BFTDETECTOR: Automatic Detection of Business Flow Tampering for Digital Content Service

I Luk Kim
Department of Computer Science
Purdue University
West Lafayette, USA
kim1634@purdue.edu

Weihang Wang
Department of Computer Science
University of Southern California
Los Angeles, USA
weihangw@usc.edu

Yonghwi Kwon
Department of Computer Science
University of Virginia
Charlottesville, USA
yongkwon@virginia.edu

Xiangyu Zhang
Department of Computer Science
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

***Abstract*—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.

***Index Terms*—JavaScript, business flow tampering, dynamic analysis, vulnerability detection**

I. 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 [29].

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).

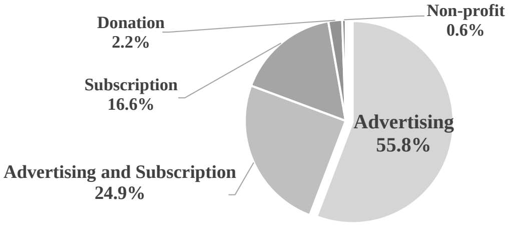


Fig. 1: Business Models of 178 Digital Content Service Providers in Alexa Top 500.

Figure 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 [32] 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 [37] indicates that *BFT has become a real-world threat*: software or browser extensions aim to circumvent paywalls (e.g. [5]) 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 [22], 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:

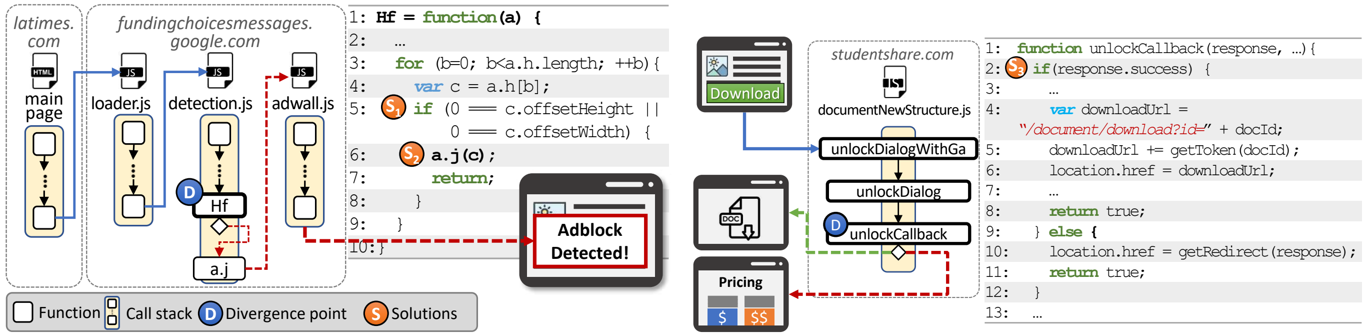
- We propose a novel system, BFTDETECTOR, to find BFT flaws. It automatically exercises business process on a content provider's website to identify the execution points that can be tampered with.
- We generalize the business models and relate the models with website implementations, using the models 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.

II. Motivation

We use two real-world examples, Los Angeles Times (LA Times) [10] and StudentShare [12], 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 [8]. When Adblockers are detected, the user is prompted with a message that directs the user to a subscription page, asking for payments.

Figure 2a shows the anti-adblocker process. When the main page of the website is loaded, it injects 'loader.js' from



```
1: Hf = function(a) {
2:     ...
3:     for (b=0; b<a.h.length; ++b){
4:         var c = a.h[b];
5:         if (0 === c.offsetHeight ||
               0 === c.offsetWidth) {
6:             a.j(c);
7:             return;
8:         }
9:     }
10:}
```

(a) LA Times

```
1:  function unlockCallback(response, ...){
2:    if(response.success) {
3:        ...
4:        var downloadUrl =
     "/document/download?id=" + docId;
5:        downloadUrl += getToken(docId);
6:        location.href = downloadUrl;
7:        ...
8:        return true;
9:    } else {
10:       location.href = getRedirect(response);
11:       return true;
12:   }
13:  ...
```

(b) StudentShare

Fig. 2: Motivating Examples.

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 to access the premium essay samples. Figure 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).

A. Business Flow Tampering Flaws

The two websites have BFT flaws. First, in LA Times (Figure 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 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.

B. 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 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).

C. 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 III-A1). 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 2a and unlockCallback() in Figure 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 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 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.

III. System Design

Overview. Figure 3 shows a brief overview of our BFT detection system, which consists of five phases:

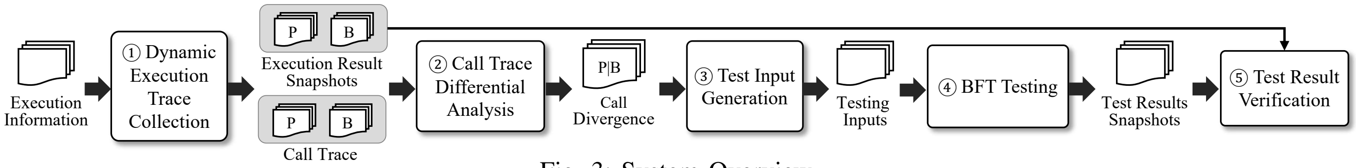


Fig. 3: System Overview

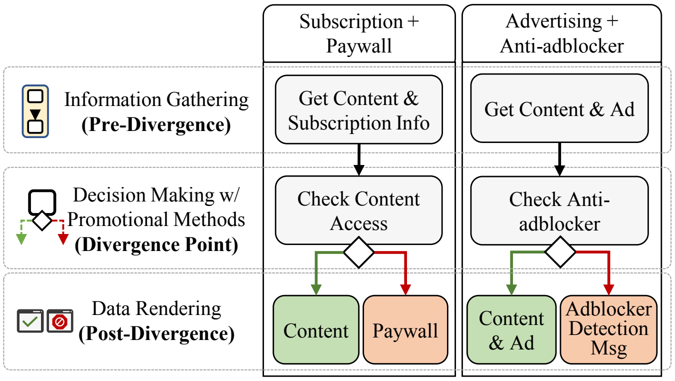


Fig. 4: Generalized Business Process.

TABLE I: 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
$TriggerPaywall(P\|JS, S)$	1. Open a browser with a session S 2. Visit all pages $\in P$ or replay JS 3. Return the session S
$CollectTrace(P\|JS, S)$	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.

TABLE II: 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.

① Dynamic Execution Trace Collection (Section III-A). 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.

② Call Trace Differential Analysis (Section III-B). 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 III-C). We generate test inputs containing statements data to be mutated by using the call divergence points from the previous step.

④ BFT Testing (Section III-D). 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 III-E). 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.

A. Dynamic Execution Trace Collection

1) 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 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).

2) 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**.

3) 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 [11] script recorded by a tester providing automated browsing.

Business Process Procedures. Table I shows three business procedures that serve as building blocks for exercising the flows in the business model. Table II 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 [1]. This enables our system to emulate website-specific browsing procedures (e.g., logging in or clicking the download button in the StudentShare case (Figure 2b)).

4) 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 III-B).

Bytecode Level Instrumentation. Instrumenting complex and often obfuscated real-world programs is challenging. Hence, we modify the V8 JS engine [15] 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 IV-D, 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 III-E.

Algorithm 1 Call Divergence Point Discovery

Input: P, B : lists of call traces collected from passing and blocking runs, where $P_i \in P$ or $B_i \in B$ is a list of call signature c, and $c_i \in c$ denotes a set (Caller, Call Statement, Callstack)

Output: CD : a list of call divergence data, where $CD_i \in CD$ 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 $CD$
```

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.

B. 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 2a (LA Times), `Hf()` is the call divergence point since the execution flows of passing and blocking runs reach the function, but only blocking flow continues to `a.j()`. Similarly, `unlockCallback()` in Figure 2b is the call divergence point.

Algorithm. Algorithm 1 describes how we identify the call divergence point. It takes two lists of call traces (P and B) that are collected in Section III-A4. 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 III-A4.

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.

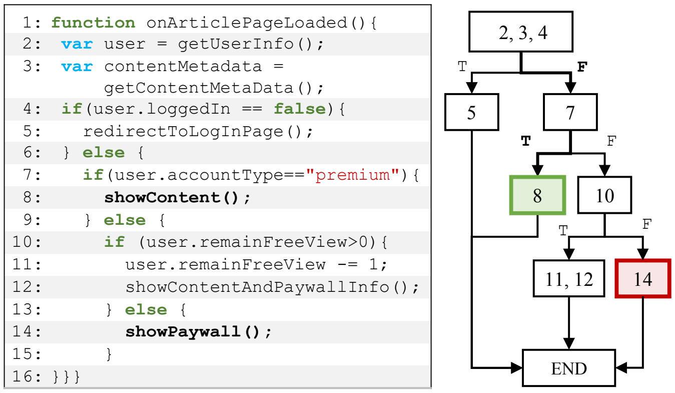
Next, it identifies call divergence points by leveraging the intersection (P_{int} and B_{int}; pre-divergence) and unique (P_{uniq} and B_{uniq}; post-divergence) call traces. In particular, we detect a divergence point if a function is (1) a callee of a common signature available in both runs and also 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 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.

C. 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 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.



```
 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: }}}
```

Fig. 5: Illustrative Example for Test Input Generation

We illustrate the test input generation process by using a simplified website with a softer promotional method as shown in Figure 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`).

D. 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 II. 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.

E. 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 III-A4), 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 [42] 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 [48]. 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 [32]. We train the classifier with a total of 1,778 snapshots collected from 13 websites using the subscription and advertising business models.

IV. Evaluation

Implementation. BFTDETECTOR [3] 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?

A. BFT Detection Results

Table III 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 III: 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 [13], Fortune [7], Automotive News [2], Forbes [6], and Bookmate [4]. 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].

TABLE IV: 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

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 IV. 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 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.

B. 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 III 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 III-B. The sixth column represents the number 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).

TABLE V: 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%)

C. 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 III-E, 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 V 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 [14] 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.

D. Performance Overhead

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

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

TABLE VI: 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

TABLE VII: 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

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 III 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 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 VI 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 III-A4.

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

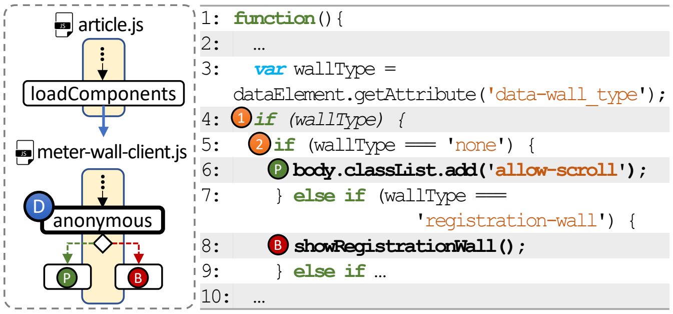
E. Comparison Study

We compare our technique with state-of-the-art technique JSFlowTamper [32] 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 VII shows the BFT detection result. As shown in Table VIIa, 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 VIIb. The first one was caused when there was no DOM mutation events related to business process as in the StudentShare example (Figure 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 [47]. 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.

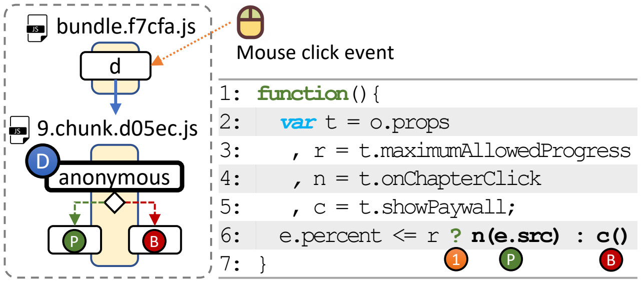
F. Case Study



```
function(){
  …
    var wallType =
dataElement.getAttribute('data-wall_type');
if (wallType) {
  if (wallType === 'none') {
    body.classList.add('allow-scroll');
    } else if (wallType ===
                  'registration-wall') {
    showRegistrationWall();
    } else if …
  …
```

Fig. 6: Business Process of Time.com

1) TIME.com: TIME [13] 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 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 (Ⓟ) or to show a registration message (Ⓑ). Our system successfully identified the divergence point (Ⓓ), and found the test input that changes the blocking flow by forcibly taking the `then` branches of the two `if` statements (① and ②).



```
1: function(){
2:    var t = o.props
3:     , r = t.maximumAllowedProgress
4:     , n = t.onChapterClick
5:     , c = t.showPaywall;
6:   e.percent <= r ? n(e.src) : c()
7: }
```

Fig. 7: Business Process of Bookmate.com

2) Bookmate.com: Bookmate [4] 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 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 (Ⓟ); otherwise, the paywall is displayed (Ⓑ). The test input our system found forcibly executes the `true` branch of the conditional expression (①).

V. Mitigation: Server side Code Randomization

As we discussed in Section II-B, 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 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 [9], 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 [30], and the dead code injection increases the code size up to 200% [31]. 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 VIII, our mitigation approach increase only 287 bytes and 8.15 ms, which we believe reasonable.

TABLE VIII: 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.

VI. 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 IV), 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 IV-A, 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).

VII. 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 [17], [18], [20], [23], [24], [40], [43], [44]. 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 [20], testing for checkout system flaws [40], feedback-directed automated test generation [16]. 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 [32] 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 2b), (3) detecting flaws related to multiple JS files in the call chain (e.g., Figure 2a).

JS Analysis Techniques. There are a variety of techniques analyzing JS code [16], [19], [21], [25]–[28], [33]–[36], [38], [39], [41], [45], [46], [49]. Jalangi [41] provides a dynamic analysis framework by instrumenting JS code. Rozzle [34] is a virtual machine that performs multi-path execution experiments in parallel to enhance the efficiency of dynamic analysis. J-Force [33] uncovers hidden malicious behaviors by forcibly exploring all possible execution paths. Dual-Force [45] is a technique that forcibly executes both Java and JavaScript code of WebView applications simultaneously to reveal hidden payloads of malware. JSGraph [35] records fine-grained details about how JS programs are executed and how their effects are reflected in DOM elements within a browser. JStap [25] is a static malicious JavaScript detector that enhances the detection capability of existing lexical and AST-based pipelines.

VIII. Conclusion

We present an automated approach to detect the BFT flaws on digital content service websites. Our novel business model based approach automatically exercises different business flows and identifies the flaws via our differential analysis algorithm. Our evaluation result shows that our approach is highly effective, discovering 315 flaws from 204 high-profile websites such as TIME, Fortune, and Forbes.

Acknowledgments

We thank the anonymous referees for their constructive feedback. The authors gratefully acknowledge the support of NSF 1908021, 1916499, 2047980, and 2145616. This research was also partially supported by a gift from Cisco Systems. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

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

[2] "Automotive news," https://www.autonews.com/, 2022.

[3] "Bftdetector git repository," https://github.com/jspaper22/bftdetector, 2022.

[4] "Bookmate," https://bookmate.com/, 2022.

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

[6] "Forbes," https://www.forbes.com/, 2022.

[7] "Fortune," https://fortune.com/, 2022.

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

[9] "Javascript obfuscator tool," https://obfuscator.io/, 2022.

[10] "Los angeles times," https://www.latimes.com/, 2022.

[11] "Puppeteer," https://developers.google.com/web/tools/puppeteer, 2022.

[12] "Student share," https://studentshare.org/, 2022.

[13] "Time," https://time.com/, 2022.

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

[15] "V8 javascript engine," https://v8.dev/, 2022.

[16] 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.

[17] 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.

[18] 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.

[19] 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. [Online]. Available: https://doi.org/10.1145/2414456.2414460

[20] 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.

[21] 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. [Online]. Available: https://doi.org/10.1109/DSN48063.2020.00026

[22] 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/, December 2017.

[23] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, “Enemy of the state: A state-aware black-box web vulnerability scanner,” in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed. USENIX Association, 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[24] 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*, ser. Lecture Notes in Computer Science, C. Kreibich and M. Jahnke, Eds., vol. 6201. Springer, 2010, pp. 111–131. [Online]. Available: https://doi.org/10.1007/978-3-642-14215-4_7

[25] 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. [Online]. Available: https://doi.org/10.1145/3359789.3359813

[26] 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*, ser. Lecture Notes in Computer Science, C. Giuffrida, S. Bardin, and G. Blanc, Eds., vol. 10885. Springer, 2018, pp. 303–325. [Online]. Available: https://doi.org/10.1007/978-3-319-93411-2_14

[27] 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.

[28] 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.

[29] insightSLICE, “Digital content creation market - global market share, trends, analysis and forecasts, 2020 - 2030,” https://www.insightslice.com/digital-content-creation-market, November 2020.

[30] T. Kachalov, “Javascript obfuscator - controlflowflattening,” https://github.com/javascript-obfuscator/javascript-obfuscator#controlflowflattening, 2022.

[31] ——, “Javascript obfuscator - dead code injection,” https://github.com/javascript-obfuscator/javascript-obfuscator#deadcodeinjection, 2022.

[32] I. L. Kim, Y. Zheng, H. Park, W. Wang, W. You, Y. Aafer, and X. Zhang, “Finding client-side business flow tampering vulnerabilities,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 222–233. [Online]. Available: https://doi.org/10.1145/3377811.3380355

[33] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, “J-force: Forced execution on javascript,” in *Proceedings of the 26th international conference on World Wide Web*, 2017, pp. 897–906.

[34] 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.

[35] B. Li, P. Vadrevu, K. H. Lee, R. Perdisci, J. Liu, B. Rahbarinia, K. Li, and M. Antonakakis, “Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions,” in *NDSS*, 2018.

[36] 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.

[37] N. Newman, “Journalism, media and technology trends and predictions 2019,” https://www.digitalnewsreport.org/publications/2019/journalism-media-technology-trends-predictions-2019/, January 2019.

[38] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “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. New York, NY, USA: Association for Computing Machinery, 2012, p. 736–747. [Online]. Available: https://doi.org/10.1145/2382196.2382274

[39] 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.

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

[41] 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*, 2013, pp. 488–498.

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

[43] 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.

[44] 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.

[45] Z. Tang, J. Zhai, M. Pan, Y. Aafer, S. Ma, X. Zhang, and J. Zhao, “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.

[46] 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.

[47] 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.

[48] 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.

[49] M. Zhang and W. Meng, “Detecting and understanding javascript global identifier conflicts on the web,” ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 38–49. [Online]. Available: https://doi.org/10.1145/3368089.3409747