate ad service providers and publishers to alarm the presented risks in WebVR and adopt AdCube.

Acknowledgments

We thank the anonymous reviewers for their concrete feedback. We are also grateful to Juho Kim for kindly consulting the user study design. This work was partly supported by an NSF CAREER award and the Institute of Information & Communications Technology Planning& Evaluation (IITP) grant funded by the Korea government (MSIT), No.2020-0-00209.

References

[1] A-Frame. A WebVR Implementation Platform. `https://aframe.io/docs/0.9.0/introduction/`. last visited: 2020-10-16.

[2] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail: Least-Privilege Integration of Third-Party Components in Web Mashups. In *Proceedings of the Annual Computer Security Applications Conference*, 2011.

[3] Devon Adams, Alseny Bah, Catherine Barwulor, Nureli Musaby, Kadeem Pitkin, and Elissa M. Redmiles. Ethics Emerging: the Story of Privacy and Security Perceptions in Virtual Reality. In *Fourteenth Symposium on Usable Privacy and Security*, 2018.

[4] Admix. An Online Advertising Service. https://admix.in/. last visited: 2020-10-16.

[5] Admix. State Farm Case Study. https://admix.in/case-studies/state-farm/. last visited: 2020-10-16.

[6] Adverty. An Online Advertising Service. https://adverty.com/. last visited: 2020-10-16.

[7] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-side Sandboxing of Third-party JavaScript without Browser Modifications. In *Proceedings of the Annual Computer Security Applications Conference*, 2012.

[8] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking Revisited: A Perceptual View of UI Security. In *USENIX Workshop on Offensive Technologies*, 2014.

[9] argon.js. WebAR JS Library. https://www.argonjs.io. last visited: 2020-10-16.

[10] AR.js. WebAR JS Library. https://github.com/AR-js-org/AR.js. last visited: 2020-10-15.

[11] Gill Barequet and Sariel Har-Peled. Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in Three Dimensions. *J. Algorithms*, 38(1):91–109, 2001.

[12] Tommy Blizard and Nikola Livic. Click-fraud monetizing malware: A survey and case study. In *International Conference on Malicious and Unwanted Software*, 2012.

[13] Jelte E. Bos, Willem Bles, and Eric L. Groen. A theory on visually induced motion sickness. *Displays*, 29(2):47–57, 2008.

[14] Anca Bradley. Avoid These 7 Annoying Ad Placement Techniques on Your Site. https://www.entrepreneur.com/article/240098. last visited: 2020-10-16.

[15] Facebook Ads Help Center. About Descriptions in News Feed Ads. https://www.facebook.com/business/help/1130862553791128. last visited: 2020-10-16.

[16] Chromium. Issue 1615523002: Transitively keep track of an isolated world's children scripts and worlds. https://codereview.chromium.org/1615523002/. last visited: 2020-10-16.

[17] Federal Trade Commission. Advertising and Marketing on the Internet : Rules. https://www.ftc.gov/tips-advice/business-center/guidance/advertising-marketing-internet-rules-road. last visited: 2020-10-16.

[18] Jonathan Crussell, Ryan Stevens, and Hao Chen. MAdFraud: Investigating Ad Fraud in Android Applications. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2014.

[19] Google Developers. Introduction to Caja. https://developers.google.com/caja. last visited: 2020-10-16.

[20] Google Developers. Introduction to Secure EcmaScript. https://github.com/google/caja/wiki/SES. last visited: 2020-10-16.

[21] Three.js Docs. Objects - Group. https://threejs.org/docs/#api/en/objects/Group. last visited: 2020-10-16.

[22] Xinshu Dong, Zhaofeng Chen, Hossein Siadati, Shruti Tople, Prateek Saxena, and Zhenkai Liang. Protecting Sensitive Web Content from Client-Side Vulnerabilities with CRYPTONS. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2013.

[23] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. AdSentry: Comprehensive and Flexible Confinement of JavaScript-based Advertisements. In *Proceedings of the Annual Computer Security Applications Conference*, 2011.

[24] Jorge Fuentes. HalloVReen: A WebVR Experiment for Kids. https://www.jorgefuentes.net/projects/halloVReen/. last visited: 2020-10-16.

[25] Mona Gandhi, Markus Jakobsson, and Jacob Ratkiewicz. Badvertisements: Stealthy Click-Fraud with Unwitting Accessories. *J. Digital Forensic Practice*, 1(2):131–142, 2006.

[26] Ceenu George, Mohamed Khamis, Emanuel von Zezschwitz, Marinus Burger, Henri Schmidt, Florian Alt, and Heinrich Hussmann. Seamless and Secure VR: Adapting and Evaluating Established Authentication Systems for Virtual Reality. In *Usable Security Workshop on NDSS*, 2017.

[27] Google Ads Help. How do I Stop Ads from Covering Text? https://support.google.com/google-ads/thread/1452412?hl=en. last visited: 2020-10-16.

[28] Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, 2012.

[29] IAB. Digital Advertising. https://www.iab.com/wp-content/uploads/2016/04/HTML5forDigitalAdvertising2.0.pdf. last visited: 2020-10-16.

[30] Lon Ingram and Michael Walfish. Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[31] Dean Jackson and Jeff Gilbert. WebGL Specification. https://www.khronos.org/registry/webgl/specs/latest/1.0/. last visited: 2020-10-16.

[32] John Koetsier. Mobile Ad Fraud: What 24 Billion Clicks on 700 Ad Networks Reveal. https://blog.branch.io/mobile-ad-fraud-what-24-billion-clicks-on-700-ad-networks-reveal/. last visited: 2020-10-16.

[33] Kiron Lebeck, Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. Securing augmented reality output. In *2017 IEEE symposium on security and privacy (SP)*, pages 320–337. IEEE, 2017.

[34] Kiron Lebeck, Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. Towards security and privacy for multi-user augmented reality: Foundations with end users. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2018.

[35] Letsee. WebAR SDK. https://www.letsee.io/ko/. last visited: 2020-10-15.

[36] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, 2015.

[37] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[38] 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 USENIX Security Symposium*, 2010.

[39] Delfina Malandrino and Vittorio Scarano. Privacy Leakage on the Web: Diffusion and Countermeasures. *Computer Networks*, 57(14):2833–2855, 2013.

[40] First Man. WebAR Serve. https://moon.firstman.com. last visited: 2020-10-16.

[41] Miriam Marciel, Rubén Cuevas, Albert Banchs, Roberto Gonzalez, Stefano Traverso, Mohamed Ahmed, and Arturo Azcorra. Understanding the Detection of View Fraud in Video Content Portals. In *Proceedings of the International Conference on World Wide Web*, 2016.

[42] Leo A. Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[43] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. Augmented reality: A class of displays on the reality-virtuality continuum. In *Telemanipulator and telepresence technologies*, 1995.

[44] Moloco. The "Axis of Evi" in Mobile Ad Fraud. https://medium.com/@moloco/bad-ad-networks-the-axis-of-evil-in-mobile-ad-fraud-89ca577de2b6. last visited: 2020-10-16.

[45] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2019.

[46] Mozilla Developer Network. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. last visited: 2020-10-16.

[47] Mozilla Developer Network. HTML Canvas Element. https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement. last visited: 2020-10-16.

[48] Mozilla Developer Network. iframe: Inline Frame Element. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe. last visited: 2020-10-16.

[49] Mozilla Developer Network. Intersection Observer. https://developer.mozilla.org/ko/docs/Web/API/Intersection_Observer_API. last visited: 2020-10-12.

[50] Mozilla Developer Network. postMessage() API. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage. last visited: 2020-10-16.

[51] Mozilla Developer Network. Same-Origin Policy (SOP). https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. last visited: 2020-10-16.

[52] Omnivirt. An Online Advertising Service. https://www.omnivirt.com/. last visited: 2020-10-16.

[53] PlayCanvas. A WebVR Implementation Platform. https://playcanvas.com/industries/vr. last visited: 2020-10-16.

[54] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *Proceedings of the USENIX Security Symposium*, 2011.

[55] Purina. WebAR Serve. https://one28daychallengear.purina.com. last visited: 2020-10-16.

[56] Vhite Rabbit. WACKARMADIDDLE: A WebVR Wack-A-Mole game. https://constructarca.de/game/wackarmadiddle/. last visited: 2020-10-16.

[57] Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-scale Legacy Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[58] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[59] Sketchfab. A WebVR Implementation Platform. https://www.sketchfab.com. last visited: 2020-10-16.

[60] Cecropia Solutions. The Hall: A WebVR demo that displays art. https://cecropia.github.io/thehallaframe/. last visited: 2020-10-16.

[61] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What Mobile Ads Know About Mobile Users. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[62] Aditya K. Sood and Sherali Zeadally. Drive-By Download Attacks: A Comparative Study. *IT Professional*, 18(5):18–25, 2016.

[63] Kevin Springborn and Paul Barford. Impression Fraud in On-line Advertising via Pay-Per-View Networks. In *Proceedings of the USENIX Security Symposium*, 2013.

[64] Statista. Active virtual reality users forecast worldwide 2014-2018. https://www.statista.com/statistics/426469/active-virtual-reality-users-worldwide/. last visited: 2020-10-16.

[65] Statista. Global consumer spending: AR/VR content and apps 2021. https://www.statista.com/statistics/828467/world-ar-vr-consumer-spending-content-apps/. last visited: 2020-10-16.

[66] Statista. VR and AR ownership in the U.S. by age 2017. https://www.statista.com/statistics/740760/vr-ar-ownership-usa-age/. last visited: 2020-10-15.

[67] Brett Stone-Gross, Ryan Stevens, Apostolis Zarras, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Understanding Fraudulent Activities in Online Ad Exchanges. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, 2011.

[68] Techjury. 43 Virtual Reality Statistics That Will Rock The Market In 2020. https://techjury.net/stats-about/virtual-reality/#gref. last visited: 2020-10-16.

[69] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[70] three.ar.js. WebAR JS Library. https://github.com/google-ar/three.ar.js. last visited: 2020-10-15.

[71] Three.js. A JavaScript 3D Library. https://threejs.org/. last visited: 2020-10-16.

[72] Tung Tran, Riccardo Pelizzi, and R. Sekar. JaTE: Transparent and Efficient JavaScript Confinement. In *Proceedings of the Annual Computer Security Applications Conference*, 2015.

[73] VentureBeat. 1 billion AR/VR ad impressions. https://venturebeat.com/2018/12/05/1-billion-ar-vr-ad-impressions-what-weve-learned/. last visited: 2020-10-14.

[74] John Vilk, David Molnar, Benjamin Livshits, Eyal Ofek, Christopher J. Rossbach, Alexander Moshchuk, Helen J. Wang, and Ran Gal. SurroundWeb: Mitigating Privacy Concerns in a 3D Web Browser. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[75] Vive. VIVE Eye Tracking SDK. https://developer.vive.com/resources/vive-sense/sdk/vive-eye-tracking-sdk-sranipal/. last visited: 2020-10-14.

[76] Mozilla VR. A-Blast: A WebVR wave shooter game. https://aframe.io/a-blast/. last visited: 2020-10-16.

[77] W3C. WebVR 1.1. https://immersive-web.github.io/webvr/spec/1.1/. last visited: 2020-10-16.

[78] W3C. WebXR Device API. https://www.w3.org/TR/webxr/. last visited: 2020-10-16.

[79] Business Wire. Global Virtual Reality Content Creation Market Expected to Grow with a CAGR of 77.10% Over the Forecast Period, 2019-2026. https://www.businesswire.com/news/home/20200224005672/en/Global-Virtual-Reality-Content-Creation-Market-Expected. last visited: 2020-10-16.

[80] Wonderleap. An Online Advertising Service. https://wonderleap.co. last visited: 2020-10-14.

[81] Mingxue Zhang, Wei Meng, Sangho Lee, Byoungyoung Lee, and Xinyu Xing. All Your Clicks Belong to Me: Investigating Click Interception on the Web. In *Proceedings of the USENIX Security Symposium*, 2019.

[82] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Hai-Xin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies Lack Integrity: Real-World Implications. In *Proceedings of the USENIX Security Symposium*, 2015.

A Further Details on the User Studies

A.1 Questionnaires for the User Study Survey

For the GCJ and CCJ attacks, we asked the participants of the user study the following questions:

1. *Did you find any promotional products or brands in your VR experience?*
2. *If yes, mark each of the findings in the given examples. (We gave examples of ad objects to choose from.)*
3. *Is there a webpage where you were exposed to more ads between two VR webpages that you explored? If so, why?*

Using the examples provided in Question 2, we could verify whether the ad object that the user claimed to have found was a genuine ad object.

For the BST and AAD attacks, we asked the participants the following questions:

1. *Did you hear any sounds of commercial clips? If yes, which sounds?*
2. *Did you find any ad videos in the background of your VR scene or on your monitor screen? If yes, which videos?*
3. *Which of the two VR websites exposes ads? And which ad is exposed?*

In Question 3, "two VR websites" refers to the normal and attack websites for each attack scenario.

B Additional Evaluation

B.1 Events Used for the FPS evaluation and FPS over Times and across Events

The following list entails nine user events and three publisher events that we used to measure FPS drops in the ad showcase in Figure 6 (§6.5):

– e1: Load and play a video ad
– e2: Attach an image ad to a camera entity
– e3: Resize the image ad
– e4: Load a 3D model ad
– e5: Change an animation status of the 3D model ad
– e6: Replace the video ad with another one
– e7: Replace the image ad with another one
– e8: Replace the 3D model ad with another one
– e9: Modify the host entity with permission
– e10: Hide the video ad (publisher event)
– e11: Change the location of the image ad (publisher event)
– e12: Resize the 3D model ad (publisher event)

Figure 9 shows measured FPS variations over time and across events. Note that Firefox caps its FPS at 60. A targeted museum site loads many entities at the initial time, resulting in a significant FPS reduction in the early stages of all three approaches, including Baseline.

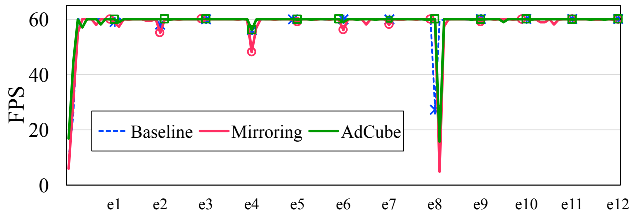


Figure 9: FPS drops in three approaches in response to interaction events.

AdCube exhibited a reliable performance that aligned with Baseline's along all timelines. In contrast, Mirroring struggled with unstable performance when the events occurred. Especially, in the eighth event, replacing a 3D model ad, Mirroring resulted in a decrease of FPS that was about 1.5 times that of Baseline.