IDENTIFICATION OF WEB SECURITY THREATS TO ONLINE BUSINESS MODELS

by

I Luk Kim

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

May 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Xiangyu Zhang, Chair

Department of Computer Science

Dr. Lin Tan

Department of Computer Science

Dr. Changhee Jung

Department of Computer Science

Dr. Pedro Fonseca

Department of Computer Science

Approved by:

Dr. Kihong Park

To my beloved family

ACKNOWLEDGMENTS

I would like to express my gratitude to all of the people who have supported me throughout the journey of my Ph.D. dissertation.

First and foremost, I would like to thank my supervisor Professor Xiangyu Zhang. He has provided me with guidance and support at every step of this journey. From the very beginning, he has been an invaluable source of knowledge and expertise in my research area. He has gone above and beyond to offer me guidance and support, even during challenging times, and has always provided me with constructive feedback that helped me improve my research work. His enthusiasm and dedication for research are truly inspiring. He worked late into the night with me for paper submissions, refining my work and offering valuable advice to enhance my professional writing skills. I am truly grateful for his contribution, and I feel fortunate to have him as my advisor.

I also would like to express my deepest gratitude to my dissertation committee, Professor Lin Tan, Professor Changhee Jung, and Professor Pedro Fonseca, for their guidance, support, and expertise throughout the entire preliminary and nal exam process. Their insightful comments and critical evaluation of my works have played an important role in shaping the final outcome of this research.

I am also grateful to my collaborators and friends who have helped me along the way. Professor Yonghwi Kwon provided me with indispensable assistance from the beginning of my research phase to the completion of my academic papers, ensuring that I had all the necessary information and that my work was of the highest quality. His extensive knowledge, technical expertise, and collaborative approach have been essential to the success of our research. Professor Weihang Wang is one of the kindest person I have ever known, and her positive attitude has been a source of inspiration. She consistently provided me with useful constructive feedback and worked hard to ensure our papers were robust and reliable. Dr. Yunhui Zheng was always generous in giving his time and expertise for my papers, even when I made last-minute requests. His willingness to help and comprehensive knowledge of our field were truly noteworthy. I would also like to thank Dr. Kyungtae Kim, Dr. Dohyeong Kim, Professor Wei You, and Professor Yousra Aafer. They have been incredibly

supportive and provided me with important input and guidance throughout my research. Their contributions, helpful comments, and genuine concern have truly made an impact on the quality and success of my research.

I have been fortunate to work in the scientific solutions group in Research Computing at Purdue University. I appreciate their financial support throughout my academic journey. I am especially grateful to my supervisors Lan Zhao and Dr. Carol Song for giving me an opportunity to work, and their willingness to guide me with patience since I started this work. Working under their supervision has been a meaningful experience, and I will always appreciate their encouragement to my Ph.D. process.

I would like to dedicate a special note of appreciation to my family and friends. My parents, Youngsun Kim and Gyeongae Na, have been my greatest source of strength and motivation, and I am forever indebted to them for instilling in me the values of hard work, perseverance, and dedication. Their sacrifices and commitment to my education have been immeasurable, and I feel blessed to have them as my parents. I also thank to my parents-in-law, Manyong Huh and Youngok Cho, for their genuine belief and love for me, which has been instrumental in reaching this milestone. Their kindness and support have given me the motivation to overcome the challenges and obstacles. I am particularly grateful to my mother-in-law. It is my honor to have her in my life, and her constant love will always be cherished. I would like to express my deepest gratitude to my love, Dr. Youna Huh, for her love, support, and understanding during my long decade of Ph.D. journey. She has stood by me through the long hours of studying and research, providing me with the emotional support and motivation I needed to complete my program. I could not have achieved this without her support. Her sacrifices, dedication, and commitment to my success have been priceless, and I owe her a debt of gratitude that I can never repay. I feel blessed to have her as my other half in life. Lastly, I would like to thank to my sister, Boram Kim, my sister-in-law, Sena Huh, my friends, Professor Hongjun Choi, Professor Hogun Park, Jaewoo Shin, and Dr. Sunghyun Myung, and all my Xaris and Siloam members. I am extremely thankful for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Online business models have become increasingly popular in recent years, providing new opportunities for entrepreneurs and established companies alike. However, along with these opportunities come new risks, particularly in the realm of web security. While traditional threats typically affect the backend systems that provide web services, attackers nowadays can also target the actual business model itself to make financial damage. The threats are becoming more difficult to discover because of the wide-scaled and complex web ecosystem that involves multiple parties.

In this dissertation, we present proposals to identify web security threats to online business models. Specifically, we first introduce a novel ad budget draining attack, AdBudgetKiller, in order to demonstrate a possible attack scenario with real-world cases and to come up with prevention methods. AdBudgetKiller automatically discloses a targeting strategy of an advertiser, then fabricate browsing profiles to dispatch advertisements from the targeted advertiser.

We also present a testing-based approach to automatically identify client-side business flow tampering vulnerabilities. In particular, our method systematically analyzes websites to gather potential tampering locations by using dynamic execution data collection. We then test the websites with tampering proposals to identify any business flow tampering vulnerabilities. Further, we present an enhanced detection method for digital content services that detects business flow tampering vulnerabilities. We perform differential analysis on collected execution traces to determine how the business flow begins to differ. Then we test if the divergence points can be tampered with.

1. INTRODUCTION

The Internet is becoming a dominant platform for various types of business models. E-commerce business is growing rapidly that the global online shopping market is forecast to reach $7 trillion by 2025 [1]. Advertising is the primary source of revenue for the overwhelming majority of online companies, and Internet advertising revenues in the United States totaled $189 billion in 2021 [2]. Revenue in the Video Streaming (SVoD) segment in the United States is projected to reach $39.2 billion in 2023. [3]. The increasing popularity of online business has created an attractive attack surface. Unlike the traditional threats that usually affect backend systems providing web services, attackers can target actual business model itself by exploiting vulnerabilities or manipulating business flows, and these attacks can cause financial damage directly. For example, most newspaper websites use paywall methods that provide a few numbers of free articles for new users, then restrict access by showing subscription messages. If an attacker can bypass the paywall method to see an unlimited number of articles without the subscription, this would cause financial loss.

One of the reasons that online business suffers from this type of security threats is the inevitable participation of multiple parties. Client-side web applications running on web browsers play an important role such as coordinating internal conditions, collecting user-specific footprints, or easing server-side work burdens. Sensitive business logic implemented in the client-side can be easily targeted, and attackers might be able to manipulate the workflow of the logic operations in a controlled manner and achieve various damages. Furthermore, the client-side web applications nowadays are often integrated with content and services from multiple sources. For example, the ad-delivery process, which is an essential business operation of many web-based systems, is usually decoupled from the server-side (i.e., content publisher) and is rather performed on the client-side with the help of an external entity (i.e., advertiser). Even if the server-side web service and system are well-protected, the service provider cannot ensure the security levels of all participated third party entities. Due to the complex integration of the client, server and other parties, online business is not immune to web security threats.

1.1 Dissertation Statement

In order to build a secure web ecosystem for online business, this dissertation is focused on developing program analysis and testing techniques to identify web security threats to online business models.

In particular, we identify a potential ad budget draining attack, called AdBudgetKiller, targeting at specific advertisers by repeatedly pulling their ads to damage the budget. We develop a testing-based approach to automatically identify client-side business flow tampering vulnerabilities. We further propose enhanced approach that discovers the business flow tampering vulnerabilities for digital content service.

1.2 Contributions

The contributions of this dissertation are as follows:

- We propose a novel ad budget draining attack system, *AdBudgetKiller*, against specific advertisers. By leveraging retargeting ads ubiquitously displayed in general websites, the attack is able to repeatedly pull out the ads belonging to the targeted advertisers and effectively drain their ad campaign budgets. We develop a novel technique called *ADHoneyClient* to automatically train users' browsing profiles and discover the targeting strategies used by advertisers using a black-box testing approach. We evaluate our technique on Alexa Top 500 public advertisers, and successfully reveal the targeting strategies of 254 out of 291 public advertisers considered in the experiments We perform distributed attacks against a controlled advertiser as well as 3 real-world advertisers using 10 distributed machines. Within an hour, the attack effectively fetched $40,958$ ads and drained up to $155.89 from the campaign budget of the targeted advertisers.

- We propose a testing-based approach to automatically identify business flow tampering vulnerabilities. In particular, our method systematically examines websites as follows; first, starting with business operation descriptions, we navigate the website and collect a set of functions that may be relevant to the business logic. Then, We analyze each candidate function and look for potential tampering locations, which may perturb

the intended behavior if modified. After that, We develop techniques to select functions that are more likely to be vulnerable and generate tampering proposals for each selected function. Finally, We revisit the website with the tampering proposals and confirm if the detection results are indeed business flow tampering vulnerabilities. We evaluate our technique on 200 popular real-world websites. With negligible overhead, we have successfully identified 27 unique vulnerabilities on 23 websites, such as New York Times, HBO, and YouTube, where an adversary can interrupt business logic to bypass paywalls, disable adblocker detection, earn reward points illicitly, etc.

- We propose an automated approach that discovers business flow tampering flaws for digital content service. Our technique automatically runs a web service to cover different business flows (e.g., a news website with vs. without a subscription paywall) to collect execution traces. We perform differential analysis on the execution traces to identify divergence points that determine how the business flow begins to differ, and then we test to see if the divergence points can be tampered with. We assess our approach against 352 real-world digital content service providers and discover 315 flaws from 204 websites, including TIME, Fortune, and Forbes. Our evaluation result shows that our technique successfully identifies these flaws with low false-positive and false-negative rates of 0.49% and 1.44%, respectively.

2. ADBUDGETKILLER: ONLINE ADVERTISING BUDGET DRAINING ATTACK

In this chapter, we present a new ad budget draining attack. By repeatedly pulling out the ads belonging to the targeted advertisers using crafted browsing profiles, we are able to reduce the chance of showing their ads to real-human visitors and trash their ad budget. With the advertiser profiles collected by an automated ad crawler *ADHoneyClient*, we infer the advertising strategies set by the targeted advertisers and train browsing profiles that satisfy their strategies. We launched large-scale attacks using distributed machines hosted on public cloud computing services. We evaluate our methods and reverse engineer the targeting strategies of 291 public advertisers selected from Alexa Top 500. We successfully revealed the strategies used by 87% of the public advertisers we considered. We also executed a series of attacks against a controlled advertiser and 3 real-world advertisers within the ethical and legal boundary. The attack result shows that we are able to fetch $40,958$ ads from the targeted advertisers and drain up to $155.89 of the ad budget within an hour.

2.1 Introduction

Online advertising is the primary source of income for many Internet companies. In the US market, Google and Facebook generated $168.44 and $112.68 billion [4] from advertising in 2022 respectively. According to a report by the Internet Advertising Bureau (IAB), the revenues generated from Internet advertising in the United States totaled $189.3 billion in the full year 2021 [2], which represents an increase of 35.4% from the revenues reported in 2020. It is estimated that the U.S. digital advertising will continue its growth and the ad revenue will reach $297.4 billion in 2023 [5].

In its basic form, online advertising entails selling spaces on websites to parties interested in showing ads for a monetary fee. However, the mechanisms and backing up the online advertising ecosystem are quite complex. The ad delivery infrastructure involves four major parties: *publishers*, *advertisers*, *ad network*, and *ad exchange*. Publishers are websites owners who offer space to display ads on their websites. Advertisers pay publishers for ad slots to

place specific ad content with embedded links. Ad networks play the role of match-makers to bring together publishers with advertisers who are willing to pay the most for the publisher's offered space. Ad exchanges are networks of ad networks. An ad exchange works similarly as an ad network, except that the buying and selling entities within an ad exchange are ad networks.

To reach the most receptive audience, advertisers often use sophisticated targeting methods to serve ads to the right viewers. The targeting strategies can either be geographical based such as serving an ad to users in a specific country or demographically focused on age, gender, etc. They can also be behavioral variables (such as a user's browsing activities and the purchase history) or contextually focused by serving ads based on the content of a website. In addition, advertisers may employ different targeting strategies. Some advertisers may value customers who placed an item in cart as more promising potential buyers than customers who simply browsed the item page. So they deliver different ads to these two types of customers. Others may consider them as equally favorable and apply the same strategy.

Retargeting is a technique where advertisers use behavioral targeting strategies to promote ads that follow users after they have expressed a prior interest in an advertiser's website, such as looked at or purchased a particular product. Retargeting is very effective as a retargeting ad is personalized to an individual user's interests, rather than targeting groups of people whose interests may vary.

Given the underlying lucrative benefits, the involved ad parties have strong incentives to conduct fraudulent activities. In fact, advertising fraud becomes a massive problem in ad industry and is ruining this billion-dollar business. Ad fraud is costing the U.S. media industry around $81 billion in 2022 and predicted to increase to $100 billion by 2023 [6].

In this chapter, we propose an innovative ad budget draining attack by precisely fetching ads from the targeted advertisers. Our technique is able to reverse engineer targeting strategies and train browsing profiles that satisfy the conditions set by the advertisers.

In summary, we make the following contributions.

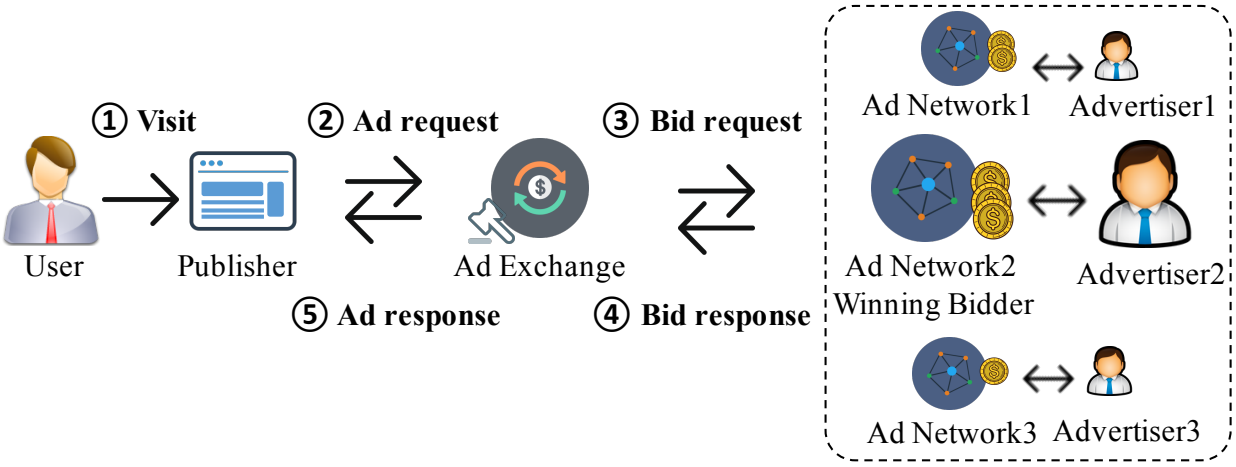


Figure 2.1. Ad ecosystem

- We propose a novel ad budget draining attack targeting at specific advertisers by repeatedly pulling their ads to trash the budget.
- We develop a black-box testing based technique to automatically infer targeting strategies and create satisfying browsing profiles.
- Out of 291 advertisers selected from Alexa Top 500, we successfully revealed the targeting strategies used by 254 advertisers.
- We launched distributed attacks against a controlled advertiser and 3 real-world advertisers. We are able to fetch 40,958 ads and drained up to $155.89 within an hour.

2.2 Online Advertising

In this section, we discuss the entities in the ecosystem and explain how retargeting ad works. We also show existing threats of the ad ecosystem.

2.2.1 Ad Ecosystem

The entities in the ad ecosystem include publishers, advertisers, ad networks, and ad exchanges. Publishers are the websites who earn money by selling ad space on their pages.

Advertisers are the buyers who pay ad networks to deliver their ads. Ad networks are the entities that connect advertisers with websites and help advertisers find right publishers. Ad exchanges are networks of ad networks, which enable ad traffic transactions among ad networks.

Fig. 2.1 explains how an ad is delivered by an ad exchange. When a user visits the publisher website ①, an ad request is sent to the ad exchange ②. The ad exchange conducts a real-time auction, where the exchange sends requests to ad networks ③. Based on the user's characteristic, ad networks respond with their offers ④. The ad exchange picks an offer and delivers the winner's ad to the user ⑤. The whole auction is done in milliseconds.

The ecosystem delivers ads for a fee. There are several pricing models and *cost per thousand impressions* (CPM) is commonly used. Assume the CPM is $7 in Fig. 2.1. The winning advertiser (Advertiser 2) pays 0.7 cents per ad, which will be split among the ad network, the ad exchange, and the publisher.

2.2.2 Retargeting Ad

E-commerce websites want attract potential customers by all means, hoping they will make purchases, become registered users, etc. The percentage of visitors attracted is called the *conversion rate*. In reality, only 2% of visitors take desired actions in their first visit [7]. Retargeting is created to attract the remaining customers by display personalized ads. It tracks website visitors and delivers customized ads when they visit other websites.

In particular, advertisers need to identify a list of high-value visitors. To do so, advertisers include a retargeting pixel, which is a small snippet provided by a retargeting service provider, in their web pages. When a user arrives, the pixel drops an anonymous cookie and enroll this visitor to the list. The anonymous cookie acts as the *browsing profile*, which is a set of IDs and memorizes browsing activities. The retargeting service providers identify unconverted visitors and deliver them personalized ads. To reach more visitors, the retargeting service providers maintain partnership with major ad networks, such as Facebook, and Google Display Network. They participates the real-time ads auctions and bids aggressively.

Retargeting is very effective. E-commerce sites can save money and efforts by selectively targeting visitors who have already expressed interests. According to Kimberly-Clark, a global leader in selling paper products, they accounted for $50 - 60\%$ conversion rates from their retargeting efforts [8]. Similarly, [9] reported that online retailer ShopStyle gained a 200% increase in retargeting conversions. Retargeting also benefits customers because the ads delivered are relevant to their interests.

2.2.3 Threats

While ad exchanges enable efficient and powerful campaigns, their pricing models make the system a highly lucrative target for cyber-criminals. For instance, to artificially inflate the actual impression amount and earn more money, a publisher can fabricate visits to publisher pages such that the advertiser's ad budget is wasted because the ads were not seen by real human visitors. Such fraudulent activity is called *impression fraud*. Although ad networks and exchanges perform real-time monitoring, it is always difficult to prevent from various kinds of fraud activities because of the huge amount of ad traffic.

2.3 Ad Budget Draining Attack

In this section, we elaborate the ad budget draining attack. The victims of our budget draining attack are people or companies that advertise their e-commerce websites using retargeting ad services. The immediate consequence of the attack is the wasted advertisement budget. Moreover, the chance of their ads being displayed can be reduced since it would be difficult to win during the ad auction with the drained ad budget. The potential attackers can be competitor advertisers who may try to drain the others' ad budget and unfairly win the competition. Another possible scenario is denial of service (DOS) attacks performed by people who seek to make ads from the targeted advertiser unavailable for the purpose of a protest. Fig. 2.2 shows the overall attack procedure. The attacker collects data about the targeted advertiser, generates *attack modules* that automatically craft browsing profiles and pull the victim's ads. Note that the attack modules can be independently deployed to launch distributed attack. Throughout these process, the attacker can drain the targeted

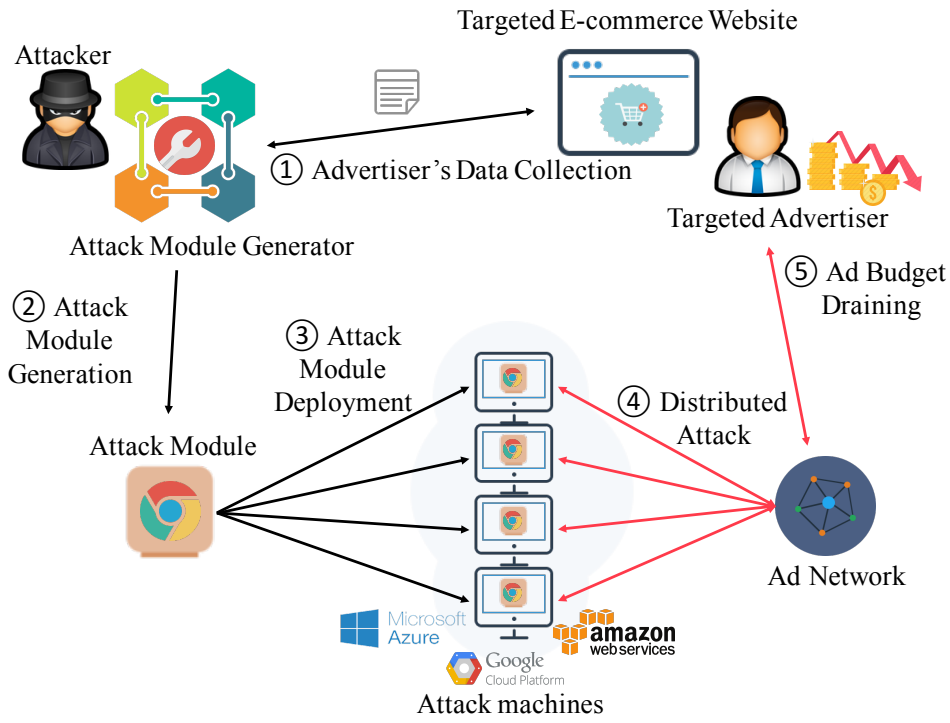


Figure 2.2. Ad budget draining attack procedure

advertiser's ad budget by repeatedly fetching ads. The details of our attack mechanism are explained in the rest of this section.

2.3.1 Overview

As discussed in Sec. 2.2, ad networks track website visitors and deliver targeted ads if they satisfy the advertising strategies. Therefore, identifying the strategies is the first step to attack a particular advertiser. Since ad networks may define arbitrary strategies, effectively reverse-engineering the retargeting logic and craft corresponding browsing profiles are the keys to reproducibly launch large-scale attacks. As shown in Fig. 2.3, *website modeling*, *advertiser profiling*, *attack module generation* and *attack distribution* are the major steps involved in the ad budget draining attack.

① **Website Modeling.** A website model represents structural designs and relationships between web pages. In order to be classified as advertisers' favored customers and eventually

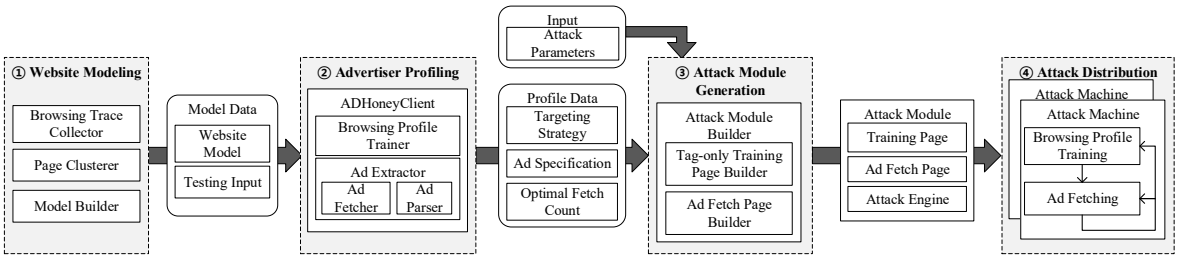


Figure 2.3. Attack Mechanism

see their ads, one effective way is to visit the advertisers' websites and trigger the tracking logic. However, identifying desired navigation sequences that effectively trigger the tracking logic (e.g., products need to be put in the shopping cart) is not trivial due to the huge search space. Therefore, our first step is to create a model for the targeted website to guide the search. In particular, we navigate the targeted website, apply clustering algorithms to the pages, then create a Finite State Machine (FSM) model. Details can be found in Sec. 2.3.2.

② **Advertiser Profiling.** In this step, we focus on inferring targeting strategies. We develop *ADHoneyClient* to automatically discover the strategies based on black-box testing techniques. We also identify the optimal ads fetch count to work around the rate limits set by the ad networks. We explain our algorithms in Sec. 2.3.3.

③ **Attack Module Generation.** The attack modules generated contain the training data and the utilities to create satisfying browsing profiles, where training data is a set of HTML page with ads tracking tags. The module also features an *fetch page* and an *attack engine*. The *fetch page* is a single HTML page with several ads slots that pull the targeted ads. The *attack engine* drives the whole training and ad fetching procedure. As ad networks may equip IP based defense mechanisms, our attack engine can leverage the public proxy lists and randomly change IP addresses to evade IP-based detections. Details can be found in Sec. 2.3.4.

④ **Attack Distribution.** The final step is to deploy the attack modules on multiple machines to launch a distributed attack. In particular, each attack module trains a browsing

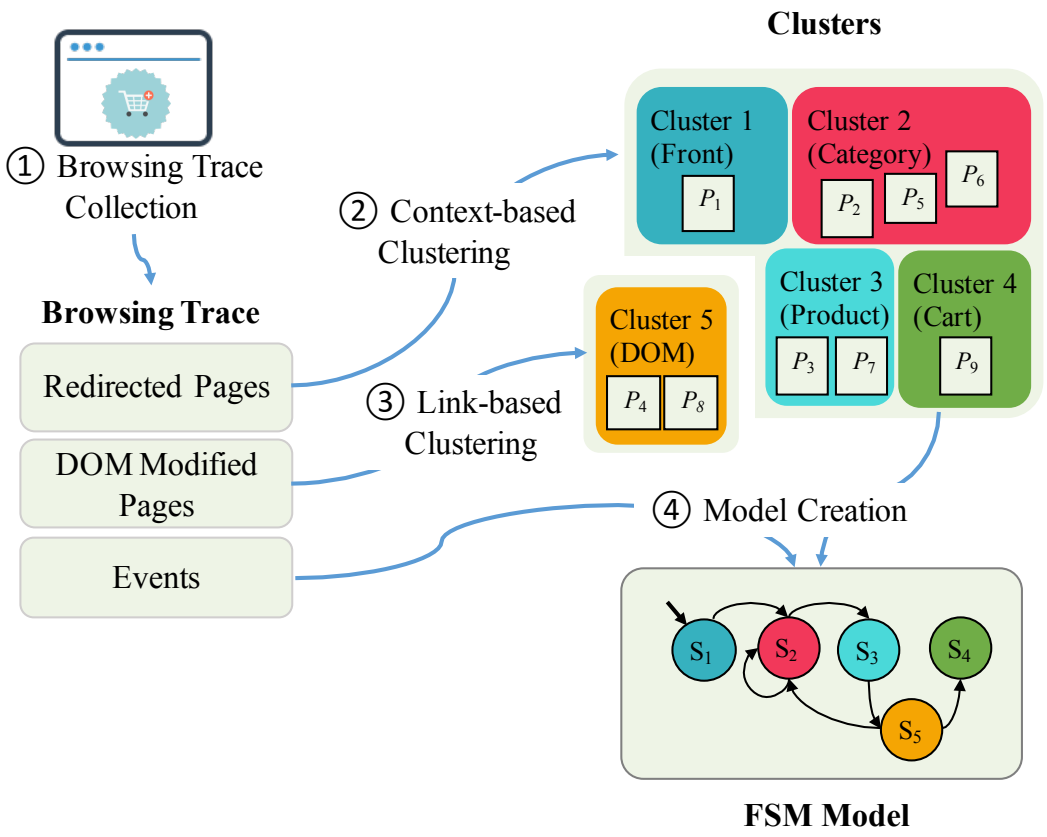


Figure 2.4. Model creation process

profile satisfying the strategy from the training pages and repeatedly fetches ads using the ad fetch page. We explain the details in Sec. 2.3.5.

2.3.2 Website Modeling

A website model describes its structure and transitions among pages. It can be used to guide the targeting strategy discovery. Fig. 2.4 shows the steps for model creation.

Browsing Trace Collector

The browsing trace collected at ① in Fig. 2.4 is used to cluster pages. The collector automatically records browsing activities while an attacker explores the targeted website.

Table 2.1. Example of collected browsing trace

Trace No	Page		Event			
	ID	URL	ID	Action	XPath	Data
1	P_1	shopping.com	E_1	click	//*[id="cat1"]	
2	P_2	./cat1	E_2	click	//*[id="prod1"]	
3	P_3	./item?prod=1	E_3	dom	//*[id="size"]	6
4	P_4	$\#P_3$	E_4	click	//*[id="cat2"]	
5	P_5	./cat2	E_5	click	//*[id="cat3"]	
6	P_6	./cat3	E_6	click	//*[id="prod2"]	
7	P_7	./item?prod=2	E_7	dom	//*[id="size"] //*[id="color"]	7 white
8	P_8	$\#P_7$	E_8	click	//*[id="AddToCart"]	
9	P_9	./cart				

Table 2.1 shows example traces. We record two types of data: pages visited and events triggered. The page data contains the HTML source code and the corresponding URL. If no redirection happens, the page ID is recorded (e.g., P_4 in Table 2.1). The event data describes the browsing action, the DOM object involved, and action attributes.

Note that we do not require a complete website model. Instead, we only need a few inputs. In practice, we observed that usually a small number of actions are sufficient to trigger the tracking logic. For example, if a visitor sees ads after she visited the advertiser's product page, only one action (i.e., visiting the advertiser's product page) is needed. However, if an advertiser targets visitors who added items to the shopping cart and left without buying, the actions of 1) visiting a product page, 2) clicking the add-to-cart button and 3) visiting the cart page are needed.

Page Clustering

With the trace collected, we group similar pages into clusters based on its functionality. For example, P_3 and P_7 in Table 2.1 are grouped together as the *product page*. We apply different clustering methods based on page types:

- The **Redirected Pages** are ones clustered by context (② in Fig. 2.4), where we compare page structures. In particular, we calculate the DOM Tree Edit Distance (TED) [10, 11] and measure the similarity using hierarchical clustering algorithms [12].
- The **DOM Modified Pages** are ones grouped using a link-based clustering method (③ in Fig. 2.4). Specifically, DOM modified pages containing page IDs (links) in a same cluster are grouped together. For example, in Table 2.1, the page P_4 is linked to the page P_3, and P_8 is linked to the page P_7. Since P_3 and P_7 are in the same cluster (*product pages*), P_4 and P_8 are grouped together.

Model Builder

The model builder connects the clusters based on the order observed in traces and assigns event data to the edges. As a result, a Finite State Machine (FSM) is created, where nodes represents *states* and edges with event annotations denote *transitions*. Fig. 2.5 shows a model created from the example traces. By having a model, we can create proper browsing profiles as many as possible, and more importantly, we can avoid creating redundant profiles.

2.3.3 Advertiser Profiling

As discussed in Sec. 2.2, ad networks track website visitors and deliver targeted ads if they satisfy the advertising strategies. As strategies are invisible to us, we have to infer the strategies by profiling the targeted advertisers.

ADHoneyClient: An Automated Ad Crawler

We need a large amount of ads related data to infer the retargeting logic. To automate the data collection process, we develop an ad crawler *ADHoneyClient* to fetch ads with customized browsing profiles and emulate browsing activities. As ads are probably the most complicated and dynamic snippets observed on general websites, *ADHoneyClient* has to handle complicated DOM objects and dynamic JavaScript. *ADHoneyClient* has the following two components:

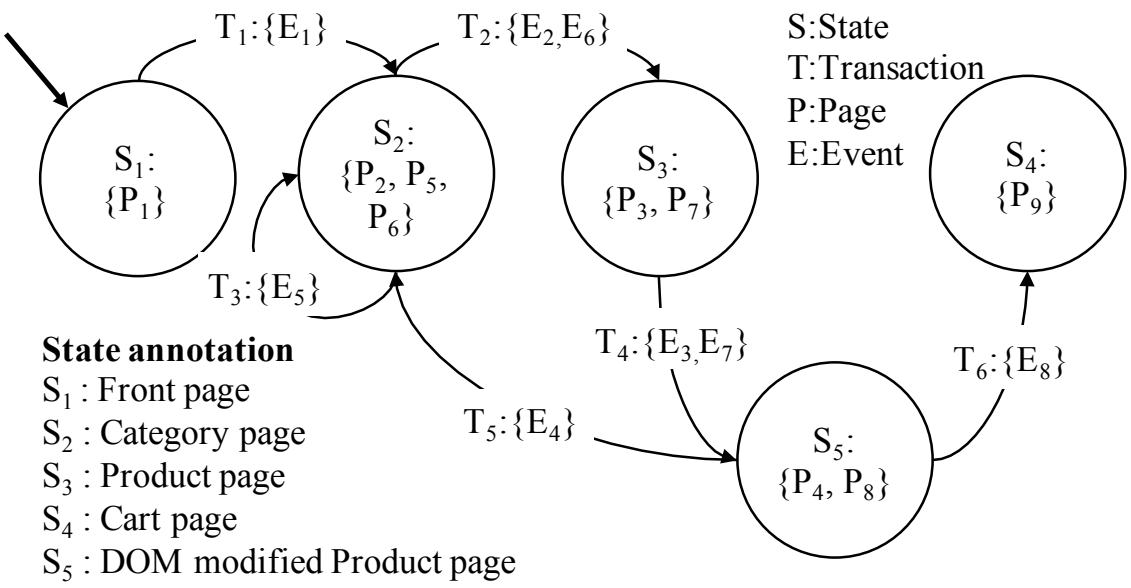


Figure 2.5. An example of website model

1) Browsing Profile Trainer. Browsing profiles are tracking IDs stored in *cookies* that memorize the browsing history. *Browsing profile trainer* crafts browsing profiles by triggering ads tracking logic. Starting from a fresh profile, the *profile trainer* produces customized browsing profiles by simulating browsing activities. It navigates the website guided by the model with example inputs. In the meantime, tracking scripts can update the browsing profile and send browsing histories to the retargeting service providers.

2) Ad Extractor. The *Ad extractor* fetches ads using the crafted browsing profile generated by the trainer. In particular, it infers the targeting strategy, ad specs and the optimal fetch count. Details will be explained in Sec. 2.3.3.

When ads arrive, *Ad parser* determines their sources, specifications (such as types and sizes) and the ad network involved. It also extracts ads related HTML tags. In particular, since ads are usually rendered in the nested `<iframe>` for security purposes, it drills down and looks for specific ids (e.g. "`google_ads_iframe_*`" for DoubleClick). Once found, it collects element attributes as well as HTML tags inside. To identify ad networks, we manually developed 53 signatures. For example, the famous retargeting ad networks *Criteo*

Table 2.2. HTML Tags Used for *Ad Parser*

HTML tag	Attribute	Ad information
`<a>`	id, href	ad_url, ad_network
`<script>`	id, src, innerHTML	ad_url, ad_network
`<noscript>`	innerHTML	ad_url
`<iframe>`	id, src, name	ad_network
`<img>,<embed>,<object>,<video>`	width, height	type, size

[13] can be identified if the `src` of `<iframe>` is `*.criteo.com/delivery/r/afr.php?`. Besides, *Ad parser* harvests all URLs included in the HTML pages pointed by ads related `<iframe>`. It also determines the size and type of the ads from the attributes of observed `<embed>`,`<object>`, `<video>` and `<img>`. We found some tags found in ads related iframes are not useful. We only collect the tags listed in Table 2.2 for better efficiency.

Advertiser Profiling

In this subsection, we explain the *targeting strategy*, *Ad specification*, and *optimal Ad fetch count* produced by *ADHoneyClient*, which will be used as the training data in the next step.

Targeting Strategy. A targeting strategy is a sequence of browsing activities which can be used to identify high-value customers. For example, advertisers can target at visitors who browsed the product pages or left something in carts without buying. A target strategy simulates the browsing activities demonstrated by such favored visitors. Take the website model in Fig. 2.5 as an example. A corresponding browsing activity example can be [*visiting a product page*, *choosing an option*, *adding it to a cart*], which can be described by a path covering states and transitions S_3, T_4, S_5, T_6 and S_4. As our website model is deterministic, the representation can be simplified to S_3, T_4 and T_6. To concertize it, we pick a page/event from each state/transaction and get a targeting strategy $[P_7, E_7, E_8]$.

Algorithm 1 explains how we generate a target strategy from the website model graph produced in Sec. 2.3.2. The output target strategy is a list of browsing activities, where each activity can be either a page or an event obtained from the model.

Algorithm 1 Finding Targeting Strategy

Input:

1. $M = (\mathbb{S}, \mathbb{T})$: website model graph, where vertex $S_{\text{i}} \in \mathbb{S}$ denotes a page cluster $\{P_m, ..., P_n\}$ and edge $T_{\text{j}} \in \mathbb{T}$ is a set of events $\{E_x, ..., E_y\}$.
2. L: Max length of elements in a candidate.
3. N: Max number of candidates to test

Output: targeting strategy $TS = [b_1, .., b_l]$: a list of browsing activities, where $b_{\text{i}} \in \{P_1, ..., P_m\} \cup \{E_1, ..., E_n\}$

```
1:  function FindTargetingStrategy(M, L, N)
2:      for i ← 1 to N do
3:          TS_c ← GenerateCand(M, L)
            /* generate browsing profile guided by TS_c and fetch Ad using ADHoneyClient */
4:          ad ← TrainAndFetchAd(TS_c)
5:          if base url of ad and M is same then
6:              return TS_c
7:      return ∅
8:  function GenerateCand(M, L)
9:      l ← RandomSelect({1, 2, ..., L}) /* length of the candidate */
10:     TS ← [ GetRandomPage(M) ] /* the activities starts with a page */
11:     for i ← 1 to l − 1 do
12:         type ← RandomSelect({"state", "transition"})
13:         if type is "transition" and TS[i − 1] ∉ an accepting state then
14:             if TS[i − 1] is an event E_k then /* continue to the next event */
15:                 TS ← Append(TS, E_{k+1})
16:             else if TS[i − 1] is a page P_k then
17:                 TS ← Append(TS, E_k)
18:         else if type is "state" then
19:             TS ← TSGetRandomPage(M)
20:     if TS has been seen before then /* removes redundant candidates */
21:         TS ← GenerateCand(M, L)
22:     return TS
23: function GetRandomPage(M)
24:     S_r ← randomly select a state from M except DOM modified states
25:     P_r ← randomly select a page in the state S_r
26:     return P_r
```

Function FindTargettingStrategy is the main procedure. It keeps generating different strategy candidates (line 3) until a desired strategy is found (line 6) or the max number of tries is reached (line 2). In particular, FindTargettingStrategy generates uncovered strategy candidates. Given a strategy, *AdHoneyClient* follows the activity sequence in a

strategy, trains the browsing profile and fetches ads (line 4). If the ads are from the targeted advertiser (line 5), a targeting strategy is found.

Function GenerateCand generates a targeting strategy candidate. It starts by randomly picking an initial a page in a state (line 10) and randomly selects pages or events as consecutive activities.

A naive way to select the next activity is to follow the transitions in the FSM website model. However, we observed it cannot effectively create diverse candidates. Instead, we randomly select a page from a state when we want to have a "state" as the next activity. In this way, we can produce more diverse models especially when the model coverage is low. For instance, a strategy generated can be $[P_7, P_1, E_1]$, where we directly go to P_1 after visiting P_7 even though there is no edge between them on the model. Intuitively, this simulates the random jumps among pages during the navigation.

In particular, when we choose to have a "transition" as the next activity (line 13), we append the consecutive event to the activity list (lines 15 and 17). If the type is "state", we select a random page from a random state (lines 24 and 25). Please note that the DOM modified states are excluded as they require DOM modification events and thus not directly accessible (line 24).

Ad Specification. Ad networks define ads parameters such as dimensions and formats (e.g. image, flash, video, etc.) that advertisers have to follow. As we will need to seed ad slots to obtain the desired ads, these specs are important too. For example, if the size of an desired ad is 300×250 but we only supports 160×600, it will not be delivered due to the inconsistency. Therefore, we also collect ads specs. Although some ad providers support responsive banners, where the size can be automatically determined at the time of fetching, ads specs are still useful as they may prevent potential inconsistencies and improve the success rate.

Optimal Fetch Count. In practice, a browsing profile may expire after repeatedly fetching a certain number of ads, as ad networks usually set a rate limit on the ads delivered to a single user. Therefore, we also need to infer the *optimal fetch count*, which is the number of the ads that can be fetched using a single profile. In particular, we monitor the fetch rate

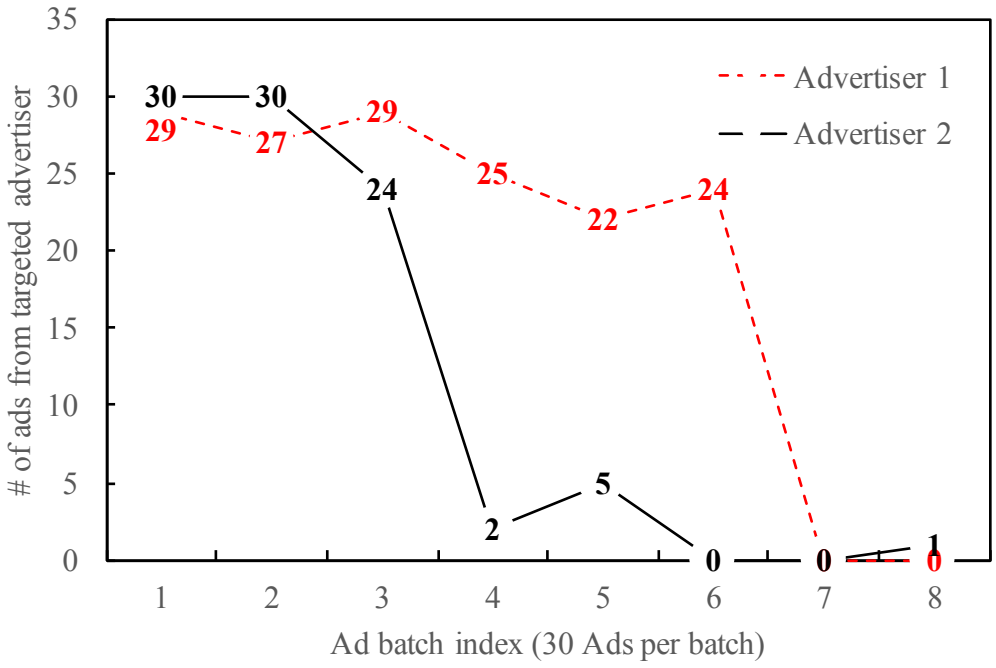


Figure 2.6. Targeted ads fetched with a single browsing profile

using a browsing profile until the rate drops significantly, we use the number of ads fetched before drop as the *optimal fetch count*.

For example, we fetch 30 ads per batch using a single browsing profile from two advertisers. Fig. 2.6 shows the number of ads fetched in each batch. For advertiser 1, the fetch rate drops to 70% at batch 6 and then to 0. Similar patterns are observed for advertiser 2. After the 3rd batch, the rate is decreased to 6%. Therefore, the *optimal fetch counts* for them are 180 and 90 respectively.

In our experience, we use 50% as the threshold to balance the efforts of creating new profiles and ads fetching. In other words, once the targeted ads fetch rate drops below 50%, we stop fetching and set the number of ads fetched so far as the *optimal fetch count*.

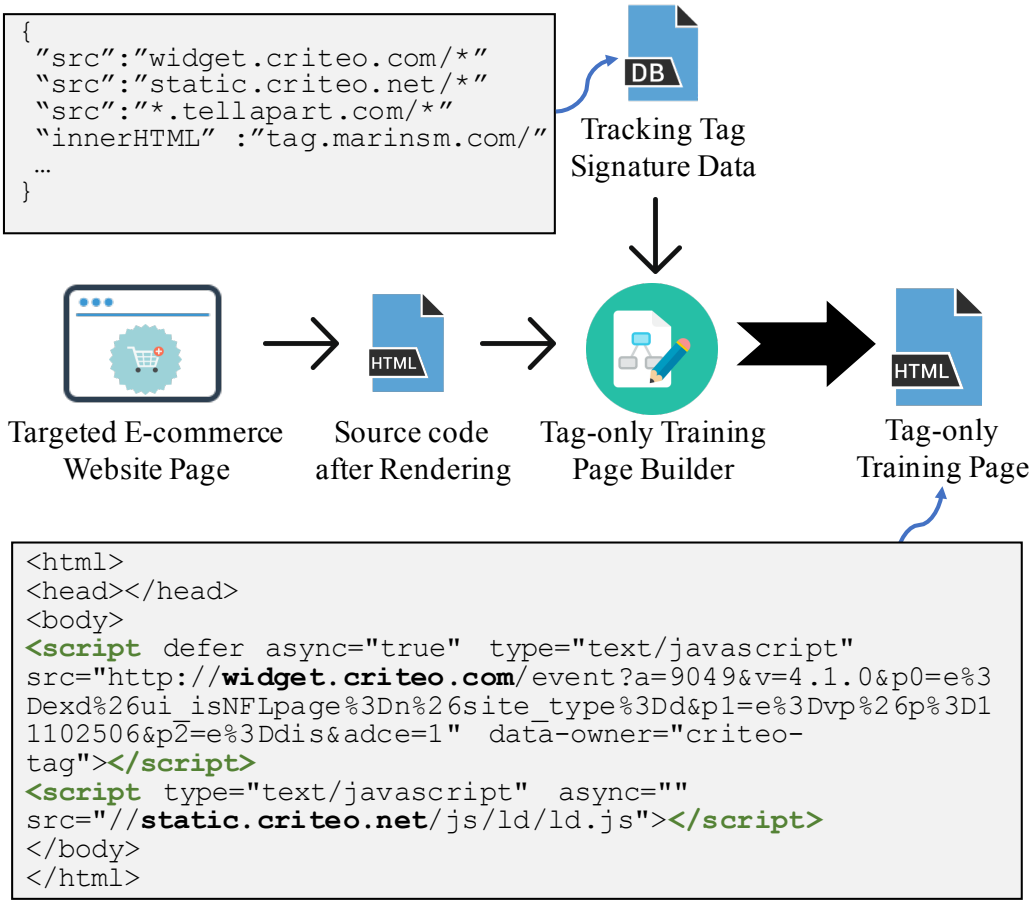


Figure 2.7. Tag-only training page building process

2.3.4 Attack Module Generation

An *attack module* contains three components: *tracking tag-only page*, the *ad fetch page*, and the *attack engine*. The first two are HTML pages for browsing profile training and ads fetching. The attack engine drives the process based on the attack parameters.

Tag-only Training Page Builder

To train browsing profiles, we emulate activities specified in targeting strategies. This is one of the most time consuming parts as we have to repeatedly create new profiles. To correctly set the tracking IDs and browsing histories, we have to trigger the tracking scripts

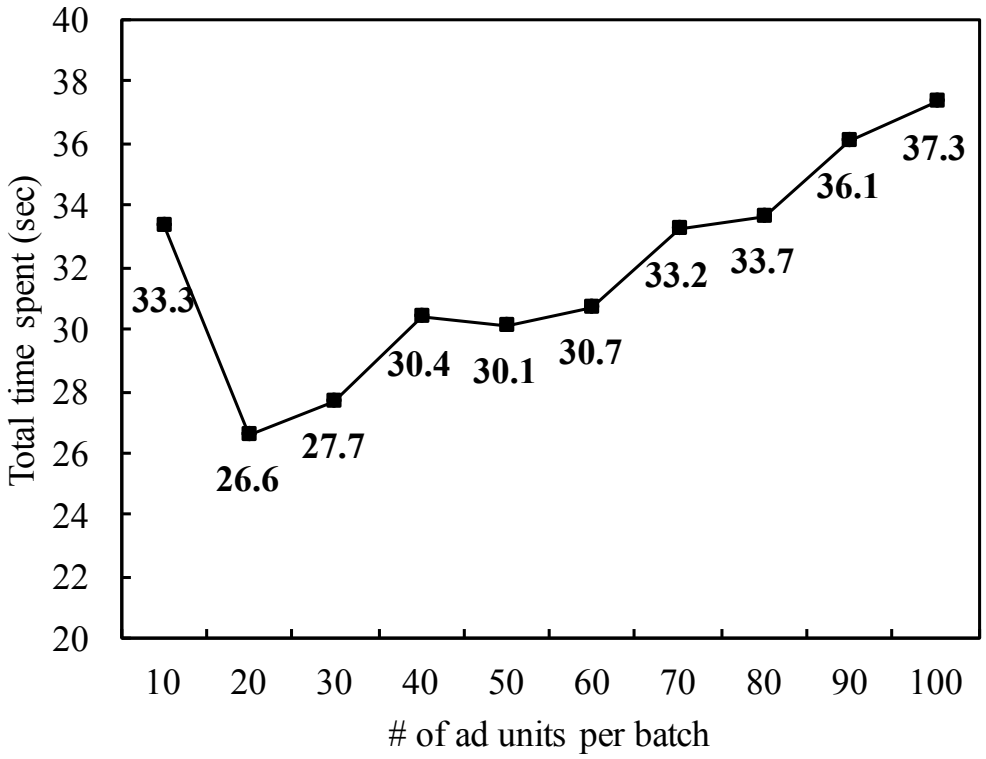


Figure 2.8. Total time spent for fetching 180 ads

(Sec. 2.3.3). Unfortunately, tracking scripts are usually executed after the page is fully loaded, which significantly drags down the attack performance. To improve its efficiency, we use the *tag-only training pages* extracted from the original pages that only contain the tracking scripts.

Fig. 2.7 shows how tag-only training pages are built. We get the HTML source code from the fully rendered page and extract JS snippets whose tags match pre-collected tracking tag signatures. We then build a tag-only training page using the extracted scripts and the mandatory DOM elements such as `<html>`,`<head>`, and `<body>`. The snippets at the bottom in Fig. 2.7 is the example output.

Ad Fetch Page Builder

An *ad fetch page* is an HTML file containing a set of ad slots. It is similar to the crafted page created for *ADHoneyClient* (Sec. 2.3.3). We configure each ad slots based on the collected ad specs. Besides, we need to optimize the number of ad slots per a batch for better efficacy. However, it is difficult to predict the appropriate number because the ad loading procedure varies. Therefore, we perform an experiment in this step to infer the optimal number of ad slots per batch.

To be specific, we compare the total time spent to fetch a particular number of ads. As explained in Sec. 2.3.3, we can only fetch a limited number of ads with a single browsing profile. Therefore, we use it as the upper bound in each experiment. For example, suppose the optimal fetch count is 180. We first fetch 10 ads each time and repeat for 18 times. Then, we try different batch size and compare the time needed to get all ads specified by the optimal fetch count (180 in this example). The results are shown in Fig. 2.8. We achieve the best performance when we fetch 20 ads per batch. Therefore, we include 20 ad slots in the *ad fetch page* in this example.

Attack Parameters

Attack parameters are a set of data used by the *attack engine* to customize the attack process. We may specify the *attack time* including the start time and duration. We can set the *attack strategy*, which can be *exhaustive* or *smart*. The exhaustive attack aims to drain the advertising budget as fast as possible. But it has high risk of getting detected. The *smart attack* is less aggressive and randomly sleeps to simulate human behaviors.

2.3.5 Attack Distribution

When the attack module is ready, it is deployed to virtual machines hosted on public cloud services, such as Amazon EC2[14], Google Cloud Platform[15], and Microsoft Azure[16]. Using public cloud services has the following advantages. First, it is cost effective. We can

launch the attack for a few cents per hour (Sec. 2.4). Second, we can evade the IP address based detections without additional cost.

The *attack engine* in the distributed attack modules repeats two operations: *browsing profile training* and *ads fetching*. It loads the *tracking-tag only pages* in sequence to train browsing profiles, and fetches ads using the ad fetch page. It repeats the whole procedure until the optimal fetch count is reached and disposes the browsing profile by flushing the cookies and local storages.

2.4 Evaluation

In this section, we describe the implementation and experiment results to validate the efficacy of our attack. We implement the *ADHoneClient* in Python based on the Selenium libraries [17]. The attack module is built as a chrome extension for easy deployment. The experiments are done on Microsoft Azure VMs. We choose the D1 v2 instances which provide the 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor, 3.5GB RAM and Windows Server 2016. The pricing plan for a single instance is $0.13 per hour.

We launch attacks against two types of advertisers. We first target at controlled advertisers, where we create the advertiser and set up the advertising strategies. The second experiment is to attack public advertisers in the wild (**after obtaining their approvals**), where advertisers run real-world e-commerce websites. More experiment details can be found in [18].

2.4.1 Controlled Advertiser

In this section, we evaluate our attack on the controlled advertiser created by us. As a valid advertiser served by a real-world ad network, we can get the actual numbers of ads being displayed, ad budget drained and the cost per $1,000$ impressions (CPM) to precisely calculate the financial damage. Besides, we can perform large-scale attacks without concerning about ethical issues. So, we can evaluate the full capacity using distributed VMs.

In particular, we created an e-commerce website that sells coffee beans and registered in an ad network. We run an ad campaign with a banner image and set weekly ad budget

Table 2.3. One hour attack against controlled advertiser

# of VM	ads	CPM	budget drained	cost	cost / drained	per VM	
						ads	drained
1	2977	$3.30	$9.82	$0.13	0.01	2977	$9.82
2	4965	$2.84	$14.10	$0.26	0.02	2483	$7.05
3	10114	$4.72	$47.74	$0.39	0.01	3371	$15.97
4	12485	$4.55	$56.81	$0.52	0.01	3121	$14.20
5	16875	$3.28	$55.35	$1.04	0.02	3375	$11.07
6	21264	$3.55	$75.49	$1.56	0.02	3544	$12.58
7	28483	$2.16	$61.52	$2.08	0.03	4069	$8.79
8	30484	$3.95	$120.41	$4.16	0.03	3811	$15.05
9	37880	$3.77	$142.81	$6.24	0.04	4209	$15.87
10	40958	$2.95	$120.83	$8.32	0.07	4096	$12.08
					Average	3506	$12.24

to $150. We target users who visit our product pages. To confirm if our ad is available to public, we visit the page to create a satisfying browsing profile. Then we visit popular websites and check if it can be fetched. As shown in Fig. 2.9, our ad is actually displayed at the right bottom corner on one of the top news websites, `nbc.com`.

We create an attack module performing *exhaustive attack* for an hour. Fig. 2.10 shows a batch of the ads fetched using the attack module, where most of the ads are from our advertiser. We also prepare a virtual image with the attack module installed. We create 10 virtual instances using the image in order to evaluate the distributed attack capability. Using the attack machines, we conduct 10 rounds of attacks with different number of attack machines.

Table 2.3 describes the result of the distributed attack against the controlled advertiser. The first column shows the number of attack machines. The second column shows the total number of our ads we fetched. We report the CPM, the budget drained and the cost. The result shows that we successfully fetched about $40k$ ads using 10 attack machines. Moreover, the number of fetched ads is increasing linearly with the number of attack machines. On average, we fetched $3,506$ ads per machine and drained $142.81 with 9 attack machines. We are able to drain 95% of the weekly budget within an hour. Note that we achieved better performance with 9 machines (instead of 10). The reason is that the CPM is measured

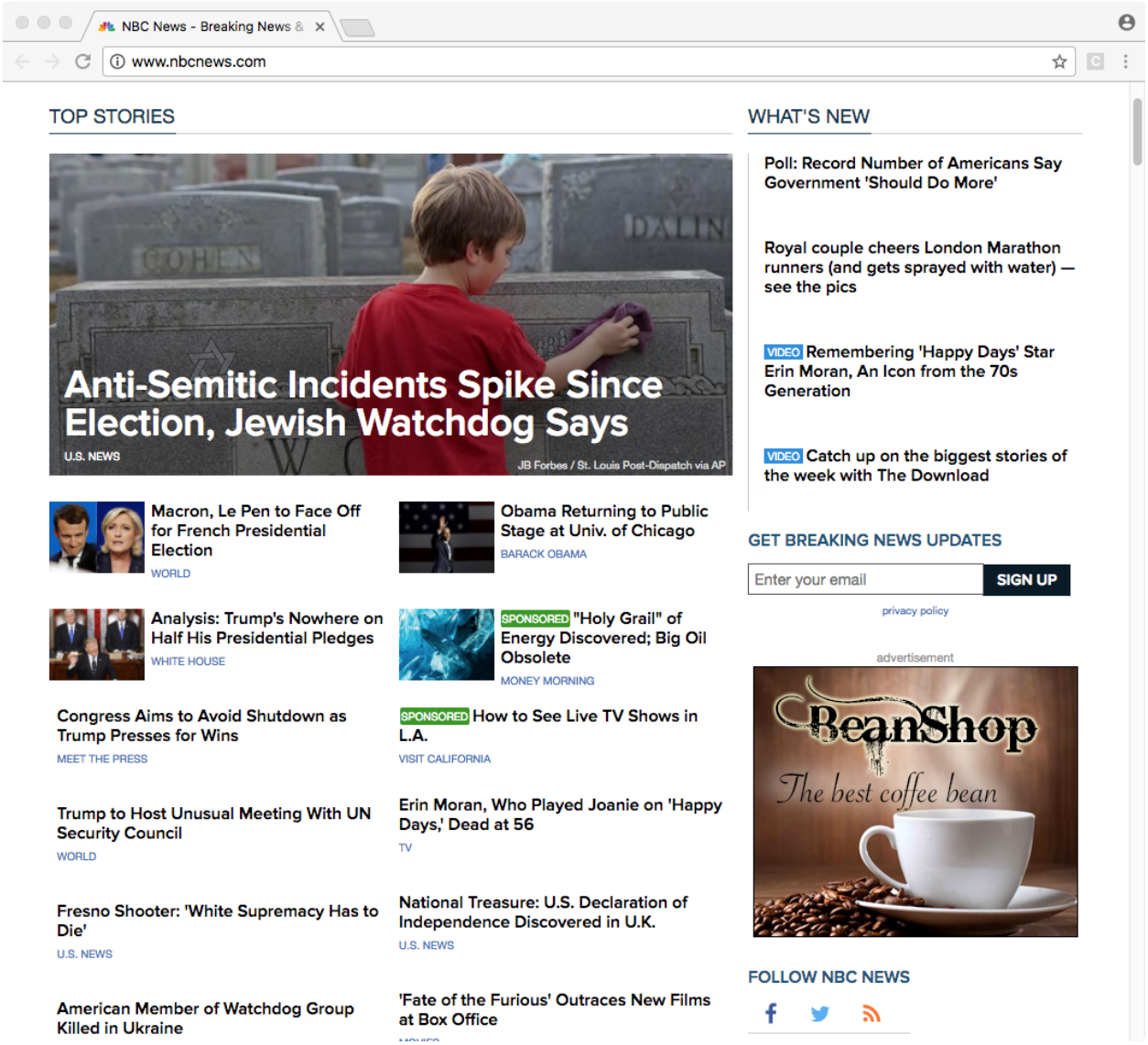


Figure 2.9. Our ad is displayed on popular sites like `nbc.com`

dynamically. Although more ads are fetched using 10 VMs, the drained budget is less comparing to 9 VMs ($2.95 vs $3.77). We report the ratio of the cost to the drained budget. The costs are merely 1% to 7% of the drained budget, which indicates that the attack using distributed machines on public cloud is extremely cost effective.

Figure 2.10. Ads fetched using the trained attack module

2.4.2 Public Advertisers

In this section, we evaluate the attack against real-world advertisers by executing attacks within the ethical and legal boundary.

Advertiser Selection

We target at advertisers who own e-commerce websites and use retargeting ad services. Although our implementation can be easily extended to support other ad networks, currently we focus on DoubleClick. Therefore, we filter out the websites listed in the shopping category in Alexa Top 500 [19] based on the following criteria: 1) websites do not have online shopping functionalities, 2) websites only providing posting and payment functionalities and 3) non-English websites. We also remove websites if they do not have ad tracking tags or

only support social networks/mobile ads. We use the remaining 291 websites to infer their targeting strategies.

As it may cause ethical issues if we run a large-scale attack to real advertisers, we reached out and requested permissions for a 10 minutes attack. We were able to get approval from 3 advertisers. We anonymize their identity and represent them as advertiser 1, 2 and 3 in the result. Besides, we only use a single attack machine for the experiments in order to minimize damages.

Revealing Targeting Strategies

The first step of the budget draining attack is to verify if the target is vulnerable. In our case, if we cannot reveal targeting strategies from a targeted advertiser, the advertiser is not vulnerable. So, we first reverse engineer the targeting strategies for each websites using our tool *ADHoneyClient*.

As shown in Table 2.4, we successfully revealed targeting strategies from 254 out of 291 websites (about 87%). The first column lists the targeting strategy categories. After targeting strategies are successfully reversed, we manually verify them the targeting strategies discovered and put them in proper categories. If we cannot interpret the intention behind the strategy, we mark them as `arbitrary activities`. The second column shows the average number of browsing activities in the targeting strategies. The third column shows the number of websites using the targeting strategies.

The results suggest that most advertisers mainly target the users who visit product pages. However, we can also see that 112 out of the 254 advertisers (about 44%) use sophisticated targeting strategies containing more than 3 browsing activities, which suggests that it is ineffective to get ads from such advertisers using the naive attack method like visiting only the product pages or the front pages.

We manually inspected why we failed on the remaining 37 websites (13%). We found that they either do not use the data collected from the tracking tags or deploy long-term targeting strategies (showing ads after a week) that is robust to a transient attack. Besides, some target users using geographic data, which is orthogonal to browsing profiles.

Table 2.4. Reversed targeting strategies

Targeting Strategy	Avg. # of activities	# of websites	Avg. training time (sec)		Rate
			Full page	Tag-only page	
Visiting a front page	1	31	4.78	0.70	6.87
Visiting a product page	1	111	4.37	0.57	7.65
Adding an item to a cart	3.48	79	8.55	1.17	7.29
Full shopping trip	4.94	28	19.59	2.00	9.77
Arbitrary activities	5	1	24.70	2.23	11.08
	6	1	22.98	2.11	10.89
	7	1	26.32	2.56	10.28
	8	1	29.34	3.09	9.50
	8	1	23.70	2.90	8.18
Total websites		254	**Avg. rate**		9.06

Table 2.5. 10 minutes attack result against selected public advertisers, and estimated damage

Advertiser	Ad Network	Targeting Strategy	Optimal Fetch Count	# of ad slots per batch	AD Size	Ad type	# of ads	Ad category	Estimated Damage	
									CPM	Budget drained/hour
1	A	Visiting a product page	180	20	300x250, 160x600, 728x90	Image	3134	E-commerce	$8.29	$155.89
2	A	Adding an item to a cart	180	20	300x250, 160x600, 728x90	Image	2742	Retail	$5.85	$96.24
3	B	Visiting a product page	90	30	300x250	Image	942	E-commerce	$8.29	$46.86

To validate the efficacy of the tag-only training approach, we conducted another experiment to show how significantly we improved performance comparing to the full page training. As described in Sec. 2.3.4, we create tag-only pages containing only tracking tags based on the targeting strategies revealed from the advertisers' websites. We record browsing profile training times using the tag-only pages and the original fully-loaded pages. According to the results in Table 2.4, the tag-only training is about 9 times faster on average.

Estimating Attack Damages

After we got the approvals, we launched the attack against 3 public advertisers. However, we cannot precisely obtain the numbers of ads displayed, CPM or budget because they are confidential business information. Instead, we use the public ad reports providing category-based average CPM for the first two quarters of 2016 [20, 21] and do our best to estimate the

damage. Although the estimation may be biased, we believe it approximately demonstrates how much ad budget we could drain with our attack against real-world advertisers.

Table 2.5 shows the result of the attack and the estimated damage. The columns 2 to 7 describe the output of the advertiser profiling. The column 8 shows the number of ads we fetched from each advertisers. We report the average CPM in column 10 and use them to calculate the estimated damage. The estimated budget drained within one hour (column 11) ranges from $46.86 to $155.89.

2.4.3 Ethical Considerations

We would like to highlight that we take the ethical issues seriously in our evaluation. This study was closely advised by a lawyer and conducted in a responsible manner. *Our evaluation process has been reviewed by IRB and we received IRB exemption.*

In the experiment of attacking a controlled advertiser, we own the advertiser's account and we pay for the charges. In the experiment with the three real-world advertisers, we explained our methods and potential damage to them. We start the experiments with their approvals. We purposely performed a proof-of-concept experiment using only 1 attack machine within 10 minutes to minimize the damage. We reported our findings and suggestions to them.

In spite of all of our efforts, due to the nature of the problem, ads from other advertisers showed up in our experiments. However, we confirmed that total rewards we collected from the untargeted advertisers as a publisher was less than $10. As the damages are distributed among all of the advertisers, the financial loss of one advertiser is negligible. More importantly, Google DFP is able to refund credits to advertisers when publishers violate their policies. We are in communication with DFP so that they can refund we earned throughout all of our experiments to the affected advertisers.

2.5 Countermeasures

In this section, we describe countermeasures against our attack. We introduce the detection and prevent methods.

2.5.1 Detection

In order to detect our attack, ad providers or ad networks can look for anomalies in ad request traffics. We discuss three possible detection approaches and their limitations in the following paragraphs.

Browsing profile based detection. The number of ad requests generated by a benign users is usually smaller than that from attack machines. Therefore, the number of ad requests per browsing profile can be used as a detection feature. For example, if a large amount of requests with the same browsing profile within certain period are observed, we can consider this as the attack situation. However, this feature may not be effective since our attack does not use the same browsing profiles for many times. Another viable feature is the number of browsing histories in a single browsing profile. In order to increase efficiency of our attack, we only train a browsing profile with few pages in a targeted website. Those profiles created in that way contain the limited number of browsing histories, but benign users normally have larger number of browsing activities.

IP address based detection. IP address based blacklists can be used too. We can mark requests suspicious if an excessive number of requests are made from the same address. The attack using the short-time-use profiles may evade the browsing profile based detection, but it cannot bypass IP address based detection because requests are from the same IP address. However, as we discussed, attack machines created using virtual instances can have different IP addresses by simply rebooting them or leveraging publicly available proxies, which makes IP address based detection less effective.

Click-Through-Rate (CTR) based detection. CTR is a metric that measures the number of clicks advertisers receive from a certain number of impressions. Our attack generate a huge number of impressions without actually clicking them. Therefore, the CTR is low. However, this can be bypassed by inserting valid clicks between our attacks so that CTR can be increased.

Our attack is similar to distributed denial of service (DDoS) attack since both attacks generate a large amount of traffic using distributed machines. Although detecting the DDoS attack is not that difficult, the attack is still powerful due to the characteristic of distributed

attack. Once an attack machine is blocked, a newly created machine can continue the attack. Therefore, it is possible that our attack can be detected with various features. However, we believe it is extremely challenging to nullify the attacks.

2.5.2 Prevention

As it is challenging for ad networks to effectively suppress the attack, in this section, we suggest practical prevention approaches for advertisers. One possible solution is to use event-based targeting strategies. They can track users who actually scroll pages or stay on the website for a certain period of time. Such event tracking utilities are already supported by many ad networks [22, 23]. Although the methods may not completely prevent the attack, it can minimize the probability of being selected as a target.

2.6 Related Work

Browsing profile manipulation. There are two existing studies that explore attack mechanisms based on browsing profile manipulation. Xing et al. [24] proposed an attack where adversaries can change the customized content of services from popular providers, such as YouTube, Amazon and Google, by manipulating browsing profiles. They provided specific attack methods for each services, and showed what attacks could be possible. While the proposed method worked well, their study only showed possibility of the attack. In contrast, our approach provided more practical, and beneficial attack mechanism for advertisers. The second attack is presented by Meng et al. [25]. They proposed a fraud mechanism to increase ad revenue for publishers by injecting higher-paying advertiser websites to publishers' pages. Although they successfully increased the revenue, their attacking perspective is different.

Ad fraud and mitigation. Representative attacks and countermeasures were discussed in [26]. Recently, Stone-Gross et al. [27] performed a large scale study on fraudulent activities in online ad exchange and suggested practical detection methods.

Recent research are focused on specific fraud activities. Among them, click fraud/spam is the most popular one. Dave et al. [28] proposed a method for advertisers to measure click spam rates and conducted a large scale measurement study of click spam across ten major

ad networks. Faou et al. [29] proposed a click fraud prevention technique using the value chain, the links between fraudulent actors and legitimate businesses. Their results showed that pressuring a limited number of actors would disrupt the ability of click fraud. Recently, Jaafar et al. [30] proposed FCFraud, a method for detecting automated clickers from the user side in order to prevent from becoming victimized attackers of click fraud. It analyzes web requests and events from user processes, classifies ad requests, and detects fraudulent ad clicks.

Another prevalent ad fraud activity is impression fraud/spam. Springborn et al. [31] showed an impression fraud via pay-per-view (PPV) networks by analyzing ad traffic from honeypot websites. Their results showed that hundreds of millions of fraudulent impressions per day were delivered by the PPV networks. Marciel et al. [32] proposed tools to audit systems of five major online video portals to investigate fraud in video ads.

Ad frauds also target on mobile apps. Crussell et al. [33] performed a study on mobile ad fraud perpetrated by Android apps. They developed MAdFraud, an automatic app analysis tool, to emulating event and extract ADs. They found that about 30% of apps made ad request are running in the background and identified 27 apps generating clicks without user interactions. Liu et al. [34] proposed a system to detect placement frauds that manipulate visual ads layouts to trigger unintentional clicks from users. They implemented a tool called DECAF and characterized the prevalence of ad frauds in 50,000 apps.

Online behavior tracking. Roesner et al. [35] investigated how third-party web tracking services performed. They showed how tracking worked, where the data can be stored, and how web tracking behaviors are classified. Englehardt et al. [36] proposed OpenWPM, a web privacy measurement platform, to show how third-party tracking cookies can be used to reveal browsing histories. Conti et al. [37] proposed TRAP, a system to unveil Google personal profiles using targeted AD. They focused on revealing topics a user is interested in instead of her actual browsing histories. Recently, Bashir et al. [38] showed information flows between ad exchanges using retargeting ad. They showed how user profiles were shared among ad exchanges by investigating 5,102 retargeting ads. Cahn et al. [39] assessed a

privacy threat caused by the third-party tracking. Our research was inspired by their study and we utilized them to build our attack method.

3. FINDING CLIENT-SIDE BUSINESS FLOW TAMPERING VULNERABILITIES

The sheer complexity of web applications leaves open a large attack surface of business logic. Particularly, in some scenarios, developers have to expose a portion of the logic to the client-side in order to coordinate multiple parties (e.g. merchants, client users, and third-party payment services) involved in a business process. However, such client-side code can be tampered with on the fly, leading to business logic perturbations and financial loss. Although developers become familiar with concepts that the client should never be trusted, given the size and the complexity of the client-side code that may be even incorporated from third parties, it is extremely challenging to understand and pinpoint the vulnerability. To this end, we investigate client-side business flow tampering vulnerabilities and develop a dynamic analysis based approach to automatically identifying such vulnerabilities. We evaluate our technique on 200 popular real-world websites. With negligible overhead, we have successfully identified 27 unique vulnerabilities on 23 websites, such as New York Times, HBO, and YouTube, where an adversary can interrupt business logic to bypass paywalls, disable adblocker detection, earn reward points illicitly, etc.

3.1 Introduction

The intrinsic complexity of the web ecosystem has created an attractive attack surface for manipulation and exploitation. Adversaries have exploited many common flaws that plague various entities in the ecosystem. Of particular interest are client-side business logic flaws. If exploited, they may lead to devastating consequences.

As a side effect of exposing partial business logic to the client-side, by perturbing the internal control flow of events, adversaries are able to change the intended behavior of a website and cause various kinds of damages. For example, suppose an application's ad delivery mechanism is developed with the intention of playing a sponsor's video before streaming the actual content. Malice can directly skip the first step to circumvent the business model of the website. Similarly, a website rewards airline miles after a participant fills out a survey.

An attacker can illegitimately earn miles without finishing the survey. A plausible approach to achieving this is to disable the condition check and force the execution of rewards logic, with the help of userscript manager utilities like Greasemonkey [40] or Tampermonkey [41].

Although OWASP strongly recommends enforcing business logic on the server-side [42], client-side implementations are commonly seen in practice. Sometimes, developers find it is easier to do so on the client-side without thinking too much about the consequences. But more importantly, it is unavoidable to devise some portion of the logic on the client-side to connect the dots in some scenarios. Web applications nowadays commonly integrate third-party content or services. In such cases, the client-side logic plays an important role in coordinating the internal states with those of multiple parties. For example, a web store may integrate a third-party payment service and implement the client-side logic to drive the checkout procedure [43]. For websites that serve a huge number of anonymous users (e.g., YouTube), it is very expensive to rely on the server-side to maintain the comprehensive states of all users.

Threat Model. We assume adversaries can manipulate client-side execution on the fly. For instance, they can trigger web page events in arbitrary orders. They can modify client-side scripts, change event handlers, bypass condition checks and alert, and send HTTP requests to servers. However, we assume they do not have access to servers so they cannot modify server-side logic.

Problem Statement. Web applications extensively incorporate code from multiple sources and thus it is desirable to understand the risks hidden in the client-side implementations. However, given the size and the complexities caused by JavaScript (JS) dynamic features and event-based executions, it is extremely hard to audit client-side scripts (including those from third parties) and identify the locations that are vulnerable to business flow tampering. To this end, we propose a dynamic analysis based approach to help developers focus on places that are more likely to be real vulnerabilities. By reporting the locations and the concrete tampering instances, our method can help developers effectively evaluate if actions should be taken to either relocate the logic to the server-side, deploy some runtime attack detection technique or incorporate additional client-side defense techniques.

In this chapter, we investigate the pervasiveness of the *client-side DOM-related business flow tampering vulnerabilities*, To the best of our knowledge, this is the first study to characterize the impact of the client-side business flow tampering vulnerabilities. As they are commonly caused by insufficient process validation, we propose an automatic detection method that addresses the following challenges:

- Pinpointing places vulnerable to business flow manipulation is difficult in multi-functional web applications.
- Dynamic web application features should be handled as code can be injected and generated on the fly.
- Code modification techniques and event-based dynamic executions make static analysis difficult.

In particular, our method systematically examines websites as follows: (1) Starting with business operation descriptions, we navigate the website and collect a set of functions that may be relevant to the business logic. (2) We analyze each candidate function and look for potential tampering locations, which may perturb the intended behavior if modified. (3) We develop techniques to select functions that are more likely to be vulnerable and generate tampering proposals for each selected function. (4) We revisit the website with the tampering proposals and confirm if the detection results are indeed business flow tampering vulnerabilities.

To understand the scope and magnitude of the vulnerabilities in practice, we evaluate our method on 200 real-world websites. We are able to detect client-side business logic tampering vulnerabilities on popular websites.Specifically, attackers can bypass paywalls and read an unlimited number of articles without paying on NYTimes and WashingtonPost. Detected flaws on Youtube and CWTV enable attackers to skip in-stream video ads. We also discover a flaw in the popular reward-earning website InboxDollars; attackers can illegitimately earn rewards points without finishing the required steps (e.g. watch videos). In our experiments, we are able to stack $3.44 reward for an hour attack with a single machine without watching videos, and if we continue this attack, we could steal around $80 per day.

In summary, we make the following contributions:

- We investigate the pervasiveness of the DOM related client-side business flow tampering vulnerabilities.
- We develop a novel dynamic analysis based approach to automatically identifying client-side business flow tampering vulnerabilities.
- We evaluate our method on 200 popular real-world websites. With negligible page loading/rendering overhead, we found 27 unique vulnerabilities, where an adversary can interrupt the business logic to bypass paywall, disable adblock detection, earn rewards illicitly, etc.

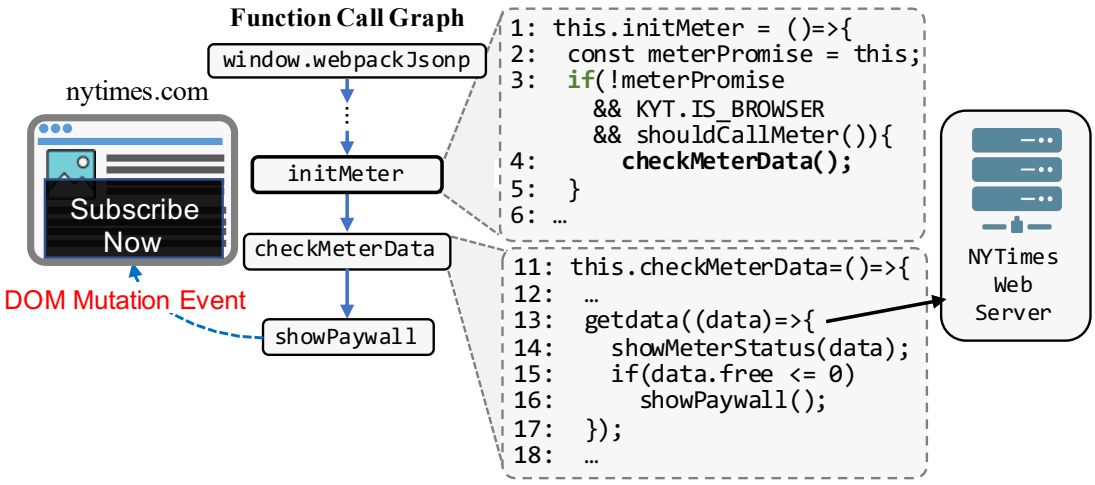
3.2 Motivation

In this section, we use two real-world examples to show (a) how business logic can be tampered with on the client-side, (b) why such vulnerabilities are common, and (c) why identifying these vulnerabilities is challenging.

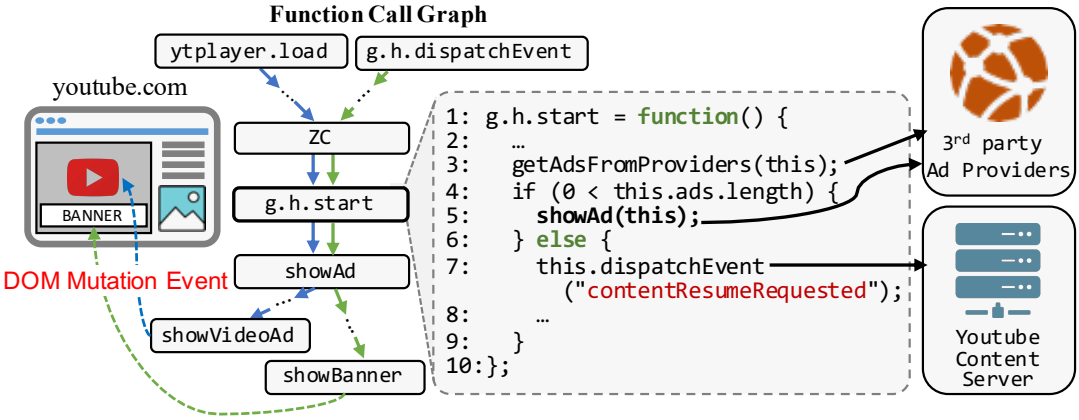
3.2.1 Bypassing a Metered Paywall

New York Times [44] (NYT) is a well known news publisher. Its main business model is a metered paywall. It allows users to read a limited number of articles for free. After that, paid subscription is required. NYT developers implemented the metered paywall in March 2011. Within three months, the system generated 224,000 subscribers [45]. With the paywall, after a user reaches the quota of 5 free articles, a subscription message box with a black-colored background covering most of the screen will be displayed.

Inspired by NYT's success, many publishers (e.g., Washington Post, The Boston Globe, and Chicago Tribune) adopted a similar paywall system. Fig.3.1(a) describes part of the simplified paywall implementation. Each time a news article is accessed, the article page is loaded as if there was no paywall. The paywall logic is implemented in a JS file loaded as part of the article page. In the JS file, function `window.webpackJsonp` invokes function `initMeter`, which further calls `checkMeterData` (line 4) that implements the paywall logic.



(a) NYTimes



(b) Youtube

Figure 3.1. Motivating examples

In the function, the meter data is first accessed (line 13). If the current user exceeds the free quota, function `showPaywall` (line 16) inserts the subscription message box. To bypass the paywall, the attacker can disable the function call `checkMeterData()`. Consequently, the subscription message box is elided and the attacker can continue to access articles for free. A demo video of the attack (hosted on an anonymous website) can be found at [46].

While OWASP recommends that critical access control should be performed solely on the server-side to avoid any client-side tampering, NYT's design simply delivers all the content to the client and relies on client-side access control to protect the content. Further

inspection suggests that there are reasons for such a flawed design. It looks like the paywall system was introduced as a well-encapsulated and stand-alone add-on (i.e., a self-contained JS file) to avoid any complex interference with the previous code-base. Properly implemented access control has to monitor each page load from the server-side, requiring substantial code changes. Furthermore, the number of free-readers is orders of magnitude larger than that of the subscribers. A correct design requires maintaining some profile for each free-reader on the server-side (e.g., the number of free articles accessed by the reader in a time duration), which would be much more expensive than the current design that only needs to maintain subscribers' profile. The current design relies on the client-side resources to deal with a large number of free readers. Such dilemmas are typical, leading to many flawed design and implementation as shown by our results in Section 3.5.

3.2.2 Skipping In-stream Ads

Ads revenue is crucial for business sustainability of streaming services like YouTube. Although service providers try to make money from other sources such as membership subscriptions, it turned out they often have to scale back and rely mostly on ads [47]. In particular, YouTube inserts ad videos before and in the middle of content videos. Recently, it even started showing Hollywood movies with ad breaks for free [48]. To implement this, YouTube has to use client-side logic because it needs to coordinate states among multiple parties and dynamically load videos from ad networks.

Fig. 3.1(b) shows a simplified version of the process. The ad videos are controlled by functions connected by blue arrows, while banner ads are managed by functions linked by green arrows. Function `ytplayer.load` is invoked during page load and eventually invokes function `g.h.start`. The function first gets available ads from third-party ad providers (line 3), then decides if ad videos should be played by checking if the list `this.ads` is empty (line 4). If yes, functions `showAd` and `showVideoAd` are called to play the in-stream video ads. Otherwise, it skips and plays the actual content (lines $7-8$). Filling up *this.ads* is by a separate thread in the background, controlled by a timer. Similarly, function `g.h.dispatchEvent` is invoked regularly to deliver ad banners via function `showBanner`. By enforcing the false

branch outcome at line 4, the ads are skipped and the user can watch the content video without watching the ads. Note that this is different from using an ad blocker to block ads. Many modern web applications are equipped with anti-adblocker mechanism, including Youtube [49]. Anti-adblockers often work by monitoring DOM object changes after ads are loaded. If no change is observed (meaning the ads are not displayed), it gets into a blocking mode requiring the user to turn off the adblocker. An anti-adblocker has to be closely coupled with the ad display function. In this case, the anti-adblocker is part of the function `shownAd()` (line 5). As such, by tampering with the JS code (i.e., line 4) directly, the ads, together with the anti-adblocker logic, are silently evaded.

A key feature of these business models is that the content publisher (or service provider) wants to ensure certain operations must be performed on the client-side, which cannot be trusted. This is not a new problem. There are many other rigorous business logics such as online shopping, credit card transactions, and online bank transactions. The key enabling technique in those business models is to associate a crypto-protected credential (e.g., token) with a user [50]. The token is shared by the multiple parties in the business model such that client-side operations can be remotely verified. For instance, an online shopping application has to interact with at least a remote payment service provider and a remote product provider. The payment is recorded by the payment service provider with the credential of the user. The product provider can *independently* check with the payment service provider to make sure the payment is in place before the product is sent. Such distributed integrity protection mechanism is heavyweight and often deployed in applications where users are properly profiled (e.g., users with accounts).

However, many web applications serve a vast number of users with most of them not properly profiled and hence do not have associated credentials. Nonetheless, the content publishers want their interest to be protected on those un-profiled users (by limiting their quota like in NYT or forcing them to watch ads like in YouTube). Such light-weight business models usually have to rely on client-side logic to conduct access control. In the YouTube example, ad networks are designed in such a way that any third party, including individuals, can bid for an ad slot (on YouTube). Most such third parties do not have the capacity to

support remote authentication (like the payment service provider in online shopping), which entails saving credentials of individual users. While a better scheme may be possible for lightweight business models in the future (e.g., through some centralized service like Google DoubleClick), client-side tampering is a realistic and prevalent vulnerability. As shown in the above examples, such vulnerabilities can lead to financial loss. If such attacks were launched at a large scale, websites may go out of business. Hence, it is in websites' best interest to identify such vulnerabilities so that they can take action to secure their interest, for example, by employing more expensive authentication schemes, deploying on-the-fly attack detection on the server-side, or even performing sophisticated client-side obfuscation.

Identifying Client-side Tampering Vulnerabilities Is Challenging. Client-side code usually comes from multiple parties. In addition, as suggested by the results in Table. 3.2, there are 8,307 JS functions on average in a single page load. Due to the overwhelming size and complexities (i.e., JS dynamic features, code minimization, and obfuscation) of the client-side code, it is impractical for developers to manually locate the vulnerable points. Developing an automated tool to expose such problems is necessary.

Therefore, we propose a dynamic search-based approach that applies a set of pre-defined tampering operations on client-side JS code. These tampering operations include enforcing branch outcomes, skipping or repeating functions. To reduce the search space, we develop an analysis technique to identify JS code elements that are likely to be business logic related and focus on tampering those. In particular, we observe that client-side business operations are usually correlated to DOM mutations. Hence, we intercept such events and collect the corresponding *candidate* JS functions. To deal with the prominent dynamic features of JS code, our technique is dynamic, modifying the underlying JS engine to instrument the internal intermediate representations of JS code on-the-fly, instead of directly instrumenting JS source code. For the aforementioned Youtube ad banner case, our technique selects 159 code elements as potential tampering candidates from 8,191 functions. The vulnerability disabling ad banners is identified after 10 trials. We have reported such problem to developers along with the NYT case.

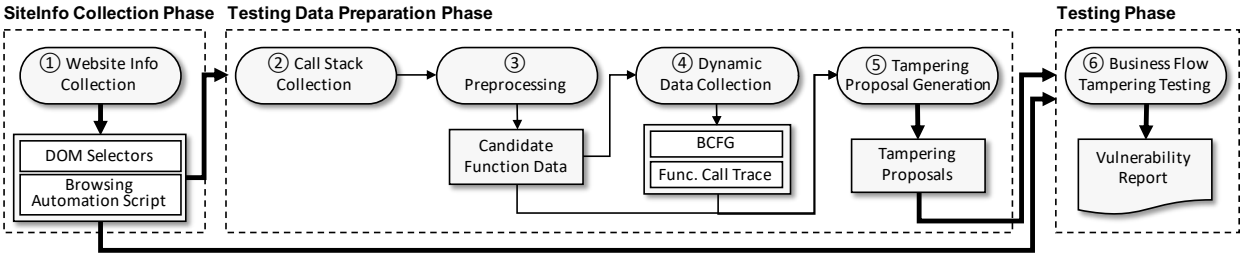


Figure 3.2. Approach overview

3.3 System Overview

Fig. 3.2 provides an overview of our vulnerability detection procedure, which can be largely divided into three phases.

Site information collection. By recording and inspecting user interactions, we collect basic information about the targeted websites. In particular, we identify DOM objects that should be monitored for mutation events and generate browsing automation scripts that allow automatic website navigation. They will be used as inputs to the whole procedure. Details can be found in Sec. 3.4.1.

Identify potential JS code elements to tamper with. We analyze the website and generate candidates for tampering. In particular, we monitor DOM mutation events and collect the corresponding call stacks. By inspecting the functions on the stack, we identify candidate functions that may be business logic related. The candidates are further ranked based on the estimation how likely they are vulnerable to tampering attacks. For each candidate, we generate potential *tampering proposals* that include the tampering points and the corresponding tampering operations. We explain the components and algorithms in Sec. 3.4.2- 3.4.4.

Vulnerability scanning by tampering testing. We repeatedly run the websites according to the generated tampering proposals to filter out proposals that cannot lead to tampering attacks. In order to reduce manual efforts to confirm if the outcomes are real attacks, we develop automated techniques to group test results, based on DOM event tracking

and clustering. Instead of examining all outcomes, testers only need to check one representative from each cluster. We produce a vulnerability report to explain the attack for each exploit. We explain the details in Sec. 3.4.5.

3.4 Design

In this section, we describe each component in detail and reason about our design choices.

3.4.1 Website Information Collection

Our system requires two pieces of information about the target website to start the procedure: (a) identifiers of DOM objects that are related to business logic and (b) browsing automation scripts. They are collected automatically by recording testers' interactions with the targeted websites. In the YouTube example discussed in Sec. 3.2, testers can record browsing activities to automate the operations such as "play video". In the meantime, testers can also specify elements or regions on the webpage that might be related to the business logic by simply clicking a button provided by our tool. Our system automatically collects DOM selectors that identify the DOM objects involved. Note that our technique does not require good code coverage of the application. Any test case that triggers the business model is sufficient. Due to the essential role of business model, a typical use case would easily cover it. Furthermore, such manual efforts are one-time. Our tool records the user operations in an automatic script that can be repeatedly executed in the scanning phase. Hence, the manual efforts required are minimal.

3.4.2 Identifying Potential Business Logic Related Functions

As testers already specified their areas of interest on the web page, we intercept the mutation events on the DOM objects and collect the corresponding (asynchronous) call stack. Then, we consider the functions on the stack that are more likely related to business logic and give them high priorities. In particular, when mutation events such as attribute updates, node modifications, or child DOM tree changes happen, a hook function will be invoked to collect the function call trace containing the functions that directly/transitively

trigger the changes. We exclude common JS libraries since they are unlikely tampering vulnerability candidates. We remove them using a whitelisting approach.

Note that we may observe different call stacks for the same mutation event. We hence construct a call tree, by merging the same functions in stack traces through a preprocessing step. Take the YouTube case in Fig. 3.1(b) as an example. The function "`showAd`" appears in two different traces. It has two different callees ("`showVideoAd`" and "`showBanner`") in the two traces. They are hence the two children of "`showAd`" in the call tree. *All nodes in the call tree of a relevant DOM mutation are considered candidates and given high priorities.*

3.4.3 Dynamic Page Data Collection

In order to overcome the challenges introduced by JS dynamic features, in this step, we collect runtime information about the candidate functions obtained in the previous step. In particular, we dynamically construct a *Business Control Flow Graph (BCFG)* for each candidate, which abstracts away path conditions that are unlikely to do with access control in business logic.

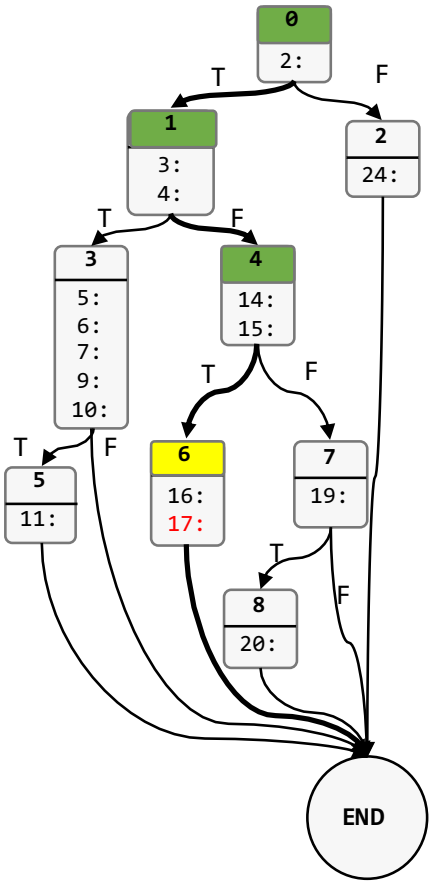
Business Control Flow Graph (BCFG)

The abstraction focuses on precluding predicates that are not related to business logic access control. Specifically, loop predicates are abstracted away as we consider loop predicates are unlikely to perform access control. While abstracting away loop predicates, we retain the loop body which may contain important function calls. *Note that BCFG is not intended to be compiled and executed. It is more a representation for us to enumerate the possible tampering schemes (called tampering proposals).*

After abstraction, BCFG mostly contains the following conditional statements: `if-then-else`, `switch-case`, and `conditional ternary operator`. A more heavy-weight analysis (e.g., one that leverages data-flow analysis) may have difficulty dealing with the various complex language features and the extremely dynamic nature of JS code, and hence is less desirable.

```
1  showAd = function(a) {
2    if (Bz(a, a.A)) {
3      var e = "contentResumeRequested";
4      if (Iz(a.o)){ //banner ad?
5        for(var i=0;i<a.o.n;i++)
6          Df(a.o);
7        a.dispatchEvent(e),
8        a.C.Jd(),
9        a.o.start();
10       if (null != a.F)
11         a.F.start();
12     }
13     else { //not banner ad
14       var f = a.o;
15       if (null != f && f instanceof gy)
16         a.C.ng(),
17         a.o.start(); //play video ad
18       else {
19         if (n)
20           a.dispatchEvent(e);
21       }
22     }
23   } else
24     a.dispatchEvent(e), Mz(a)
25 }
```

(a) Source code



(b) Business Control Flow Graph

Figure 3.3. Source code and Business Control Flow Graph (BCFG) of function `showAd` with each node representing a basic block with a unique id followed by the statements in the block

Example. Fig. 3.3a depicts a simplified version of the function `showAd` discussed in the YouTube motivating example. Function "`showAd`" renders ads differently based on the ad types (e.g., video ads or banner ads). At line 4, it checks the ad type (variable "`a.o`") and verifies if it's a banner ad. If so, the function resumes the player ("`a.dispatchEvent(e)`" at line 7) and displays the banner ad by invoking function "`a.o.start`" at line 9. Otherwise, the function plays the video ad by invoking function "`a.o.start()`" at line 17. Fig. 3.3b shows the BCFG of the function "`showAd`", where each box represents a block of instructions and each arrow denotes control flow. In particular, the execution path (BB_0, BB_1, BB_4 and

BB_6) represents the video ads delivery procedure. The yellow-colored block BB_6 contains the function call linking to the next function on the call stack. Besides, BB_6 is control-dependant on the green-colored blocks (BB_0, BB_1 and BB_4). Observe that the loop predicate at line 5 is abstracted away.

In addition to stack traces and BCFG, we collect DOM mutation types, function execution frequencies, positions in source code, and the source URL. Such information is needed in the later scanning phase. Note that we convert the position information (of a code element) in the row and column format (in the source code) to its IR offset used inside the JS engine as tampering is performed by the modified engine. For example, the position of function call statement "`showAd(this)`" in Fig. 3.1 is "`row:5, column:9`". We convert it to "`offset:89`" with 89 the IR identifier of the statement.

3.4.4 Tampering Proposal Generation

With the call trees of DOM mutations and the BCFGs of the functions in the call trees, the next step is to generate a set of tampering proposals that specify the code location to tamper with and the tampering operation. Although these functions and predicates have a higher priority compared to others, due to the large search space, we develop additional techniques to further rank the functions and predicates. In particular, we first rank functions using a learning-based method. Then the BCFGs of ranked functions are traversed in order to derive tampering proposals.

Candidate Function Ranking

As we will show in Section 3.5, the number of functions in the call trees of DOM mutations is still very large. Ideally, we would like to develop a technique to determine which of these functions are more likely to contain business access control. However, a solely program analysis based solution may not have the desirable effectiveness as runtime information provides strong hints. For example, a business access control JS file tends to be loaded before many other JSs; the function that performs access control often has high execution frequency than the content delivery function guarded by the access control; the URL of a

business model related JS file tends to share common domain name as the main page, etc. Unfortunately, these properties are uncertain and their importance is difficult to determine by humans. Therefore we propose a learning-based method to predict the likelihood of a function containing business access control. We then rank the functions based on their likelihood.

Feature selection. Based on our observation of the properties that are possibly important, we select 10 features, as shown in Table 3.1. We use tampering locations we already know (in a small number of web applications) to evaluate the significance of the features. To refine selected features, ANOVA F-test [51] was leveraged. The null hypothesis of the test is that the feature takes the same value independently of the output value to predict. As a result, highly significant features were chosen for our classification task. All features are normalized individually by subtracting the mean and scaling to the unit variance.

Estimation of likelihood by learning a classifier.

After selecting features, a classifier is learned from the training data. As our data may be biased, we explore using the Balanced Random Forest (BRF) [52], the weighted-SVM (w-SVM) [53], and the weighted-Logistic Regression (w-LR) [54], which are more interpretable than other classifiers (e.g., neural networks) and more robust when they have a small scale of training data. Balanced Random Forest (BRF) is an ensemble algorithm by a balanced

Table 3.1. Ten features for function ranking

ID	Features of Candidate Function `fn`
F1	Domain similarity between the website URL and `fn`'s script URL
F2	The loading order of the script containing `fn`
F3	The number of appearance of `fn` among all call stacks
F4	The position of `fn` on its call stack
F5	The collecting order of the call stack with `fn`
F6	The length of the call stack with `fn`
F7	The number of times `fn` is called
F8	The number of times `fn`'s callee is called
F9	The number of branches in `fn`
F10	`fn`'s callee directly mutates DOM (1: yes, 0: no)

bootstrapping method. In particular, we use 1,000 as the number of estimators. In addition, the gini impurity is used for split criterions. W-SVM (with RBF kernel) and w-LR use the ratio of class labels in their cost functions and put more weight on the rare cases to alleviate bias. In the testing stage, probabilities or regression values are used to estimate the likelihoods and are ranked by the scores.

To collect the training set for selecting features and learning the classifier, we first prepare a few confirmed tampering cases. After the training, we keep using the trained classifier to rank the candidate functions without any additional training. The result of the feature selection and learning will be discussed in Sec. 2.

Tampering Proposal Generation

Tampering proposals are generated by Algorithm 2. It takes two inputs: 1) the ranked candidate functions, where each function has its abstracted BCFG, the call site to its callees in the call tree, and the URL of the source code. 2) the *tampering strategy*, which can be either *bypass* or *repeat*. Intuitively, *bypass* skips a function call to see if the logic can be altered in the desired way, while the *repeat* strategy generates proposals that repeatedly invoke the callee.

The output is a list of tampering proposals indicating where and how the execution should be tampered with. In particular, a tampering proposal consists of (1) the URL of the script containing the candidate function, (2) the source code *offset* of the tampering point, (3) the *branch index*, and (4) the tampering *action*. The *branch index* specifies a branch outcome that should be enforced. For example, a basic block ended with an `if` statement may have two outgoing branches. If we want to execute the true branch next, we assign 0 to the branch index. Otherwise, we set the branch index to 1. Beside predicates, we may also tamper with the execution of non-predicate statements. In this case, the proposal simply specifies the location of the statement without the branch index information. The *action* indicates how the execution should be tampered with at a particular tampering point. The action can be *disable callee*, *disable caller*, *force branch outcome*, or *repeat callee*. Details of each action can be found in Sec. 3.4.5.

Algorithm 2 Tampering Proposal Generation

Input:

F: candidate functions sorted by the likelihood having tampering points. $f \in F$ is a function with the BCFG, the call site to its callee on stack, and the URL of the script having f.

ts: the tampering strategy, which can be *bypass* or *repeat*

Output:

T : tampering proposal (script_URL, offset, branch index, action) $\in T$

```
1:  function GenerateTamperingProposals(F, ts)
2:    T ← []
3:    for each f ∈ F do
        // f.callsite is the call site in f to its callee on stack
4:      co ← GetOffset(f.callsite)
5:      if ts is bypass then
6:        T ← T ∪ (f.url, co, none, "disable callee")
7:        B ← GetControlDepBasicBlocks(f.callsite)
8:        for each b ∈ B do
            // b.branch_cnt is the number of outgoing paths of basic block b
9:          for i ← 0 to b.branch_cnt do
10:           if HasPath(b, i, f.callsite.basic_block) then
                // skip the existing path from b to the callsite
11:             continue
              // b.branching_stmt is the last stmt before branching in b.
12:           bco ←GetOffset(b.branching_stmt)
              // generate a non-existing path starting from b.branching_stmt
13:           T ← T∪ (f.url, bco, i, "force branch outcome")
14:       fo ← GetOffset(f)
15:       T ← T∪ (f.url, fo, 0, "disable caller") // disable function f
16:     else
          // repeatedly invoke the callee of f on stack
17:       T ← T∪ (f.url, co, none, "repeat callee")
```

For each candidate function f, we first locate the locations where f invokes its callees observed on the stack (line 4). For example, in Fig. 3.1(b), the invocation statement at line 5 is the call site where function `g.h.start` invokes its callee `showAd`. Although a candidate function may invoke multiple callees in the execution, we separate them and create a trace for each invocation. Therefore, in our representation, a candidate function only has one call site to its callee in a trace. Under the strategy *bypass*, we generate a proposal that skips the invocation of the callee function (line 6). Then, we obtain the basic blocks that the call site control-depends on (line 7), where each basic block returned has a number of outgoing

branches (i.e. 2 branches for basic blocks with an `if` statement, and n branches for basic blocks ended with a `switch-case`).

Now we want to generate proposals that follow paths that are different from the one on the stack. To do so, we check if the call site is reachable through a particular path. Among them, we skip the path connecting the predecessor block and the call site (line 11). Since we want to explore the remaining paths even the path conditions are not met, we generate proposals for such paths (line 13) so that we can force the branch outcome.

The algorithm also creates a proposal to skip executing the entire function f (line 15). On the other hand, if the generation strategy is *repeat*, the algorithm creates a proposal that repeatedly invokes the callee function (line 17).

3.4.5 Business Flow Tampering Testing

After the tampering proposals are generated, we use a testing-based approach to confirm the real vulnerabilities. For each proposal, we leverage the automated script (recorded in the earlier phase) to load the target website. When the modified JS engine gets a script specified by the tampering proposal, it mutates the bytecode IR on-the-fly according to the action specified in the proposal. After a batch of tests, we gather test results and cluster them based on similarities. Finally, a tester confirms the success of the testing by checking the clustered results, which are usually just a few screen shots showing if the access control is circumvented. In this section, we first discuss how our system manipulates the business flow. Then, we describe how we filter out test results using DOM event tracking and clustering techniques in order to minimize manual efforts.

Tampering Actions

As mentioned before, there are four possible actions: *disable callee*, *disable caller*, *forced branching*, and *repeat callee*. Next, we explain how they are supported.

Disable callee. When the interpreter encounters the function call expression specified by the tampering location, it skips the call.

Disable caller. We disable the bytecode generation for statements in the function, which is equivalent to generating an empty function. This is because we still need the definition of the disabled function. Otherwise, the interpreter may crash if the disabled function is referred to somewhere. Disabling caller can be beneficial because it disables all function calls from other locations even not in the call stacks we collected. For example, in Fig. 3.1(b), if the website also plays the ad in the middle of playing the content, it can be turned off by disabling the execution of function `showAd`, instead of disabling all the function calls. Furthermore, callback functions triggered by native functions (e.g. event handler) or by external JS libraries can only be disabled by this method since call statements are not accessible.

Forced Branching. The branching target is forcefully set regardless of the result of the predicate condition. However, we still interpret the condition expression because it may have sub-operations (e.g. function calls).

Repeat callee. This tampering action is for duplicating desirable behavior (e.g. getting rewards) by repeatedly invoking the function. A naive approach to repeating callee would be to interpret the function call statement twice. However, business logic normally requires network interaction between the client and web servers. So, it is very likely the duplicated requests without interval will be ignored or considered as an error. We could add intervals by calling the `sleep` function at runtime in the JS engine. However, this may block the single-threaded JS engine and substantially interrupt the normal execution. To solve this problem, we use `setTimeout` function and register the function to be repeated as a callback function of the timer event.

Test Result Screening

After finishing each test trial, we need to check if the tampering proposal successfully alters the original business flow in an intended way. Since our approach relies on DOM changes, a simple solution is to track the existence of DOM mutation events. For the NYT example mentioned in Sec. 3.2, if the DOM mutation event that is triggered when displaying the subscription message box is reproduced in testing, this tampering proposal

is not successful. However, even if the DOM mutation event is not triggered, it does not mean that this test trial succeeds for various reasons. For instance, the tampered execution may stop showing the subscription message box, but it also blocks other DOM objects, such as the article content. It is also possible that the event is not triggered because the page is crashed. A more complex scenario is that the text message disappears, but the black box still exists. Since there could be countless outcomes depending on web applications, a tester's intervention is inevitable to make the final decision. In order to minimize the manual efforts, we group test results using a similarity-based clustering technique. Instead of asking testers to check every result, they can just check one in each cluster. Such number is smaller according to our experiment in Section 3.5. Furthermore, for the rest of the testing batches, the tester only needs to check when a new cluster is found. To be specific, for each test trial, when the original DOM mutation event is not triggered, we take a screenshot, get the corresponding HTML source code, and store as a test result entry. After testing a batch of tampering proposals, we group the collected test results with a similarity-based clustering algorithm. We use the Structural Similarity Index Method (SSIM) [55] for screenshots, and Tree Edit Distance (TED) [10] algorithm for HTML files to compute the structural similarity between DOM trees. As a metric of the clustering algorithm, we combine the two similarity scores since they complement each other. Specifically, clustering with the image similarity metric usually generates fine-grained clusters especially if a screen changes frequently. In the ad banner case from the motivating example, screenshots might vary depending on the screenshot taking time since the video content is being played; therefore, there would be many clusters. In this case, DOM tree similarity metric would reduce the number of clusters if we combine them together. On the other hand, image similarity shows better performance if DOM structures change dynamically, such as a front page of newspaper websites loading dynamic contents. As a clustering algorithm, we select Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [56]. The advantage of DBSCAN is it does not require the number of clusters as an input unlike other algorithms such as k-means. It is an important factor since we do not have any clues about how many clusters exist in the test results.

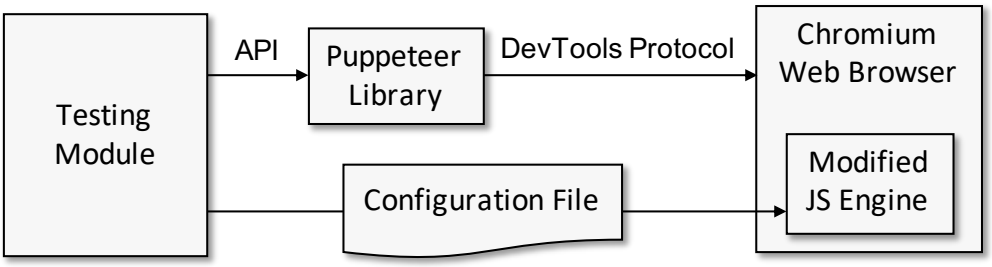


Figure 3.4. System modules and flows

Note that we do not need this step for the *repeat* tampering strategy, instead, we can simply check if the original DOM mutation event is triggered after the specified timeout.

3.5 Evaluation

3.5.1 Implementation

Our system[1] is implemented in Python and Node.js, and the modified JS engine is based on V8 6.6.74. Fig. 3.4 describes the modules of our testing system. The testing module leverages the Puppeteer library[57], which provides high-level APIs to control Chromium over the DevTools Protocol[58]. To collect dynamic data and perform the business flow tampering testing, we instrument the target JS code by modifying V8 engine [59] in Chromium. In particular, the instrumentation works as follows: once the V8 engine loads a script file, an Abstract Syntax Tree (AST) is built for each function and further translated to bytecode by the bytecode generator. We modify `InterpreterCompilationJob` class to generate BCFGs for JS functions after the ASTs are built. The `BytecodeGenerator` class is also modified to collect dynamic data and mutate execution. Comparing to code rewriting approaches [60], we modified the JS engine because it brings in additional benefits. First, it can easily handle dynamically generated codes as well as other sophisticated code modification techniques discussed in Sec. 3.2. Second, it has fewer side-effects. For example, it can work with code integrity checking techniques (e.g., Subresource Integrity (SRI) features [61]).

[1]↑We plan to make our system available at https://github.com/yirugi/JSFlowTamper

Table 3.2. Statistics of websites from 5 categories

Category	Total JS Size (KB)	# of JS	# of Functions	# of Branches
Newspapers	6,313	451	11,403	9,700
Magazine	4,334	258	8,046	6,884
Online Media	4,761	240	7,902	6,737
Surfer Rewards	2,521	132	3,931	3,330
Travel	4,402	220	7,031	5,854
Average	4,814	281	8,307	7,049

3.5.2 Research Questions

We investigate the following research questions in order to evaluate the effectiveness of our system:

RQ1. How much overhead does our system introduce?

RQ2. How many real-world business flow tampering vulnerabilities can our system find?

RQ3. How well do we estimate the likelihood of vulnerable functions? In particular, what are the results of the feature selection and the learning algorithm?

RQ4. How effective are tampering testing and result screening?

RQ5. How effective is our system on reducing search space?

To answer these research questions, we run our system on 200 real-world websites. The websites are collected from 5 different categories in Alexa Top 500 since they use the most common business models, such as advertisement, paywall, and point reward.

3.5.3 Experimental Methodology and Results

RQ1: Performance Overhead

Table 3.2 shows the benchmark statistics clustered by categories. On average, 8,307 functions can be observed in a single page load, which points to the needs of our approach. We measure three kinds of overhead, the first one is to collect the stack trace of DOM mutation events, the second is caused by the instrumentation to collect dynamic page information

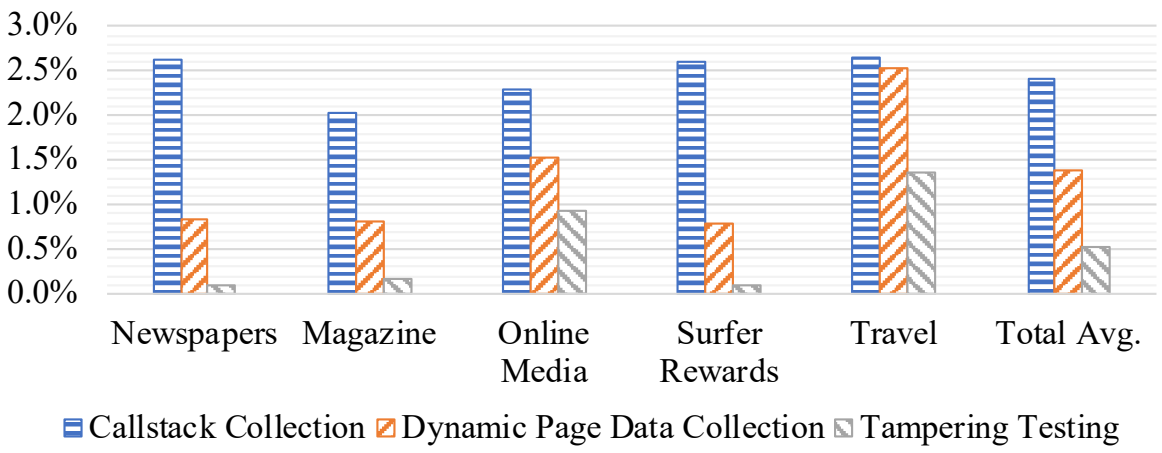


Figure 3.5. Normalized execution overhead

such as function execution frequencies, and the third is the tampering testing overhead. To reduce non-determinism caused by dynamic page content (e.g., ads), we crawl the pages and resources to a local directory, and then load the local pages and resources with and without our technique. The former is consider the baseline. We run each of the 200 websites 10 times and average the execution time. Fig. 3.5 depicts the normalized overhead. The first 5 sets of bars show the overhead observed in each category and the last set denotes the average overhead. The overhead for the call stack collection step is 2.41% (90ms) on average. We observed the average number of DOM mutation events triggered during page loading is 60.55. Hence, the overhead of handling one mutation event is around 1.5ms. Similarly, the overhead for dynamic page data collection step is 1.39% (70ms). We do not measure the overhead of executing tampering proposals for all 200 websites as it causes exceptions and early termination in many cases, skewing the real overhead. From the cases that terminate normally, the average overhead is 0.53% (4ms) which is lower than dynamic page data collection.

Table 3.3. Result of our testing on 200 websites

Case No.	Website	T.A.*	Vulnerable Operation	Case No.	Website	T.A.*	Vulnerable Operation
C-01	BostonGlobe	DCE	Paywall	C-14	CNBC	DCE	Anti-Adblock
C-02	NYTimes	DCE		C-15	CWTV	FE	
C-03	CWTV	DCE	Video Ad	C-16	CBS	FE	
C-04	FoxNews	DCE	Anti-Adblock	C-17	SeattleTImes	DCE	
C-05	NewsWeek	DCR	Offer Notification	C-18	MiamiHerald	DCE	
				C-19	DenverPost	DCE	
C-06	CBS	DCR	Video Ad	C-20	ETOnline	DCE	
C-07	Youtube	FE		C-21	AMC	DCE	
C-08	ETOnline	DCE		C-22	DallasNews	DCE	Paywall
C-09	AMC	FE		C-23	WashingtonPost	FE	
C-10	CartoonNetwork	FE		C-24	ChicagoTribune	DCE	
C-11	Fox	FE	Offer Notification	C-25	Youtube	DCE	Etc
C-12	PCMag	FE		C-26	HBO	FE	
C-13	Business-Standard	DCE		C-27	Inboxdollars	RC	

T.A: Tampering Action
*: FE = Forced Branching, DCE = Disable Callee, DCR = Disable Caller, RC = Repeat Callee

RQ2: Effectiveness in Finding Vulnerability

Our technique discovers 27 vulnerable cases from 23 websites as shown in Table 3.3. The first and the fifth columns show the case number while the second and the sixth columns describe the website. The third and the seventh columns indicate the tampering action, and the last ones contain the vulnerable business logic. Observe many websites are mainstream content publishers. We use the first 5 cases to produce the training set for the function ranking model. After we trained the classifier, we found 22 more cases. Besides the NYT case in the motivation section, we found 4 more vulnerable paywall systems (C-01, 02, and 22 - 24). Instead of using paywall, some websites show an offer notification popup at the front page. We found 4 cases that the offer popup can be disabled (C-05, and 11 - 13). From the websites in online media category, most of the findings are about skipping the ads before or in the middle of video playing (C-03, and 06 - 10). We also try to tamper with the protection method for their ad-related business logic against adblockers. We found 9 cases (C-04, and 14 - 21) in which the anti-adblock techniques can be bypassed. Youtube

Table 3.4. Function ranking with classifiers

(a) Function rank using 3 algorithms

Case No.	Avg. Rank of Func. w/ Tampering Point		
	w-LR	w-SVM	BRF
C-01	3.5	13.2	2.1
C-02	26.1	16.2	13.4
C-03	4.4	5.7	6.3
C-04	1.4	5.2	1.8
C-05	2	1.5	1.1
Average	7.48	8.36	4.94

(b) Function ranks of the 22 successful case (BRF)

Case No.	Rank of Func. w/ Tampering Point		Case No.	Rank of Func. w/ Tampering Point	
	BRF	Random		BRF	Random
C-06	1	82.5	C-17	3	10.4
C-07	4	90.4	C-18	1	3.8
C-08	8	47.4	C-19	1	9.3
C-09	1	31.4	C-20	9	46
C-10	6	25.2	C-21	4	69.5
C-11	5	19.4	C-22	1	7.1
C-12	1	4.3	C-23	2	63.2
C-13	3	4.6	C-24	1	9.1
C-14	1	5.6	C-25	1	3.5
C-15	1	8.5	C-26	2	12.3
C-16	8	26.3	C-27	11	12.4
Average				3.41	26.92

shows ad banners in the middle of video playing, and this can be skipped by disabling callee (C-25). HBO prompts a user to provide personal information in order to watch free episodes right before video starts. This can be skipped by forced branching (C-26). Inboxdollars gives points to users at the end of watching a video. Our system found a way to repeat the rewarding operation using the repeat callee tampering action (C-27).

We have uploaded demos of our findings (recorded screens of the successful tampered cases) to a private website[2]. We only use these findings for research purpose. We have responsibly reported the vulnerabilities to the victim websites and are in communication with them for possible defense solutions.

RQ3: Feature selection and learning algorithm

We use candidate functions from the first 5 cases in Table 3.3 as the training set. For these cases, we test every tampering proposals. If a vulnerability is found, the candidate function containing the tampering proposal is marked as a positive sample. The others are marked as negative samples. At the end, we acquire 56 positive samples and 402 negative

[2]↑https://sites.google.com/view/tampering-cases/

samples. Using the training set, we perform the feature selection process for the 10 features in Table 3.1. We conduct the ANOVA test to find out which features are significant. As a result, the first 5 features whose p-value is less than 0.1 are selected (F1, F4, F5, F7, and F8). Next, in order to check which learning algorithm works best for our scenario, we tested 3 classifiers discussed in Section 3.4.4 using the training set with the 5 websites. In particular, we learn each classifier on 3 randomly picked websites and test them on the rest of the websites for its cross-validation. In order to evaluate the learned models, we order candidate functions for each website based on the likelihood scores and got the rank of the first function containing the real vulnerability. We perform this evaluation test 10 times, then calculate the average rank values, and Table 3.4a describes the results of the 3 classifiers, as a result, BRF shows the best average rank value. Table 3.4b shows the function ranks of the 22 successful cases we find after we apply the trained classifiers (BRF). In order to evaluate its efficacy, we also select functions 10 times randomly, then get the averaged rank of functions containing the real vulnerability. As we can see in the table, the rank values with the classifier show significantly better performance than the random method. In 10 of the 22 cases, we find the vulnerability at the first candidate function using the ranking.

RQ4: Effectiveness of Tampering Testing and Result Screening

Table 3.5 shows the effectiveness of tampering testing and test result screening. The second column shows the total number of tampering proposals. The third column describes the number of tests until we find a successful case. The last two columns show the effects of the test result screening, the number of test results after DOM event-based screening, and the number of results after similarity-based clustering. The last column also indicates that the number of results requiring a tester's confirmation. In this experiment, we test 10 tampering proposals in one batch. As we mentioned, the first 5 cases are collected from randomly picked candidate functions, and the rest 22 cases are found with the help of the candidate function ranking method. In the first 5 cases (C-01 to C-05), the numbers of tests vary from 10 to 170. The worst case is almost 80% of the total tampering proposals (C-02), and on average, we test around 50% of the tampering proposals. The clustering and

Table 3.5. Function ranking and screening results

Case No.	# of T.P.	# of Tests to Success	# of Results after E.S.	# of Results after Clustering
C-01	264	170	32	4
C-02	191	150	107	17
C-03	176	90	42	16
C-04	150	20	13	4
C-05	208	10	8	2
Average	197.80	88.00	40.40	8.60
C-06	486	10	10	4
C-07	281	20	8	3
C-08	440	20	16	2
C-09	99	10	10	3
C-10	460	20	12	2
C-11	45	20	14	6
C-12	209	10	8	4
C-13	93	10	3	1
C-14	66	10	10	3
C-15	211	10	5	3
C-16	225	50	27	3
C-17	173	10	2	1
C-18	35	10	4	1
C-19	623	10	10	1
C-20	722	20	18	4
C-21	113	10	7	2
C-22	53	10	1	1
C-23	529	10	7	2
C-24	933	10	10	3
C-25	159	10	2	2
C-26	57	10	6	3
C-27	42	19	-	-
Average	275.18	14.50	9.05	2.57

T.P.: Tampering Proposal, E.S.: Event-based Screening

screening significantly reduce manual efforts such that testers only need to check 8 results on average, in comparison to the hundreds proposal executions. In the 22 cases found later (C-06 to C-27), the number of tests needed to expose the real vulnerabilities is tremendously reduced using the function ranking method. The average is 14.50, which is only 5% of the total tampering proposals. Because of the reduction, testers only need to check 2.57 results on average. In addition, we investigate the cases that do not have vulnerabilities, and we observe that testers have to check 27.93 clusters on average. As checking a cluster is as simple as inspecting a screenshot, we consider such manual efforts are manageable.

RQ5: Effectiveness in Reducing Search Space

In order to reduce search space, we collect call stacks by observing DOM mutation events, and we remove redundant functions and those from JS libraries. Moreover, the functions have zero call count during dynamic data collection are also removed in the tampering proposal generation. To evaluate the effectiveness of our filtering method, we collect statistics for the 27 successful cases. Specifically, we collect the total number of 3 types of data (JS files, functions, and branches) we have to consider before and after filtering. Fig. 3.6 shows the normalized numbers of each data types from 27 cases, and the last bar denotes the average

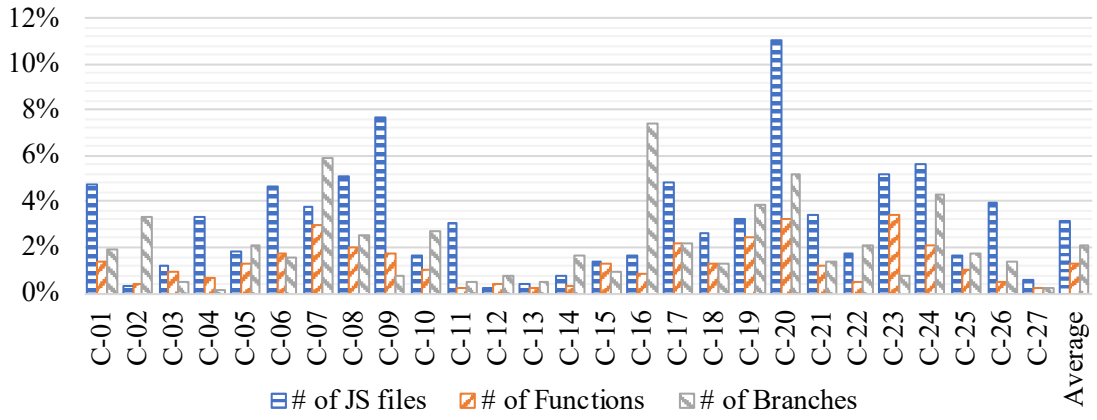


Figure 3.6. Effectiveness in reducing search space

number. As we can see, we could reduce the number of each data type substantially. For instance, we need to investigate 18,658 functions without filtering if we want to find the case C-22. However, after filtering, there are only 84 functions left, and this is only 0.45% of the original number. On average, we only need to inspect 3.18% of the JS files, 1.31% of the functions, or 2.13% of the branches of those in the original execution.

3.5.4 Case Study

In this section, we show two case studies to demonstrate how our system finds the business flow tampering vulnerabilities.

Bypassing Adblock Detection

The website we use in this case study (C-16) is `cbs.com` which is one of the biggest television networks in the US. They provide the subscription-based online streaming service,

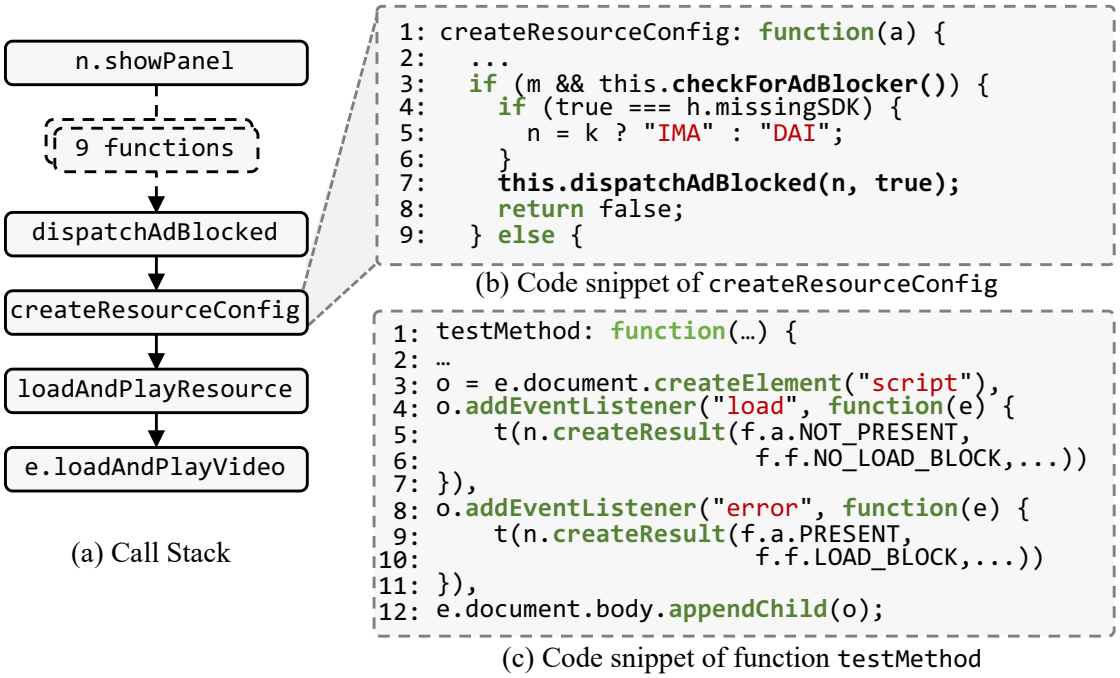


Figure 3.7. Bypassing adblock detection in cbs.com

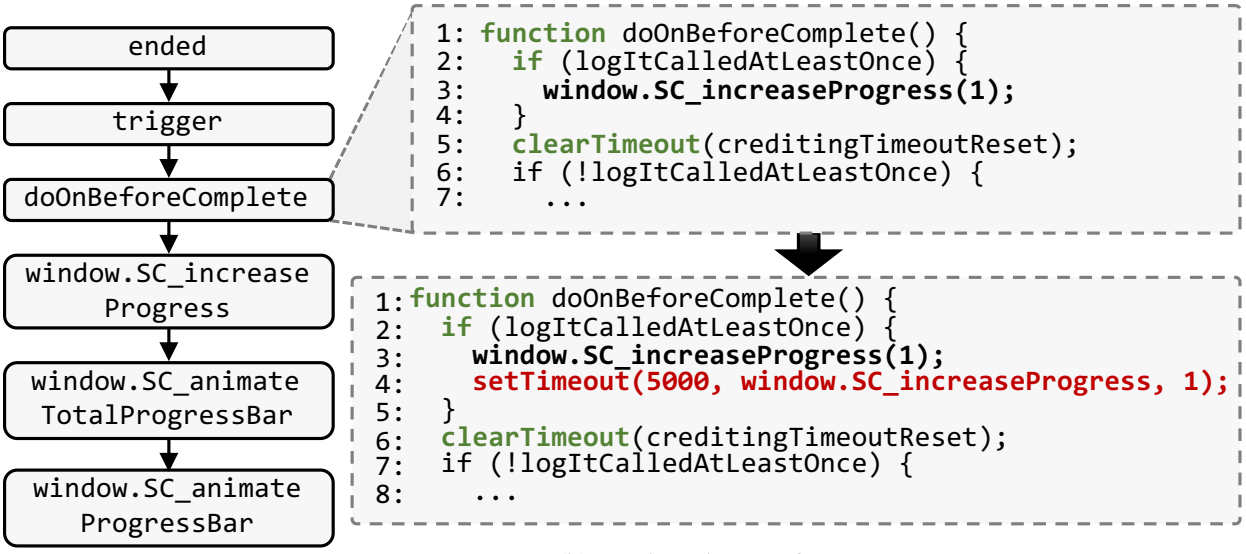
and some of their episodes can be watched for free with video ads from sponsors. They also protect their business logic using the anti-adblock technique. If a user tries to watch a free episode with an adblocker-enabled web browser, the website blocks the actual contents with a warning message.

In order to find if their anti-adblocker technique can be bypassed using our system, we collect call stacks by tracking the warning message. After preprocessing, we have 95 functions, which are only 0.86% of the total functions on the page. There are 225 total tampering proposals, and after 5 batches, which is 50 trials, we found the vulnerability. Note that after the screening, only 3 test results required a manual check.

To analyze how our system found the success case, we checked the vulnerability report. Our system found the tampering location in function `createResourceConfig`. Its call stack and code snippet are described in Fig. 3.7(a) and (b). It checks the presence of adblocker using `checkForAdBlocker` (line 3), and it calls `dispatchAdBlocked` to show the warning message (line 7). If we follow the function `checkForAdBlocker`, the function `testMethod` in Fig. 3.7(c) tries to inject a script containing "*ad*" string in its url (line 12) since the adblock applications usually block those scripts. The tampering proposal that forces the false branch of the if statement at line 3 succeeds. As shown, our system successfully found a way to bypass the anti-adblocker technique. With the tampered business flow, users can watch free episodes without watching video ads, and this would affect the business model of the website.

Repeating Point Reward

Inboxdollars (C-27) is an online marketing company that connects consumers and advertisers, and consumers can earn cash rewards for engaging in a variety of web activities. According to their website, total cash paid to members surpasses $50 million in 2016[62]. One of the services they provide is video reward; they offer points after a user watches a video containing ad. To be specific, when a video player reaches the end of the video, it increases a progress bar indicating the current reward status. In this case, we tried to repeat the rewarding activity. In order to start testing, we first recorded browsing interactions for logging in and clicking a play button, then gathered a DOM identifier by selecting the



(a) Call Stack

(b) Code snippet of `doOnBeforeComplete` before and after modification

Figure 3.8. Repeating point reward in inboxdollars.com

progress bar. The total number of functions that appeared during testing is 12,642, and we could reduce it to 27 which is only 0.21% of the total number. Our system successfully found the tampering location with 19 trials out of 42 tampering proposals. As we discussed in Section 3.4.5, it did not require the manual work to check the success case.

Fig. 3.8 illustrates the call stack and the code snippet of the function `doOnBeforeComplete` that has vulnerability. Specifically, when the video is finished, the player calls the function `ended`, then the function `trigger` calls the function `doOnBeforeComplete`. In the function, it first checks if the video has finished (line 2). If so, it calls the function `SC_increaseProgress(1)` to send a reward request to its server as well as to increase the progress bar. In order to repeat the call, the modified JS enigne added the setTimeout function containing the name of the function and a parameter, indicating the function will be called after 5,000 msec. Using this vulnerability, we could get multiple rewards after watching a single video. As mentioned, the rewarded points can be exchanged to actual cash. This directly causes financial damage to the website. We were able to stack $3.44 reward points for an hour attack

with a single machine, and if we continue this attack, we would get around $80 per day. We did not exchange the points we got from the vulnerability, and we are in communication with Inboxdollars so that they can deploy defense mechanism.

3.6 Threats to Validity

There are a number of threats to the validity of our conclusion. Part of our technique (i.e, function ranking) requires training. Although our training task is quite simple with well defined features, we only use 458 training samples. While the training set and the test set are strictly separated and our results indicate the effectiveness of the trained model, it may be possible that the training set is not representative and hence the ranking model may not be optimal. We will study the effect of including more cases in the training set in our future work. Our results are only acquired on 200 top-ranked websites as our technique is heavyweight testing-based, requiring processing thousands of dynamically loaded JS files and substantial dynamic contents. It is possible that these 200 websites are not representative. We plan to test on more websites. Checking the final results requires human efforts. It is possible that we may miss some real vulnerabilities. We currently only support simple tampering operations, which may not disclose complex business model flaws.

3.7 Related Work

Our work builds on extensive previous work on automatically testing web applications for vulnerabilities. We briefly describe relevant approaches, as well as previous works that detect business logic vulnerabilities in web applications.

Multiple Path Execution. Our work shares some similarity with recent work to explore execution paths by forcing program execution on JS programs [63], native binary programs [64], mobile apps [65, 66], and kernel rootkits [67]. Forced execution was first proposed in [67], which brute-forces control-flow at branches to explore program paths. X-Force [64] moves forward by designing a crash-free engine. In our work, forcing branch outcome is one of the tampering actions. However, our technique addresses a much broader problem. Guided mutation testing for JS web applications develops generic mutation testing

approaches based on common mistakes made by JS programmers [68, 69]. Our technique mutates places specific to business models. It features sophisticated methods to narrow down the candidates of such mutation. Symbolic and concolic execution based techniques [70–73] have also been proposed to analyze JS programs. Despite their great potential, handling substantial dynamic features in complex websites remains a challenge.

Business logic vulnerability detectors. Recently, researchers have proposed a number of techniques to test web applications for business logic vulnerabilities [43, 74–80]. These techniques focus mostly on the detection of web-based single sign-on systems and third-party payment systems. Wang et al. are the first to analyze logic vulnerabilities on merchant websites [43], and [75] studied logic flaws on popular web single sign-on systems. These techniques follow an API-oriented methodology that dissects the workflow in a particular application by examining how individual parties affect the arguments of related API calls. Sun et al. proposes a static detection of logic vulnerabilities in e-commerce web applications by combining symbolic execution and taint analysis to detect invariant violations of correct payment logic [74]. [77] proposes an approach to invoking static verification of the safety property of multiparty online services.

Dynamic JS code analysis. Dynamic analysis is commonly used to deal with the highly dynamic nature of JS applications. [81] builds a call graph on client-side codes embedded in server-side codes as string literals. It handles all possible client-side JS code variation by symbolically executing server-side code. AjaxRacer [82] detects AJAX event race errors in JS web applications by testing pairs of user events that are potentially AJAX conflicting. [83] proposes a dynamic slicer providing a comprehensive analysis to identify data, control, and DOM dependencies for client-side JS code. ConfictJS [84] finds conflicts between JS libraries by identifying potentially conflicting libraries and testing them with generated client applications that may suffer from the corresponding conflicts.

4. BFTDETECTOR: AUTOMATIC DETECTION OF BUSINESS FLOW TAMPERING FOR DIGITAL CONTENT SERVICE

Digital content services provide users with a wide range of content, such as news, articles, or movies, while monetizing their content through various business models and promotional methods. Unfortunately, poorly designed or unprotected business logic can be circumvented by malicious users, which is known as business flow tampering. Such flaws can severely harm the businesses of digital content service providers.

In this paper, we propose an automated approach that discovers business flow tampering flaws. Our technique automatically runs a web service to cover different business flows (e.g., a news website with vs. without a subscription paywall) to collect execution traces. We perform differential analysis on the execution traces to identify divergence points that determine how the business flow begins to differ, and then we test to see if the divergence points can be tampered with. We assess our approach against 352 real-world digital content service providers and discover 315 flaws from 204 websites, including TIME, Fortune, and Forbes. Our evaluation result shows that our technique successfully identifies these flaws with low false-positive and false-negative rates of 0.49% and 1.44%, respectively.

4.1 Introduction

Digital content services are web-based e-businesses providing users access to various online content, including news, entertainment, and technology articles. Those contents are delivered in diverse formats such as text, audio/video, or image. For example, Netflix, Amazon Prime Video, and The New York Times are well known digital content providers. Digital content services take up a significant portion of the e-commerce business. Specifically, the global digital content creation market size is estimated to be $11 billion USD in 2019 and is expected to reach $38.2 billion by 2030 [85].

Business Models of Content Service Providers. Content providers use a few *business models* to monetize their services. For example, news websites allow access to premium

articles only to the users who have subscribed or paid for the access. Social networking services such as Facebook make profits via advertisements instead of asking for payments from users directly. We define four business models as follows:

1. **Advertising** model delivers promotional marketing messages (i.e., texts, images, and videos) to users and content providers earn revenue from advertisers.

2. **Subscription** model typically uses a *paywall* method to restrict access to certain content for the users who have not subscribed or paid for the content.

3. **Donation** model relies on voluntary contributions to support service providers (e.g., giving donation money).

4. **Non-profit** model is usually adopted by organizations dedicated to public or social benefit (e.g., Wikipedia).

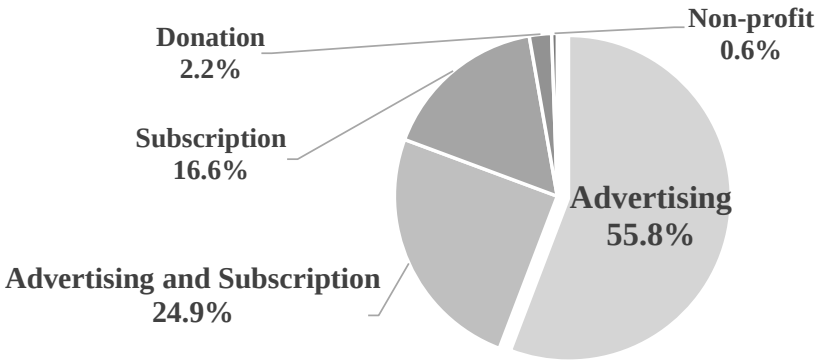


Figure 4.1. Business Models of 178 Digital Content Service Providers in Alexa Top 500.

Figure 4.1 shows the business models of 178 digital content service providers we collected from Alexa top 500[1]. Advertising (80.7%, including the websites with both advertising and subscriptions) is the most common business model, followed by a subscription (or paid content) business model. The result shows that the business models are common for digital content service providers, and advertising and subscriptions are the two most popular models.

[1]↑The remaining 322 websites are *not* digital content providers. For example, websites like Dropbox and Overleaf provide online *application services* (e.g., data creation and sharing functionalities), not focusing on delivering digital contents. They are based on the subscription model.

Promotional Methods. A *promotional method* is a strategy facilitating business models to maximize profits by either preventing adversarial techniques or directing users for payment.

1. **Anti-adblocker**: The advertising business model has been the most popular income source for digital content service providers. However, Adblockers which allows users to obtain contents without seeing the advertisements imposed a significant threat. *Anti-adblocker* is a promotional method that detects the presence of Adblockers to prevent users with Adblockers from accessing content. To access the content, users have to disable/uninstall Adblockers or purchase an ad-removal pass.

2. **Paywall**: *Paywall* is a promotional method used in the subscription business model. It restricts access to content and asks for a subscription. There are two types of paywalls: hard and soft. A *hard paywall* requires a paid subscription to access any digital content, and a *soft paywall* allows users to view the content a certain number of times before requiring a paid subscription.

Business Flow Tampering (BFT). A recent work [86] introduces the concept of *Business Flow Tampering* (BFT), which when successfully happens, allows an attacker to access content without going through a legitimate business flow (i.e., by changing the execution flow of the business model implementation). While it requires a strong adversary who is capable of monitoring and perturbing the execution of client web programs, the study shows that various digital content services suffer from the BFT.

The consequence of the BFT can be catastrophic. For example, a service provider that earns most of its revenue from subscriptions would go out of business if users can circumvent the subscription process (i.e., paywall). Moreover, a report [87] indicates that *BFT has become a real-world threat*: software or browser extensions aim to circumvent paywalls (e.g. [88]) are becoming increasingly popular. As a response, content providers put their effort into protecting their revenue by using techniques against BFTs. For example, almost 40% of the top 1,000 websites use anti-adblocker [89], showing the substantial interest of the content providers on the BFT.

The cause of BFT is essentially an improper business model implementation that relies on the insecure JavaScript execution (that can be manipulated by attackers) for critical

logic. Hence, it is crucial to identify the implementation flaws so that protection strategies can be applied. Unfortunately, a detection method outlined by the existing work requires substantial manual effort and domain expertise, hence not scalable.

Proposed Approach. In this paper, we propose an automated approach that discovers business flow tampering (BFT) flaws in the web client programs of digital content services. To handle various web services where implementations of them may vary, our approach leverages the fact that those web services share a few business models and their key business flows (i.e., processes). Focusing on the business model, we develop generic approach that is less dependent on concrete implementations of the web services.

Leveraging the business models, we propose a differential analysis-based technique to identify the BFT flaws. First, we run a web service twice where the first execution covers a legitimate business flow (e.g., accessing content with a subscription) while the second execution tries to do the same operation without going through the same business flow (e.g., without the subscription). Second, we perform a novel differential analysis on the two executions to pinpoint the critical implementation of the business flow (e.g., checking the subscription). Third, our approach automatically generates test inputs that can tamper with legitimate business flows and executes the web service with the test inputs to find the flaws.

The key enabling technique of our approach is a novel differential analysis technique that systematically locates the execution points that diverge, followed by execution mutations. Specifically, we mutate the execution of client-side JavaScript programs by adding, modifying, and removing statements. Our system also automatically validates test results using a clustering algorithm (i.e., Balanced Random Forest classification). Mutated executions (e.g., skipping subscription checking) achieving similar results to the executions of legitimate business flows (e.g., access to premium content with subscription) suggest there can be BFT flaws. To this end, our approach can automatically identify BFT flaws with little to no manual effort and human interactions.

In summary, we make the following contributions:

- We propose a novel system, BFTDetector, to find BFT flaws. It automatically exercises business process on a content provider's website to identify the execution points that can be tampered with.
- We generalize the business models and relate the models with website implementations, using the models to exercise and trace diverse business flows.
- We develop a differential analysis algorithm to identify a critical decision point of the business model by comparing call traces between multiple executions.
- We apply our approach to 352 real-world digital content service providers from Alexa top 500, and find 315 flaws from 204 websites including TIME, Fortune, and Forbes.

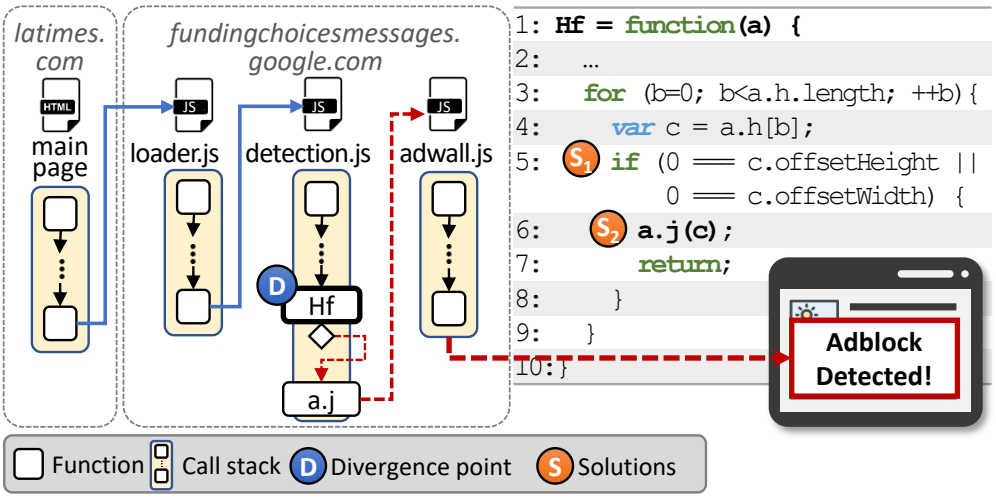
4.2 Motivation

We use two real-world examples, Los Angeles Times (LA Times) [90] and StudentShare [91], to demonstrate how our system can detect BFT in the real-world websites.

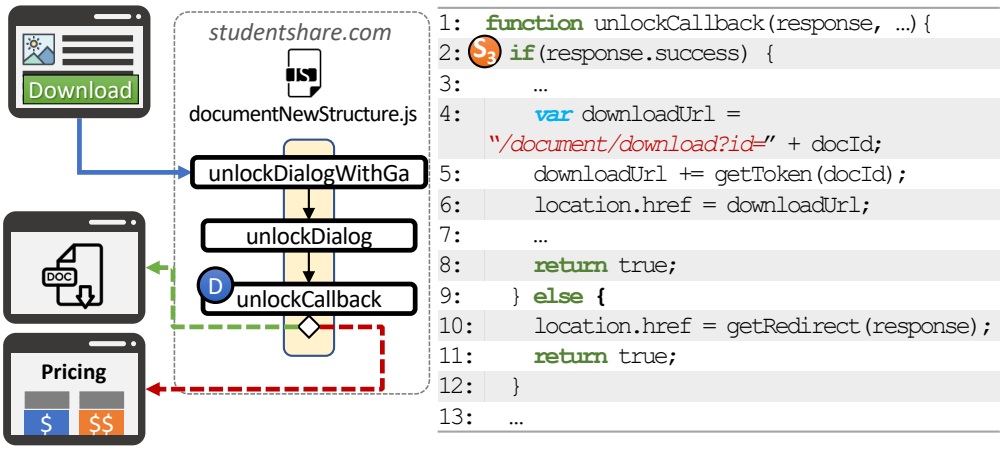
Advertisements on LA Times. LA Times is one of the most popular newspaper service providers in the US, and it uses advertising and subscription business models. Non-subscribers can see a limited number of articles by seeing ads on article pages. However, Adblockers can remove those advertisements, undermining the business model. To safeguard their income source, LA Times utilizes an anti-adblocker technique provided by Google Funding Choices [92]. When Adblockers are detected, the user is prompted with a message that directs the user to a subscription page, asking for payments.

Figure 4.2a shows the anti-adblocker process. When the main page of the website is loaded, it injects '`loader.js`' from Google Funding Choice. After a series of function calls, the loader script also injects '`detection.js`' that detects Adblockers. Specifically, the function `Hf()` gets a list of DOM objects containing ads (line 1). For each DOM object, it checks the sizes of the inserted ads (line 5), and if any of them are not being shown properly, `a.j()` is called. Lastly, '`adwall.js`' is injected to show the Adblocker detection message.

Subscription on StudentShare. The StudentShare site offers a large number of essay samples. It provides a limited number of free essays, and a monthly subscription is required



(a) LA Times



(b) StudentShare

Figure 4.2. Motivating Examples.

to access the premium essay samples. Figure 4.2b shows its business process for downloading premium essays. When a user clicks the download button, it invokes the function `unlockDialogWithGa()`, which further calls the function `unlockDialog()` to check if the user has access to the essay. Then, the callback function `unlockCallback()` is triggered when a response from the server arrives. It checks the variable '`response.success`' (line 2), and starts downloading if the variable's value is '`true`' (lines 2∼8). Otherwise, the user is redirected to a subscription page (lines 10∼11).

4.2.1 Business Flow Tampering Flaws

The two websites have BFT flaws. First, in LA Times (Figure 4.2a), the function call `a.j(c)` (line 6) that shows the Adblocker detection message can be bypassed by removing the call statement, or altering the result of the `if` statement (line 5). Second, in the StudentShare website (Figure 4.2b), any premium essays can be downloaded without purchasing the subscription by forcibly entering the `true` branch (lines 3∼8) of the `if` statement (line 2). These attack scenarios are highly achievable because the important business process written in JavaScript (JS) are *running on the client-side*, and an attacker can tamper with the flow using JS debuggers provided by web browsers. To this end, the tampering flaws can compromise the business well-being of these websites.

4.2.2 Business Model vs. Implementation

The underlying cause of the BFT flaws is a *discrepancy* between the assumption of the business models and the models' implementation. In other words, the business models do not assume the possibility of tampering with the processes, while the real-world implementation of the business models can be tampered with. Ideally, it is secure to implement the business models and the promotional methods with two principles: 1) important business process should be handled on the server-side, and 2) the client only displays final data rendered at the server. However, the above principles are not well obeyed in practice: (i) developers are often unaware or overlook the possibility that JS code can be tampered with on the client-side. In Figure 4.2b, decisions to initiate download or redirect to a subscription page are critical business logic that can be tampered with, as they are on the client-side. (ii) existing web ecosystems' complex internal structures make it hard to achieve the principles. For example, ad ecosystems today integrate multiple 3rd-parties and run complex bidding processes multiple times to provide effective interest-based ads. The ad ecosystems decide to run them on the client-side due to the efficiency (i.e., running them on the server will cause significant overhead).

4.2.3 BFTDetector: Automated Tampering Detection

Our approach automatically detects the existence of BFT flaws, including the location of the flawed code and its cause. Specifically, we automatically identify a business model of the website by analyzing execution traces of the website following different business flows (see Section 4.3.1). We then conduct differential analysis to identify divergence points of the executions across different business flows. For instance, we detect the function `Hf()` in Figure 4.2a and `unlockCallback()` in Figure 4.2b as *divergence points* (D) because the executions of the different business flow paths *become different* from the points (Post-Divergence).

Lastly, our technique tries to test whether the divergence points can be tampered with by forcibly executing a branch or skipping statements. Specifically, in Figure 4.2a (LA Times), our system visits the page with Adblockers, and try to mutate the original execution at the divergence point (`Hf()`) by *flipping the* `if` *branch* (S_1) or *skipping the call* '`a.j(c)`' (S_2). In Figure 4.2b (StudentShare), we attempt to download a premium essay without a subscription by *forcibly executing the* `true` *branch* of (S_3), as if it were part of the subscription flow.

4.3 System Design

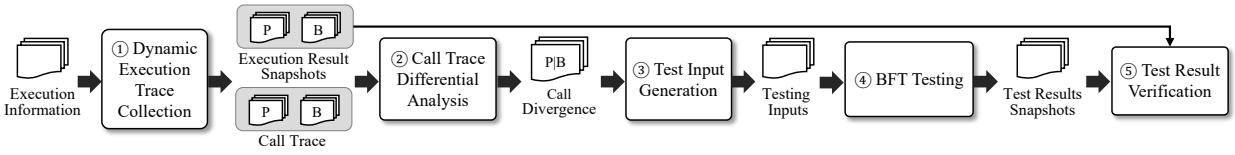


Figure 4.3. System Overview

Overview. Figure 4.3 shows a brief overview of our BFT detection system, which consists of five phases:

① Dynamic Execution Trace Collection (Section 4.3.1). BFTDetector collects dynamic execution trace by exercising business processes according to the business model. The output includes call traces and execution result snapshots which are essentially screenshots and HTML/DOM data.

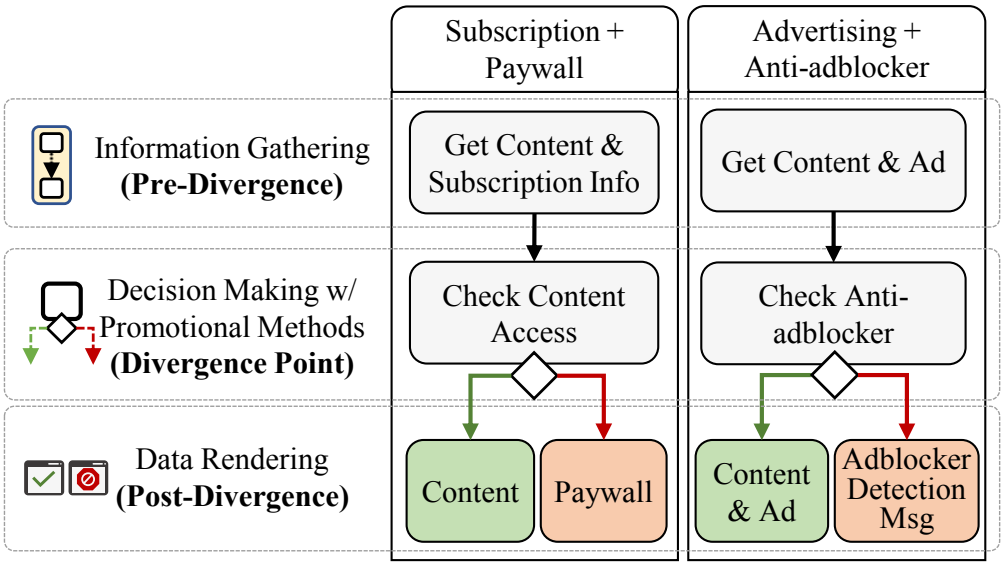


Figure 4.4. Generalized Business Process.

② **Call Trace Differential Analysis (Section 4.3.2).** Our system performs differential analysis on the function call trace collected for different business flows, identifying call divergences points where executions start to differ.

③ **Test Input Generation (Section 4.3.3).** We generate test inputs containing statements data to be mutated by using the call divergence points from the previous step.

④ **BFT Testing (Section 4.3.4).** Our system repeatedly visits the web page to mutate the execution according to the test inputs generated from the previous step.

⑤ **Test Result Verification (Section 4.3.5).** We measure whether our system successfully tampers with the business process by comparing snapshots from the test and the results from the original execution. A machine learning technique is used to calculate the degree of similarity between snapshots.

Table 4.1. Business Process Procedures

Procedure Name	Browsing Operations
$Login(JS)$	1. Open a browser 2. Perform logging in by replaying JS 3. Return the session S
$TriggerPaywall(P \mid JS, S)$	1. Open a browser with a session S 2. Visit all pages $\in P$ or replay JS 3. Return the session S
$CollectTrace(P \mid JS, S)$	1. Open a browser with a session S 2. Visit any page $\in P$ or replay JS 3. Collect execution trace & snapshot 4. Close the browser 5. Repeat 3 times*

*: In all the evaluated cases, we have reached a fixed point within three times repetition.

4.3.1 Dynamic Execution Trace Collection

Business Model Driven Trace Collection

Given a website using known business models such as advertising and subscription, we automatically exercise the website to execute the business process. Figure 4.4 shows examples of generalized processes of business models. The two diagrams on the right side represent the processes of two business models and the corresponding promotional methods, and the left side shows a generalized business process. The service providers first gather information and then decide with respect to the promotional methods and users' current states (e.g., whether a user made a payment or not). The business flow diverges as a result of the decision, delivering different contents to the users (e.g., showing premium content for a paid user, or redirecting to a subscription page for a guest). Observe that the decision-making logic causes business flows to diverge (i.e., divergence point), which can be tampered (i.e., BFT).

Table 4.2. Business Process Execution Driver

Business Model	Promotional Method	Browsing Procedure	
		Passing Run	**Blocking Run**
Subscription	Hard Paywall	① $S = Login(JS_{login})$ ② $CollectTrace(P_{sub}, S)$	① $CollectTrace(P_{sub}, \varnothing)$
		① $CollectTrace(P_{free}, \varnothing)^*$	① $CollectTrace(P_{sub}, \varnothing)$
	Soft Paywall	① $CollectTrace(P_{free}, \varnothing)$	① $S = TriggerPaywall($ $P_{paywall}, \varnothing)$ ② $CollectTrace(P_{free}, S)$
Advertising	Anti-adblocker	① $CollectTrace(P_{any}, \varnothing)$	① Enable Adblocker extension ② $CollectTrace(P_{any}, \varnothing)$

*: If free pages are also available.

Definition of Passing and Blocking Runs

To identify the divergence point in the business model, we first obtain executions covering two different business flows: a business flow delivering desired content and another flow blocking the content. Concrete executions of the two business flows are defined as *passing* and *blocking* runs.

1. **Passing Run.** A passing run is an execution that successfully delivers the digital content (e.g., an execution with a paid paywall or with advertisements displayed).

2. **Blocking Run.** A blocking run represents the business flow that blocks digital content delivery for various reasons (e.g. no subscription, or Adblocker detected).

For instance, successfully downloading the premium essay with a valid subscription in the StudentShare is a **passing run**, while redirecting to the subscription page is a **blocking run**.

Automated Business Flow Execution Driver

Our system automatically exercises business flows with respect to the business model to obtain the passing and blocking runs. We first define three key business process procedures,

where each procedure is a sequence of browsing operations (e.g., open a browser and visit a page) that can exercise key implementations of the business models when executed. We then obtain the passing and blocking runs by executing the business process procedures on the websites.

Variables of Business Process. We define five variables to describe business process procedures (and browsing operations).

1. P_{sub} is content pages requiring a subscription.
2. P_{free} is a list of free pages (accessible without a subscription).
3. $P_{paywall}$ indicates a maximum number of pages allowing free access of content, before it triggers a paywall.
4. P_{any} represents any content pages.
5. JS is a Puppeteer [57] script recorded by a tester providing automated browsing.

Business Process Procedures. Table 4.1 shows three business procedures that serve as building blocks for exercising the flows in the business model. Table 4.2 shows the browsing procedures for each promotional method to exercise the two distinct business flows (i.e., passing and blocking runs). Our system repeats the collection process three times in order to gather enough execution traces that contain business processes. Our system also supports replaying tester-recorded browsing activities (in JS file format) from the Chrome DevTool recorder [93]. This enables our system to emulate website-specific browsing procedures (e.g., logging in or clicking the download button in the StudentShare case (Figure 4.2b)).

Call Trace Collection

We collect function call traces during the execution driven by the business flow execution driver. On a function call, we record the (1) *Caller* function, (2) *Function Call Statement*, and (3) *Callstacks*. Intuitively, the call trace includes information about who (Caller) called whom/at where (Call statement) and in which circumstance (Callstack), and we call this set

of data a *call signature*. The callstack is stored as a hashed string to enable fast comparison in the differential analysis (Section 4.3.2).

Bytecode Level Instrumentation. Instrumenting complex and often obfuscated real-world programs is challenging. Hence, we modify the V8 JS engine [94] to dynamically instrument at the JS bytecode level (we have changed around 1,600 LOC). This design choice also handles various difficult-to-instrument primitives such as anonymous/asynchronous functions and dynamically generated code.

Performance and Space Optimization. Call trace collection incurs high overheads due to, in part, a high volume of function calls. To minimize the overhead, we optimize the built-in call stack collection procedure. Specifically, when we retrieve a full-sized call stack from the browser, it constructs an object containing various unnecessary information (e.g., metadata of scripts, functions, and stack trace), leading to substantial performance and memory overhead. Hence, we prune out the unnecessary items in the call stack. In addition, we deploy a blacklisting approach filtering out JS files that are not relevant to the business process of our interest, such as internal functions of common JS libraries (e.g., jQuery) or third-party tracking code. For those libraries, we only trace the interface functions in our call trace (i.e., the first call to the libraries). As we show in Section 4.4.4, the above optimizations successfully reduce the overhead by half. Besides the call trace, we also record a snapshot (screenshot and HTML/DOM data) of the resulting page for each run. The recorded snapshots are used in the test result verification step described in Section 4.3.5.

Contributions. BFTDetector automatically explores the business process (passing and blocking runs) of the target website using our business process execution driver. In addition, it collects dynamic execution traces efficiently with bytecode-level instrumentation and optimizations.

4.3.2 Call Trace Differential Analysis

Given the collected call traces of the passing and blocking runs, we perform a differential analysis to identify a divergence point representing the critical decision-making point in the business model. For instance, in Figure 4.2a (LA Times), `Hf()` is the call divergence point

Algorithm 3 Call Divergence Point Discovery

Input: P, B : lists of call traces collected from passing and blocking runs, where $P_\text{i} \in P$ or $B_\text{i} \in B$ is a list of call signature c, and $c_\text{i} \in c$ denotes a set (Caller, Call Statement, Callstack)

Output: CD : a list of call divergence data, where $CD_\text{i} \in CD$ denotes a set (Divergence function, Call statement, Passing or Blocking Run)

1: **function** ExtractCallDivergence(P, B)
2: $CD \leftarrow \{\}$
3: $P_{int} \leftarrow$ Intersection(P)
4: $B_{int} \leftarrow$ Intersection(B)
5: $P_{uniq} \leftarrow P_{int} -$ Union(B)
6: $B_{uniq} \leftarrow B_{int} -$ Union(P)
7: $PB_{int} \leftarrow$ Intersection($\{P_{int}, B_{int}\}$)
8: **for each** $pb \in PB_{int}$ **do**
9: **for each** $p \in P_{uniq}$ **do**
10: **if** $pb.Callstack \subset p.Callstack$ **then**
11: $CD \leftarrow CD \cup \{p.Caller, p.Call\_Stmt, \text{“}Passing\ Run\text{”}\}$
12: **for each** $b \in B_{uniq}$ **do**
13: **if** $pb.Callstack \subset b.Callstack$ **then**
14: $CD \leftarrow CD \cup \{b.Caller, b.Call\_Stmt, \text{“}Blocking\ Run\text{”}\}$
15: **return** CD

since the execution flows of passing and blocking runs reach the function, but only blocking flow continues to `a.j()`. Similarly, `unlockCallback()` in Figure 4.2b is the call divergence point.

Algorithm. Algorithm 3 describes how we identify the call divergence point. It takes two lists of call traces (P and B) that are collected in Section 4.3.1. Each element (P_i and B_i) in the lists contains the *call signatures*, consisting of (caller, call statement, and callstack) as discussed in Section 4.3.1.

We first obtain intersections for each list of call traces P and B (lines 3∼4). Intersection() gathers call signatures that exist in all the runs in a set, essentially pruning out execution flows that are not necessary. For example, assume that our system targets a newspaper website using the subscription business model. In the passing runs, we visit three subscription-only article pages with a paid account, and visit the pages without the account in the blocking runs. Intersection() identifies and keeps call signatures from essential business processes triggered every time such as checking subscription, filtering out processes that are not always appearing (e.g., a video available only in one of the article pages).

Then, we identify unique call signatures for passing (P_{uniq}) and blocking runs (B_{uniq}) at lines 5∼6 using the function UNION that combines call signatures. Specifically, to obtain P_{uniq}, we subtract the union of call signatures of blocking runs (UNION(B)) at line 5. Similarly, we obtain B_{uniq} by subtracting the union of P from the intersection of B at line 6. For instance, our major interest from the previous example is to identify and exercise the *exclusive business flows* that depend on the outcome of subscription checking. The subtraction procedure can prune out executions from common business flows, such as getting subscription data.

Next, it identifies call divergence points by leveraging the intersection (P_{int} and B_{int}; pre-divergence) and unique (P_{uniq} and B_{uniq}; post-divergence) call traces. In particular, we detect a divergence point if a function is (1) a callee of a common signature available in both runs and also is (2) a caller of a distinct call signature existing on one side of the runs, and (3) their context is identical. Specifically, the algorithm first gets the intersection of P_{int} and B_{int} (line 7) to obtain common call signatures on both runs (pre-divergence). Then, it checks whether the call stack before the divergence (PB_{int}) can be found in after the divergence (i.e., post-divergence represented by P_{uniq} and B_{uniq}) at lines 10 and 13. If it finds such a case, the caller of post-divergence is considered a call divergence point, and we store it to CD (lines 11 and 14). For example, `unlockCallback()` in Figure 4.2b (the StudentShare example) is the call divergence point. This is because (1) the call signature `unlockDialog()` → `unlockCallback()` is available in both passing and blocking runs (pre-divergence), and (2) there exist call signatures from post-divergence: `unlockDialog()` → `getToken()` and `unlockDialog()` → `getRedirect()`, (3) with the same call stacks.

Contributions. We propose and design a differential analysis to identify divergence points where critical business decisions are made. Our algorithm automatically finds divergence points that can be tested to find BFT flaws.

4.3.3 Test Input Generation

We generate test inputs that can potentially bypass blocking executions flows, or change them to passing flows by leveraging the identified call divergence points. A test input contains

```
1: function onArticlePageLoaded(){
2:  var user = getUserInfo();
3:  var contentMetadata =
        getContentMetaData();
4:  if(user.loggedIn == false){
5:    redirectToLogInPage();
6:  } else {
7:    if(user.accountType=="premium"){
8:      showContent();
9:    } else {
10:     if (user.remainFreeView>0){
11:       user.remainFreeView -= 1;
12:       showContentAndPaywallInfo();
13:     } else {
14:       showPaywall();
15:     }
16: }}}
```

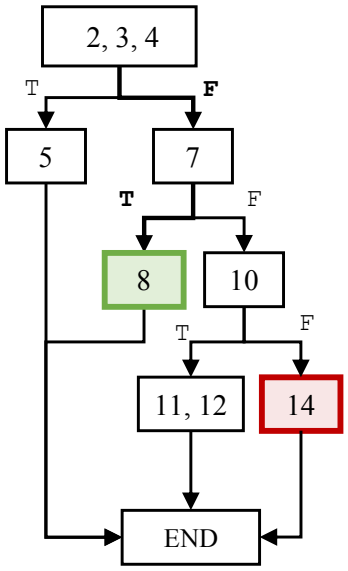


Figure 4.5. Illustrative Example for Test Input Generation

pairs of (1) a *mutation point* and (2) a *mutation action*. The mutation point indicates a position of an expression/statement to be changed, and the mutation action describes how to alter the point based on the type of the expression/statement. For example, if the type is a *conditional* (e.g. `if`, `switch`, `ternary`, etc.), the action can be an identification of the branch (e.g. `true`/`false`, or an index of a `switch-case`) that is to be entered forcibly. For a *function call* type, the action can be `skip`, which essentially disables the call statement.

Building CFGs for Mutation. We build CFGs for each function containing the divergence points in the call divergence data to compute control dependencies. If a call divergence is from a passing run, we generate a test input containing a list of mutation points and mutation actions that can drive the execution flow to the call divergence point. For a call divergence point from blocking runs, we first generate a test input that skips the call statement. Finally, we also generate separate test inputs that alter the branch outcomes.

We illustrate the test input generation process by using a simplified website with a softer promotional method as shown in Figure 4.5. Specifically, `onArticlePageLoaded()` is triggered when a user clicks an article page. It then gets information about the user and the

article (lines 1 and 2). If not logged in, it redirects to a login page (line 5). If logged in, it checks the account type and then shows the contents for premium users (lines 7 and 8). For a non-paid user, it shows the content with paywall information if the user's free view count is not used up (lines 10∼12); otherwise, it shows the paywall (line 14).

Assume that `onArticlePageLoaded()` is a call divergence point that has two branches caused by the call statements `showContent()` from the passing run at line 8 and `showPaywall()` from the blocking run at line 14. For the call divergence from the passing run, the call statement at line 8 is dependent on the statements at lines 7 and 4. Therefore, the generated test input that can trigger the call in any circumstances is `(4:false,7:true)`. On the other hand, in order to bypass the call statement at line 14 (which is dependent on the statements at lines 10, 7, and 4), we generate four test inputs: `(4:true)`, `(7:true)`, `(10:true)`, and `(14:skip)`.

4.3.4 Testing Business Flow Tampering (BFT)

We perform BFT testing to find whether executions with mutations can lead to a passing run. For each test input, our system runs the web application by following the automated browsing procedures described in Table 4.2. We then apply the mutation actions at the mutation point by intercepting the interpretation process and adjust the bytecode generated via the modified V8 engine. For instance, to apply the `skip` mutation action, we disable the bytecode generation for a target function call statements in '`BytecodeGenerator::VisitCall`'. For the conditional statements and expressions, we simply copy the same bytecode of the desired block to every branch outcome instead of changing the outcome itself. We record a snapshot (a screenshot and HTML/DOM data) of each test page for verification.

4.3.5 Test Result Verification

Once each test is completed, we examine the snapshots recorded in the dynamic execution trace collection and the BFT testing steps to verify the detected tempering flaws.

Crash Detection. Since our system forcibly mutates original execution flows, it may corrupt the execution context causing unexpected crashes, such as accessing undefined objects,

or calling function without proper arguments. We discard snapshots collected from crashed executions because the results might not be valid (and may mislead the classifier training process as well). We examine the amount of successfully rendered information to detect a crashed execution. Compared to the non-crashed execution (i.e., execution from the trace collection step in Section 4.3.1), if an execution renders substantially less information, we consider them as crashed. Intuitively, a crashed execution tends to terminate the execution before it renders all the elements. To estimate the amount of visually rendered elements, we take a screenshot of the webpage and leverage Shannon's entropy [95] that measures the level of complexity. For the HTML/DOM data, we utilize their content sizes. We combine those two metrics and compare them with average values from the original executions snapshots from passing and blocking runs. If it contains less than 40% of the non-crashing runs, we consider it crashed.

Test Result Classification. Intuitively, if a test result's snapshot (i.e., a screenshot and HTML/DOM) is similar to the snapshots of the passing runs, the mutated execution may indicate the existence of a flaw. Hence, we utilize similarity scores between the snapshots, and use them as a metric for a machine learning technique. We first extract common data available for each set of snapshots collected from the passing and the blocking runs, and these two data sets are used to check similarities. To be specific, we gather common pixels between the screenshots, then calculate the structural similarity index measures [55]. For HTML/DOM data, we compare the existence of DOM elements. This method using the common data is beneficial for computing structural similarities not disrupted by various contents inside. By doing this process, we can get a total of 4 similarity scores, 2 scores (screenshot and HTML/DOM data) from the passing and blocking runs each, and they are used as features of a classifier. We employ Balanced Random Forest (BRF) as a classification algorithm. Note that our training dataset is easy to be biased since the number of results containing flaws is much less than the not-flawed ones. We use the BRF classifier because it is designed to be robust for imbalanced dataset as it is less inclined to over-fitting. As a training dataset, we utilize flawed websites presented in the recent work [86]. We train the

classifier with a total of 1,778 snapshots collected from 13 websites using the subscription and advertising business models.

4.4 Evaluation

Implementation. BFTDetector [96] is written in Python and JS (Node.js). We use Chromium (91.0.4460) compiled with modified V8 JS engine (9.1.203). All experiments are performed on a machine with an Intel Core i9 3.60 GHz CPU and 16 GB RAM running Ubuntu 20.04 LTS.

Website Selection. For evaluation, we collect websites providing digital content services from various resources, such as Google News, Yahoo News, or Alexa Top 500, then select websites: 1) using one of the 3 promotional methods of the business models, 2) eligible for automated browsing, and 3) providing passing and blocking runs.

We classify the collected websites by the promotional methods. For the paywall methods, we first find websites having membership/subscription payment pages. If some paid content is accessible, it indicates the website uses a soft paywall method; otherwise, it is a hard paywall. For the anti-adblocker method, we utilize an adblocker browser extension, and if we observe content differences (except for advertisements) between websites with and without the adblocker extension, we classify it as anti-adblocker. If a website uses multiple promotional methods, we obtain each case per the methods. To this end, we selected 449 cases in 352 websites.

Research Questions. We evaluate BFTDetector to answer the following five research questions:

- **RQ1.** How effective is our system in detecting BFT flaws?
- **RQ2.** How efficient is our system in reducing search space?
- **RQ3.** How effective is our test result verification method?
- **RQ4.** What is the performance overhead of our technique?
- **RQ5.** How is our system compared to other approaches?

4.4.1 BFT Detection Results

Table 4.3 shows the result and statistics of the BFT detection. The first column shows the promotional methods. The numbers of websites for each method are in the second column, and the third column represents the number of flaws identified.

Table 4.3. BFT Detection Result and Statistics

Promotional Method	# Sites	# Flaws		# Funcs (A)	# Calls	# Divg.[1] (B)	Ratio (B/A)
Hard Paywall	45	31		13,245	1,408,472	93	0.70%
Soft Paywall	127	67		10,313	899,207	258	2.50%
Anti-adblocker	277	217		12,885	1,396,466	19	0.15%
Total	449	315	**Avg.**	12,148	1,234,715	123	1.02%

1: Divergences.

Discovered BFT Cases. BFTDetector identified 315 flaws. Specifically, a total of 31 websites with hard paywalls and 67 with soft paywall methods were found to be flawed, and this includes popular websites, such as TIME [97], Fortune [98], Automotive News [99], Forbes [100], and Bookmate [101]. Furthermore, we found flaws of the anti-adblocker methods from 217 websites. We manually verified the 315 flaws by following each website's business flow. For instance, for soft-paywall websites, we check if we can view articles more than the number of free access with the mutation. All flaws we found are deterministically and reliably exploitable. Details of the discovered cases including demo videos can be found on our website[2].

We reviewed the websites that our system was unable to find any BFT flaws. Since manual and thorough investigation is required, we selected 2 cases for each promotional method, a total of 6 websites as in Table 4.4. New Scientist and Journal & Courier does not have the BFT flaws since their business processes are operated in the server-side. On the other hand, we discovered that it was necessary to alter multiple locations simultaneously to bypass the hard paywall of AZ Central and the anti-adblocker of NY Daily News. From

[2]↑https://sites.google.com/view/bftcases

Table 4.4. 6 Websites with No Flaws Detected

Promotional Method	Website	Investigation Result
Hard Paywall	New Scientist	Server-side logic
	AZ Central	Multiple alteration needed
Soft Paywall	Journal & Courier	Server-side logic
	Orlando Sentinel	Dynamic execution
Anti-adblocker	Daily Herald	Unable to analyze (Large codebase)
	NY Daily News	Multiple alteration needed

the Orlando Sentinel case, we find that a few similar functions containing the same business process were being executed randomly. This protection technique, known as *cloning*, creates clones of basic blocks or functions that can be executed interchangeably by selecting one of them dynamically. Lastly, we failed to identify potential flaws in Daily Herald, due to, in part, the large and complex codebase (e.g., 7,175 functions).

Findings. The detection result shows that our approach is effective in finding BFT flaws; BFTDetector revealed 315 BFT flaws from real-world 449 cases.

4.4.2 Efficiency in Reducing Search Space

BFTDetector can pinpoint potential flawed locations from a large amount of functions. In order to show the efficiency in reducing search space, we collect the number of functions interpreted in a single run, and calls triggering them. We repeat the test 10 times for each web application, then calculate the average values. The fourth and fifth columns in Table 4.3 show the result of the test. The result indicates that there are 12,148 functions on average in a single run, and they trigger about 100 times higher number of calls. Since our system gathers call signatures from 6 runs (3 runs each passing and blocking sides), the average number of calls our system needs to handle would be about 6 millions. By using the huge number of call signatures, our system extracts call divergence by performing the call trace differential analysis we discussed in Section 4.3.2. The sixth column represents the number

Table 4.5. Test Result Verification

		Actual	
		Flawed	**Not Flawed**
Predicted	**Flawed**	**TP** = 1,645 (98.56%)	**FP** = 197 (0.49%)
	Not Flawed	**FN** = 24 (1.44%)	**TN** = 39,417 (99.51%)

of the call divergence our system discovers after the differential analysis, and there are only 123 divergences left after the analysis on average.

Findings. Our evaluation result shows that our approach reduces the search space efficiently (1.02% of the original number of function).

4.4.3 Effectiveness of Test Result Verification

In the course of performing our BFT testing on the 449 websites, a total of 42,128 snapshots were generated. As we discussed in Section 4.3.5, BFTDetector first checks if a test result is from a crashed execution. As a result of the crash detection, our system successfully filtered out 845 error snapshots. Furthermore, our test result classification process classified 1,842 of the remaining 41,283 test results as flawed. Specifically, we manually validate all the test cases and the classification results. If the prediction from our system is *flawed*, we revisit the website with the mutated execution and then check whether our system successfully tampers with the business flow. If it succeed, we consider the classification result is valid (true positive); otherwise, the prediction is incorrect (false positive). On the other hand, if the prediction is *not flawed*, we first compare the screenshots of the snapshots from the test result and the blocking run. If they are identical, the prediction is valid (true negative). Otherwise, we revisit the website with the mutation. If the new mutation triggers the BFT flaw, the prediction is not valid (false negative). If not, the prediction is valid (true negative).

Table 4.5 shows the confusion matrix of the test result classification. Within the 41,283 snapshots, our approach correctly classified 1,645 test results as flawed, while 39,417 are not flawed. The result indicates that our classification method using 4 similarity scores is

effective with a false negative rate of 1.44% (24 cases) and a false positive rate of 0.49% (197 cases). We investigated the 24 false negative cases, and found that most of them are from the anti-adblocker method. For instance, NWITimes [102] displays ads covering about 80% of the screen when the main page is loaded. If their anti-adblocker technique detects blocked ads, it shows a warning message. One of our test inputs was able to mutate the execution to prevent the warning message from appearing while the ads are not displayed. However, the ad space is also removed, allowing 80% of the screen to be filled by remaining content or a blank. The page is not similar to passing runs (webpages with ads) since the ad contents in the mutated run do not exist. It is also different from blocking runs (webpages without ads, but with an adblocker warning) because the blocking run screen is covered by the warning message.

Findings. The evaluation indicates that our verification technique successfully classifies test results with low false-positive (0.49%) and false-negative (1.44%) rates.

4.4.4 Performance Overhead

Table 4.6. Performance Overhead

	Interpretation			**Call Trace Collection**	
	Native	**Our Approach**		**Built-in Method**	**Optimized + Blacklisting**
Total	65.32 ms	65.76 ms	**Total**	13.32 sec	6.54 sec
Per Function	6.46 μs	6.74 μs	**Per Call**	10.74 ms	5.2 ms

Throughout the detection process of our system, there are two operations that can induce the overhead: 1) instrumentation, and 2) call trace collection. Our system instruments the tracking code by modifying the interpreter of the JS engine. Also, when the tracking operation is triggered, it collects a call signature containing the call stack. As Table 4.3 shows, there are 1,234,715 function calls on average in a single run, which indicates our system needs to retrieve the call trace data about one million times for each run. To measure

Table 4.7. BFT Detection using JSFlowTamper

(a) Detection Results on 315 Flawed Websites

Business Model	✓	✖
Hard Paywall	8	23
Soft Paywall	17	50
Anti-adblocker	77	140
Total	102	213

(b) Reasons of Detection Failure

Reason of Failure	# Cases
No DOM mutation event	63
No dynamic data collected	17
Random selector	84
No succeed tampering trial	49

✓: Flaws found, ✖: Flaws not found

the performance overhead, we record the elapsed time of the two operations for 10 times while our system performs the automated browsing, then we calculate the average values. Table 4.6 shows the experiment results. The first two columns of the first row represent the total execution time for a single run, and the second row indicates the interpretation time per function. The result shows that the code instrumentation only took 0.28μs per function $(6.74 - 6.46)$, and 0.44ms $(65.76 - 65.32)$ in total. Furthermore, the rest of the columns indicate the overhead caused during the call trace collection. The third column shows the results of using the built-in method of V8 JS engine as a baseline, and the last column denotes the results after deploying our optimized method along with the blacklisting approach as described in Section 4.3.1.

Findings. The result (i.e., reduce the overhead by half) shows that our optimizations are effective and BFTDetector can handle a heavy workload.

4.4.5 Comparison Study

We compare our technique with state-of-the-art technique JSFlowTamper [86] on the 315 flawed websites our system discovered. Note that we compare the source code of JSFlowTamper and BFTDetector to confirm that JSFlowTamper implements a subset of BFTDetector's

methods. It means that JSFlowTamper can only find the *same or fewer flaws* than the flaws BFTDetector detects. To this end, we focus on how many flaws JSFlowTamper can detect from the 315 flaws found by BFTDetector. Since JSFlowTamper does not provide automatic method, we manually prepared 315 sets of inputs including: 1) Puppeteer JS code performing automated browsing, and 2) DOM object selectors related to business process. We also manually reviewed the test results to verify the flaws, although it provides test result grouping to minimize human effort.

We determine the reasons for detection failure for JSFlowTamper as follows: 1) No DOM mutation event and No dynamic data collected: They are directly from JSFlowTamper's error messages, 2) Random selector: we observed that JSFlowTamper failed to identify prepared DOM selectors as the server-side code randomizes the selectors, 3) No succeed tampering trial: JSFlowTamper finishes without errors but no BFT flaws are found. This happens because the business model's core implementation is not related to DOM selectors (e.g., using predicates) or the core logic is executed without function calls which JSFlowTamper cannot handle. Table 4.7 shows the BFT detection result. As shown in Table 4.7a, JSFlowTamper was able to find the flaws only in 102 websites. Additionally, we examined 213 unsuccessful cases to determine the reason they failed, and each of them was caused by one of four reasons in Table 4.7b. The first one was caused when there was no DOM mutation events related to business process as in the StudentShare example (Figure 4.2b). Secondly, we also found that the system failed to collect dynamic data in 17 cases. The third reason was due to the random selector. Since JSFlowTamper utilizes DOM selectors to catch DOM mutation events, it cannot perform the detection if a targeted web application is equipped with randomization techniques, such as in [103]. Lastly, there were 49 cases where JSFlowTamper could not find flaws even after testing every trial. This indicates that the system was unable to locate functions that need to be tampered with.

Findings. JSFlowTamper can only find 32.38% (102 out of 315) of the flaws that BFTDetector can find.

4.4.6 Case Study

TIME.com

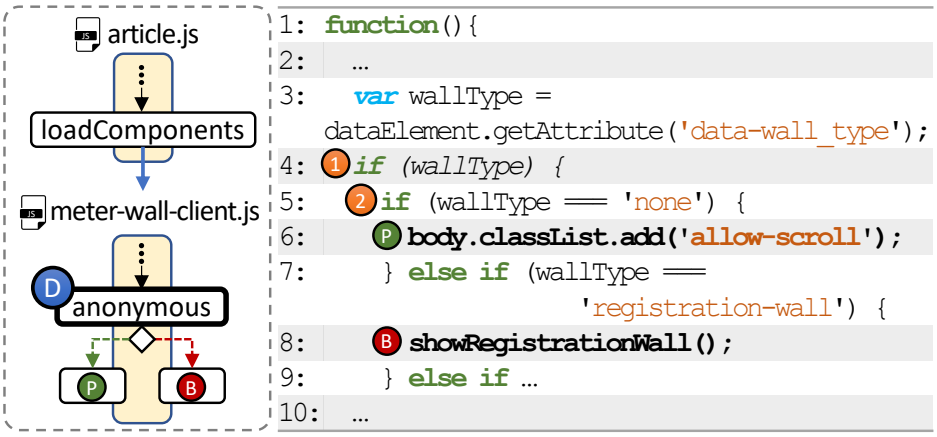


Figure 4.6. Business Process of Time.com

TIME [97] is a popular news magazine website employing a soft paywall for the subscription business model. It allows users to access 2 articles for free; after that, it shows a subscription message blocking the article page. To start test, we gathered 3 free pages; 2 for $P_{paywall}$ to trigger the paywall and 1 for P_{free}. Our system collected 129,774 call signatures on average for each run, and it extracted 156 divergence points in total. We observe 11,403 functions and 635,445 calls on average in a single run, showing that our approach efficiently reduced the search space. From the divergence points, we generated 124 test inputs and, after trials, found 1 input that allows us to access more than 2 articles without a subscription. Figure 4.6 shows the flaw. When an article page is loaded, `loadComponents()` in '`article.js`' injects '`meter-wall-client.js`' dynamically. After a series of calls, the logic inside the anonymous function (`function()`) determines whether to allow access for the article by allowing scroll (P) or to show a registration message (B). Our system successfully identified the divergence point (D), and found the test input that changes the blocking flow by forcibly taking the `then` branches of the two `if` statements (1 and 2).

Bookmate.com

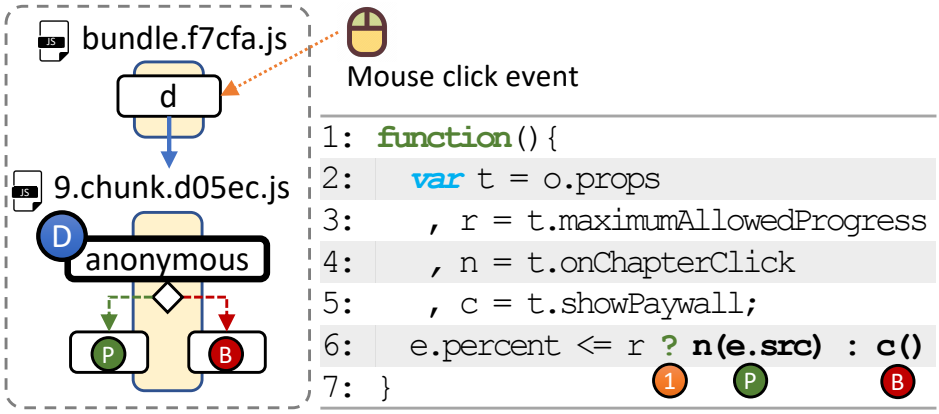


Figure 4.7. Business Process of Bookmate.com

Bookmate [101] is a social ebook subscription service, has 3 million readers and a catalog of over 500,000 books. They employ the subscription business model with the hard paywall method. The first one or two chapters of books are free to access, but users need to subscribe to a premium plan for $8 per month to read more. In order to trigger the subscription paywall, a series of mouse click events is required instead of just visiting a page. We recorded 2 Puppeteer scripts (JS_{free} and JS_{sub}) containing the browsing actions using Chrome DevTool, then fed them into our tool for replay. During the dynamic execution trace collection, our system collected 6,071 functions and 50,506 call signatures. After analyzing the collections, 7 test inputs from 2 divergence points are generated. To this end, we found 1 input that can unlock the chapter limitation. Figure 4.7 shows the divergence point and call stacks of the test input. When a user clicks a chapter, the triggered mouse click event is handled by the function `d()`. Then, it calls the anonymous function (i.e., `function()`) in the different script, which is a divergence point containing both paths to passing and blocking runs. The function gathers data, and checks if the clicked chapter (‘`e.percent`’) exceeds the maximum number of free chapters (‘`r`’) in line 6. If the clicked chapter is within the ‘`maximumAllowedProgress`’, it shows the chapter (P); otherwise, the paywall is displayed

(B). The test input our system found forcibly executes the `true` branch of the conditional expression (1).

4.5 Mitigation: Server side Code Randomization

As we discussed in Section 4.2.2, migrating *every important business logic* to the server-side to solve the business flow tampering flaws is not only impractical but also causing substantial overhead on the server side, leading to a high maintenance cost. Hence, in this section, we present, implement, and evaluate a *practical solution* (that does not cause high costs and substantial disruption in the existing service), which is a *server-side code randomization*. It generates new JS code each time a request is received from the client. Note that code randomization techniques are normally expensive. Hence, our solution is to leverage BFTDetector to identify the flawed logic, and apply the randomization on the identified code only.

Implementation. We implemented a proof of concept method to demonstrate the effectiveness of the mitigation approach. Specifically, we configure a proxy server imitating Google Funding Choice providing anti-adblocker service as in the LA Times case (Figure 4.2a). It intercepts requests from web browsers, then returns a JS file by applying the code randomization to the flawed function (`Hf()`) that our system discovered. To implement random code generation, we use an open-sourced JS obfuscator [104], which includes various anti-analysis technique (e.g. control flow flattening and string encryption). We test the mitigation approach on LA Times.

Result against BFTDetector. BFTDetector failed on the mitigation setup; it was unable to identify any divergences. This is because BFTDetector locates statements and functions using file offsets, that are randomized by the proposed mitigation. Also, BFTDetector analyzes branches to infer the business models, where our mitigation approach eliminates branches via the control flow flattening technique.

Result against Manual Analysis. The server-side code randomization also make manual analysis difficult. JS debuggers cannot set breaking points or track variables since locations of code and variables are constantly changing.

Efficiency. One concern of the mitigation approach can be a performance since JS code obfuscation techniques normally incur lots of computational and memory overhead. For example, the control flow flattening slows down the performance up to 1.5x [105], and the dead code injection increases the code size up to 200% [106]. To compare the performance overhead, we record the total time of the obfuscation operations applied only to the flawed function and to the entire code. As in Table 4.8, our mitigation approach increase only 287 bytes and 8.15 ms, which we believe reasonable.

Table 4.8. Performance Overhead of Mitigation Approach

		Vul. Func. Only	Entire Code
File Size	**Before**	184 B	64,316 B
	After	471 B	134,510 B
Time Overhead		8.15 ms	623.07 ms

Limitations. It is not immune to a code-reuse attack. Although we generate random code for every request, that does not mean that previously generated codes are invalid. Furthermore, if a flawed function contains only a few statements (e.g., a single call statement), the code randomization may not be effective.

4.6 Discussion

Ethical Considerations. The findings of this study are strictly for research purposes. Our disclosures do not include detailed information that could be used to reproduce the tampering. We have reported the flaws to all digital content providers, and we are actively in contact with them for potential mitigations.

Limitations. While our system is highly effective, it is also not free of limitations. First, BFTDetector performs the BFT testing using one input at a time. If multiple divergence points need to be mutated together (as shown in Table 4.4), our approach would fail to detect the flaws. Second, we use a file offset as an identification of JS objects (e.g. functions, or statements). BFTDetector may fail to locate JS objects embedded in HTML because the

offset varies based on its contents. Although we have not yet observed the cases in which important business logic is implemented in embedded in HTML, our differential analysis may miss divergence points in such cases. Third, while our test result verification shows low false-positive/negative rates, the 1,778 training dataset from 13 websites may not represent all possible cases.

Handling Soft-paywall Websites. In Section 4.4.1, we observe that BFTDetector detects fewer flaws in soft paywall websites. To understand the reason behind this, we inspected the 60 soft-paywall websites that BFTDetector could not find flaws and found the following 4 cases are observed frequently. (1) 14 websites require multiple execution mutations (the paywall is implemented across multiple files), which we do not support. (2) 7 websites are high-ranked Alexa websites. They use a protection technique called cloning. (3) 4 websites randomly decide the free-access policy (e.g., # of free-access pages), while we assume a deterministic policy. (4) 2 websites implement the business logic on the server side. For the remaining 33 websites, we found neither a flaw nor BFTDetector's limitations on them (probably not vulnerable).

4.7 Related Work

Testing-based Web Application Flaw Detection. Our work is closely related to automated web application testing for flaw detection. Black-box testing is widely used to generate test cases and check applications for vulnerabilities [76, 80, 107–112]. Testers analyze the system and create test cases to check if the test cases expose flaws. Previous work has employed black-box testing on web applications for various purposes, including detection of side-channel vulnerabilities [107], testing for checkout system flaws [80], feedback-directed automated test generation [113]. Their common goal is to improve the coverage of the execution space to discover buggy, abnormal or malicious behavior. Nonetheless, they are not suitable for detecting BFT flaws, which need to precisely pinpoint business logic related functions.

JSFlowTamper [86] is the state-of-the-art detection technique for BFT flaws. Unlike JSFlowTamper that only focuses on testing DOM selectors, BFTDetector defines and leverages

business models. The business models help discover new BFT flaws related to predicates and function calls, beyond DOMs. Also, BFTDetector proposes the differential analysis-based algorithm to automatically identify divergence points, while JSFlowTamper requires manual effort and domain expertise to identify DOM selectors. Furthermore, BFTDetector automates the end-to-end process, while JSFlowTamper focuses on manual dynamic testing. Lastly, BFTDetector solved JSFlowTamper's limitations: (1) handling randomized DOM selectors, (2) handling websites without DOM mutation events (e.g., Figure 4.2b), (3) detecting flaws related to multiple JS files in the call chain (e.g., Figure 4.2a).

JS Analysis Techniques. There are a variety of techniques analyzing JS code [63, 71, 73, 113–126]. Jalangi [71] provides a dynamic analysis framework by instrumenting JS code. Rozzle [73] is a virtual machine that performs multi-path execution experiments in parallel to enhance the efficiency of dynamic analysis. J-Force [63] uncovers hidden malicious behaviors by forcibly exploring all possible execution paths. Dual-Force [122] is a technique that forcibly executes both Java and JavaScript code of WebView applications simultaneously to reveal hidden payloads of malware. JSGraph [123] records fine-grained details about how JS programs are executed and how their effects are reflected in DOM elements within a browser. JStap [120] is a static malicious JavaScript detector that enhances the detection capability of existing lexical and AST-based pipelines.

REFERENCES

[1] eMarketer, *Global ecommerce forecast 2022*, https://www.insiderintelligence.com/content/global-ecommerce-forecast-2022, 2022.

[2] IAB, *Iab internet advertising revenue report full-year 2021 results*, https://www.iab.com/insights/internet-advertising-revenue-report-full-year-2021/, 2022.

[3] Statista, *Video streaming (svod) - us, statista market forecast*, https://www.statista.com/outlook/dmo/digital-media/video-on-demand/video-streaming-svod/united-states, 2023.

[4] eMarketer, *Worldwide digital ad revenues*, https://www.insiderintelligence.com/forecasts/5d13a07a64fe7d034c2cc15a/5d139fb0b88aeb0b7c481d6c/, 2023.

[5] Oberlo, *Us digital ad spending (20172027)*, https://www.oberlo.com/statistics/us-digital-ad-spending, 2023.

[6] B. of Apps Research, *Ad fraud statistics*, https://www.businessofapps.com/ads/ad-fraud/research/ad-fraud-statistics/, 2023.

[7] F. Tech, *Retargeting vs. remarketing*, https://www.floridatechonline.com/blog/business/retargeting-vs-remarketing/, 2017.

[8] A. R. Bv, *9 remarketing/retargeting services which drive your online sales*, https://www.adwordsrobot.com/en/blog/9-remarketing-retargeting-services-which-drive-your-online-sales, 2017.

[9] P. LaFond, *How retailer shopstyle gained a 200% increase in retargeting conversions with trusignal and mediamath*, http://www.mediamath.com/blog/how-retailer-shopstyle-gained-a-200-increase-in-retargeting-conversions-with-trusignal-and-mediamath/, 2016.

[10] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[11] M. Pawlik and N. Augsten, "Tree edit distance: Robust and memory-efficient," *Information Systems*, vol. 56, pp. 157–173, 2016.

[12] P. Willett, "Recent trends in hierarchic document clustering: A critical review," *Information Processing & Management*, vol. 24, no. 5, pp. 577–597, 1988.

[13] *Criteo dynamic retargeting*, http://www.criteo.com/, 2017.

[14] *Amazon elastic compute cloud (amazon ec2)*, https://aws.amazon.com/ec2/, 2017.

[15] *Google cloud platform*, https://cloud.google.com/, 2017.

[16] *Azure*, https://azure.microsoft.com, 2017.

[17] *Selenium browser automation*, http://www.seleniumhq.org/, 2017.

[18] *Technical report - adbudgetkiller: Online advertising budget draining attack*, https://sites.google.com/budgetkiller, 2017.

[19] *Alexa top 500 - shopping category*, http://www.alexa.com/topsites/category/Top/Shopping, 2017.

[20] Salesforce, *Marketing cloud advertising index q1 2016 report*, https://www.marketingcloud.com/sites/exacttarget/files/deliverables/salesforce-advertising-index-q1-2016-advertising-studio.pdf, 2016.

[21] Salesforce, *Marketing cloud advertising index q2 2016 report*, https://www.marketingcloud.com/sites/exacttarget/files/salesforce-advertising-index-q2-2016-advertising-studio-v8.pdf, 2016.

[22] Adroll, *Event segments*, https://help.adroll.com/hc/en-us/articles/212014288-Event-Segments, 2017.

[23] *Perfect audience - event audiences*, http://support.perfectaudience.com/knowledgebase/articles/233996-understanding-action-lists, 2017.

[24] X. Xing, W. Meng, D. Doozan, A. C. Snoeren, N. Feamster, and W. Lee, "Take this personally: Pollution attacks on personalized services.," in *USENIX Security*, 2013, pp. 671–686.

[25] W. Meng, X. Xing, A. Sheth, U. Weinsberg, and W. Lee, "Your online interests: Pwned! a pollution attack against targeted advertising," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 129–140.

[26] N. Daswani, C. Mysen, V. Rao, S. Weis, K. Gharachorloo, and S. Ghosemajumder, "Online advertising fraud," *Crimeware: understanding new attacks and defenses*, vol. 40, no. 2, pp. 1–28, 2008.

[27] B. Stone-Gross, R. Stevens, A. Zarras, R. Kemmerer, C. Kruegel, and G. Vigna, "Understanding fraudulent activities in online ad exchanges," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, ACM, 2011, pp. 279–294.

[28] V. Dave, S. Guha, and Y. Zhang, "Measuring and fingerprinting click-spam in ad networks," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 175–186, 2012.

[29] M. Faou *et al.*, "Follow the traffic: Stopping click fraud by disrupting the value chain," in *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*, IEEE, 2016, pp. 464–476.

[30] M. S. Iqbal, M. Zulkernine, F. Jaafar, and Y. Gu, "Protecting internet users from becoming victimized attackers of click-fraud," *Journal of Software: Evolution and Process*, 2017.

[31] K. Springborn and P. Barford, "Impression fraud in on-line advertising via pay-per-view networks.," in *USENIX Security*, 2013, pp. 211–226.

[32] M. Marciel *et al.*, "Understanding the detection of view fraud in video content portals," in *Proceedings of the 25th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2016, pp. 357–368.

[33] J. Crussell, R. Stevens, and H. Chen, "Madfraud: Investigating ad fraud in android applications," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ACM, 2014, pp. 123–134.

[34] B. Liu, S. Nath, R. Govindan, and J. Liu, "Decaf: Detecting and characterizing ad fraud in mobile apps.," in *NSDI*, 2014, pp. 57–70.

[35] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, 2012, pp. 12–12.

[36] S. Englehardt *et al.*, "Cookies that give you away: The surveillance implications of web tracking," in *Proceedings of the 24th International Conference on World Wide Web*, ACM, 2015, pp. 289–299.

[37] M. Conti, V. Cozza, M. Petrocchi, and A. Spognardi, "Trap: Using targeted ads to unveil google personal profiles," in *Information Forensics and Security (WIFS), 2015 IEEE International Workshop on*, IEEE, 2015, pp. 1–6.

[38] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson, "Tracing information flows between ad exchanges using retargeted ads," in *Proceedings of the 25th USENIX Security Symposium*, 2016.

[39] A. Cahn, S. Alfeld, P. Barford, and S. Muthukrishnan, "What's in the community cookie jar?" In *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*, IEEE, 2016, pp. 567–570.

[40] A. Boodman, *Greasemonkey project homepage*, https://www.greasespot.net/, 2019.

[41] J. Biniok, *Tampermonkey project homepage*, https://tampermonkey.net/, 2019.

[42] *Owasp ajax security cheat sheet*, https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/AJAX_Security_Cheat_Sheet.md#dont-rely-on-client-business-logic, 2019.

[43] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to shop for free online–security analysis of cashier-as-a-service based web stores," in *Security and Privacy (SP), 2011 IEEE Symposium on*, IEEE, 2011, pp. 465–480.

[44] T. N. Y. Times, *Breaking News, World News & Multimedia*, https://www.nytimes.com/, 2019.

[45] P. Preston, *A paywall that pays? Only in America*, https://www.theguardian.com/media/2011/aug/07/paywall-that-pays-only-in-america, 2011.

[46] *Business flow tampering success cases*, https://sites.google.com/view/tampering-cases.

[47] J. Alexander, *YouTube Premium is changing because it has to*, https://www.theverge.com/2018/11/29/18116154/youtube-premium-free-ads-subscription-red, 2018.

[48] G. Sloane, *YouTube is now showing ad-supported Hollywood movies*, https://adage.com/article/digital/youtube-starts-showing-free-hollywood-movies-ad-breaks/315631/, 2018.

[49] J. Valinsky, *Some Adblock Plus users are reporting problems with YouTube*, https://digiday.com/social/youtube-adblock-problems/, 2016.

[50] *Security token*, https://en.wikipedia.org/wiki/Security_token.

[51] Y. Saeys, I. Inza, and P. Larrañaga, "A review of feature selection techniques in bioinformatics," *bioinformatics*, vol. 23, no. 19, pp. 2507–2517, 2007.

[52] C. Chen, A. Liaw, and L. Breiman, "Using random forest to learn imbalanced data," *Tech. Rep*, pp. 1–12, 2004.

[53] R. Akbani, S. Kwek, and N. Japkowicz, "Applying support vector machines to imbalanced datasets," in *European conference on machine learning*, Springer, 2004, pp. 39–50.

[54] Z. Zheng, X. Wu, and R. Srihari, "Feature selection for text categorization on imbalanced data," *ACM SIGKDD Explorations Newsletter*, vol. 6, no. 1, pp. 80–89, 2004.

[55] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[56] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *Kdd*, vol. 96, 1996, pp. 226–231.

[57] *Puppeteer*, https://pptr.dev/.

[58] *Chrome devtools protocol*, https://chromedevtools.github.io/devtools-protocol/.

[59] *V8 javascript engine*, https://v8.dev/.

[60] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *ACM SIGPLAN Notices*, ACM, vol. 42, 2007, pp. 237–249.

[61] *W3c recommendation - subresource integrity*, https://www.w3.org/TR/SRI/, 2016.

[62] *Inboxdollars - about us*, http://corporate.inboxdollars.com/about-us/company/.

[63] K. Kim *et al.*, "J-force: Forced execution on javascript," in *Proceedings of the 26th international conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2017, pp. 897–906.

[64] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, K. Fu and J. Jung, Eds., USENIX Association, 2014, pp. 829–844. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng.

[65] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "Iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, pp. 44–56.

[66] R. Johnson and A. Stavrou, "Forced-path execution for android applications on x86 platforms," in *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*, IEEE, 2013, pp. 188–197.

[67] J. Wilhelm and T. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *Recent Advances in Intrusion Detection, 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7, 2007, Proceedings*, C. Krügel, R. Lippmann, and A. J. Clark, Eds., ser. Lecture Notes in Computer Science, vol. 4637, Springer, 2007, pp. 219–235. DOI: 10.1007/978-3-540-74320-0_12. [Online]. Available: https://doi.org/10.1007/978-3-540-74320-0%5C_12.

[68] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, IEEE, 2013, pp. 74–83.

[69] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for javascript web applications," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 429–444, 2015.

[70] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Security and Privacy (SP), 2010 IEEE Symposium on*, IEEE, 2010, pp. 513–528.

[71] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ACM, 2013, pp. 488–498.

[72] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015, pp. 842–853.

[73] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 443–457.

[74] F. Sun, L. Xu, and Z. Su, "Detecting logic vulnerabilities in e-commerce applications.," in *NDSS*, 2014.

[75] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *Security and Privacy (SP), 2012 IEEE Symposium on*, IEEE, 2012, pp. 365–379.

[76] A. Sudhodanan, A. Armando, R. Carbone, L. Compagna, *et al.*, "Attack patterns for black-box security testing of multi-party web applications.," in *NDSS*, 2016.

[77] E. Y. Chen, S. Chen, S. Qadeer, and R. Wang, "Securing multiparty online services via certification of symbolic transactions," in *Security and Privacy (SP), 2015 IEEE Symposium on*, IEEE, 2015, pp. 833–849.

[78] D. Fett, R. Küsters, and G. Schmitz, "An expressive model for the web infrastructure: Definition and application to the browser id sso system," in *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, 2014, pp. 673–688.

[79] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the ear: Discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, 2011, pp. 251–262.

[80] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications.," in *NDSS*, 2014.

[81] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Building call graphs for embedded client-side code in dynamic web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 518–529.

[82] C. Q. Adamsen, A. Møller, S. Alimadadi, and F. Tip, "Practical ajax race detection for javascript web applications," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2018, pp. 38–48.

[83] J. Ye, C. Zhang, L. Ma, H. Yu, and J. Zhao, "Efficient and precise dynamic slicing for client-side javascript programs," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 449–459.

[84] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: Finding and understanding conflicts between javascript libraries," in *Proceedings of the 40th International Conference on Software Engineering*, ACM, 2018, pp. 741–751.

[85] insightSLICE, *Digital content creation market - global market share, trends, analysis and forecasts, 2020 - 2030*, https://www.insightslice.com/digital-content-creation-market, Nov. 2020.

[86] I. L. Kim *et al.*, "Finding client-side business flow tampering vulnerabilities," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 222–233, ISBN: 9781450371216. DOI: 10.1145/3377811.3380355. [Online]. Available: https://doi.org/10.1145/3377811.3380355.

[87] N. Newman, *Journalism, media and technology trends and predictions 2019*, https://www.digitalnewsreport.org/publications/2019/journalism-media-technology-trends-predictions-2019/, Jan. 2019.

[88] *Bypass paywalls*, https://github.com/iamadamdev/bypass-paywalls-chrome, 2022.

[89] D. Coldewey, *Thousands of major sites are taking silent anti-ad-blocking measures*, https://techcrunch.com/2017/12/27/thousands-of-major-sites-are-taking-silent-anti-ad-blocking-measures/, Dec. 2017.

[90] *Los angeles times*, https://www.latimes.com/, 2022.

[91] *Student share*, https://studentshare.org/, 2022.

[92] *Google's funding choices*, https://fundingchoices.google.com/, 2022.

[93] *Chrome devtools recorder: Record, replay and measure user flows*, https://developer.chrome.com/-docs/devtools/recorder/, 2021.

[94] *V8 javascript engine*, https://v8.dev/, 2022.

[95] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE mobile computing and communications review*, vol. 5, no. 1, pp. 3–55, 2001.

[96] *Bftdetector git repository*, https://github.com/jspaper22/bftdetector, 2022.

[97] *Time*, https://time.com/, 2022.

[98] *Fortune*, https://fortune.com/, 2022.

[99] *Automotive news*, https://www.autonews.com/, 2022.

[100] *Forbes*, https://www.forbes.com/, 2022.

[101] *Bookmate*, https://bookmate.com/, 2022.

[102] *The times of northwest indiana*, https://www.nwitimes.com/, 2022.

[103] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. 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*, 2016, pp. 205–216.

[104] *Javascript obfuscator tool*, https://obfuscator.io/, 2022.

[105] T. Kachalov, *Javascript obfuscator - controlflowflattening*, https://github.com/javascript-obfuscator/javascript-obfuscator#controlflowflattening.

[106] T. Kachalov, *Javascript obfuscator - dead code injection*, https://github.com/javascript-obfuscator/javascript-obfuscator#deadcodeinjection, 2022.

[107] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, 2011, pp. 263–274.

[108] N. Skrupsky, P. Bisht, T. Hinrichs, V. Venkatakrishnan, and L. Zuck, "Tamperproof: A server-agnostic defense for parameter tampering attacks on web applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, ACM, 2013, pp. 129–140.

[109] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner.," in *USENIX Security Symposium*, vol. 14, 2012.

[110] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE Symposium on Security and Privacy*, IEEE, 2010, pp. 332–345.

[111] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan, "Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, ACM, 2010, pp. 607–618.

[112] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Detection of Intrusions and Malware, and Vulnerability Assessment, 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings*, C. Kreibich and M. Jahnke, Eds., ser. Lecture Notes in Computer Science, vol. 6201, Springer, 2010, pp. 111–131. DOI: 10.1007/978-3-642-14215-4_7. [Online]. Available: https://doi.org/10.1007/978-3-642-14215-4%5C_7.

[113] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, ACM, 2011, pp. 571–580.

[114] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[115] G. Li, E. Andreasen, and I. Ghosh, "Symjs: Automatic symbolic testing of javascript web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 449–459.

[116] N. Nikiforakis *et al.*, "You are what you include: Large-scale evaluation of remote javascript inclusions," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 736–747, ISBN: 9781450316514. DOI: 10.1145/2382196.2382274. [Online]. Available: https://doi.org/10.1145/2382196.2382274.

[117] M. Zhang and W. Meng, "Detecting and understanding javascript global identifier conflicts on the web," ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 38–49, ISBN: 9781450370431. DOI: 10.1145/3368089.3409747. [Online]. Available: https://doi.org/10.1145/3368089.3409747.

[118] Z. Chen and Y. Cao, "Jskernel: Fortifying javascript against web concurrency attacks via a kernel-like structure," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, IEEE, 2020, pp. 64–75. DOI: 10.1109/DSN48063.2020.00026. [Online]. Available: https://doi.org/10.1109/DSN48063.2020.00026.

[119] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual browser: A virtualized browser to sandbox third-party javascripts with enhanced security," in *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, H. Y. Youm and Y. Won, Eds., ACM, 2012, pp. 8–9. DOI: 10.1145/2414456.2414460. [Online]. Available: https://doi.org/10.1145/2414456.2414460.

[120] A. Fass, M. Backes, and B. Stock, "Jstap: A static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, D. Balenson, Ed., ACM, 2019, pp. 257–269. DOI: 10.1145/3359789.3359813. [Online]. Available: https://doi.org/10.1145/3359789.3359813.

[121] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "Jast: Fully syntactic detection of malicious (obfuscated) javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, C. Giuffrida, S. Bardin, and G. Blanc, Eds., ser. Lecture Notes in Computer Science, vol. 10885, Springer, 2018, pp. 303–325. DOI: 10.1007/978-3-319-93411-2_14. [Online]. Available: https://doi.org/10.1007/978-3-319-93411-2%5C_14.

[122] Z. Tang *et al.*, "Dual-force: Understanding webview malware via cross-language forced execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 714–725.

[123] B. Li *et al.*, "Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions," in *NDSS*, 2018.

[124] J. Wang, W. Dou, C. Gao, Y. Gao, and J. Wei, "Context-based event trace reduction in client-side javascript applications," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2018, pp. 127–138.

[125] F. S. Ocariza Jr, G. Li, K. Pattabiraman, and A. Mesbah, "Automatic fault localization for client-side javascript," *Software Testing, Verification and Reliability*, vol. 26, no. 1, pp. 69–88, 2016.

[126] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 144–156.