Finding Client-side Business Flow Tampering Vulnerabilities

Reading report prepared by xxx

Citation

Kim, I. Luk, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. "Finding client-side business flow tampering vulnerabilities." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 222-233. 2020.

Abstract

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 thirdparty 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.

Key Targeted Problem

Web applications must expose a portion of the logic to the client-side to coordinate multiple parties, which can be tampered with on the fly. Therefore, vulnerabilities in the exposed logic can be used, leading to business logic perturbations and financial loss. This paper targets the problem of identifying the vulnerabilities in the exposed logic of web applications.

Inputs

The domain link to the web application to be tested.

Outputs

Reports of vulnerabilities.

Why the paper is accepted?

Usefulness:

The technique introduced in this article can help web application developers to automatically discover vulnerabilities in the common logic of their web applications without having to enter anything other than the domain link to the web application. This technology also generates tampering proposals for the execution of the attack process, thereby helping developers understand how the attackers can use discovered vulnerabilities to attack their web applications.

Novelty/impact:

No practical tools can identify vulnerabilities in the exposed logic of web applications automatically before. They found that the vulnerability problem is quite common and can be quite serious to hurt the business models of web applications.

Elegance:

The paper is well written and presented.

Summary

This paper proposes a technique to scan the client-side of the web application, discovered vulnerabilities, and generate tampering proposals to confirm real vulnerabilities automatically.

To motivate their work, this paper lists 2 real-world examples：

1. For news publishers such as New York Times, user subscription is their main business model. However, the vulnerabilities in their web application that allow users to bypass their metered paywalls are making them lose their main business income.
2. For streaming service providers such as YouTube, ads revenue is their main business model. However, the vulnerabilities in their web application that allow users to skip watching ads are making them lose part of their main business income.

Meanwhile, the client-side code is typically provided by multiple parties. The overwhelming size and complexity of the client-side code make it impractical for developers to manually find vulnerable points.

To resolve such questions, this paper proposes a dynamic analysis approach to detect vulnerabilities in web applications. Figure 2 shows the design of their approach.

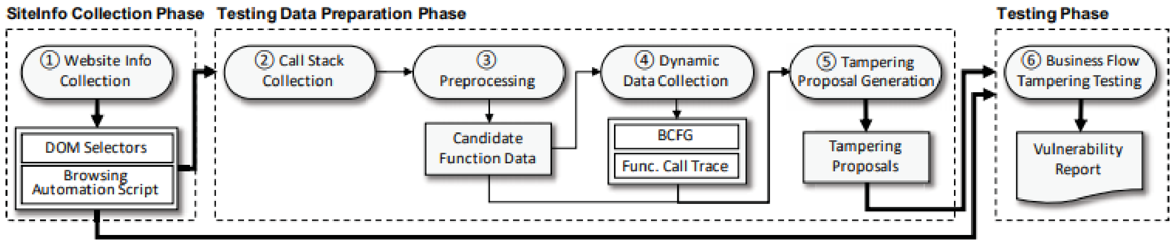
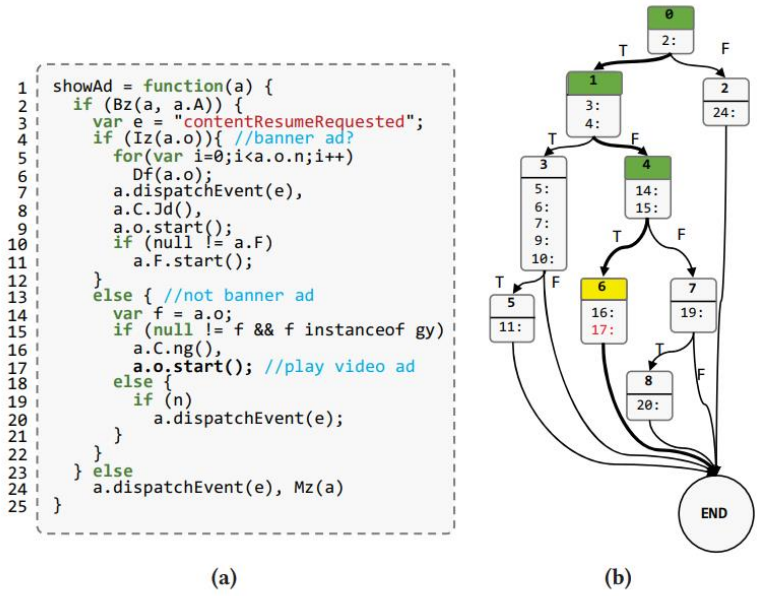

SiteInfo Collection Phase
Testing Data Preparation Phase
Testing Phase
① Website Info Collection
DOM Selectors
Browsing Automation Script
② Call Stack Collection
③ Preprocessing
Candidate Function Data
④ Dynamic Data Collection
BCFG
Func. Call Trace
⑤ Tampering Proposal Generation
Tampering Proposals
⑥ Business Flow Tampering Testing
Vulnerability Report

Figure 2: Approach overview

The system will first collect the target website information, including the DOM objects and automation scripts. Then it will intercept the mutation events on the DOM objects to collect the corresponding call stack. It will then construct a call tree to identify potential business logic. In the call tree, all nodes are considered candidates and given high priorities. Then, the system collects runtime information on these candidate functions. It will dynamically construct a Business Control Flow Graph (BCFG) for each candidate. It will also collect DOM mutation types, function execution frequencies, positions in source code, and the source URL information.

The BCFG is a kind of control flow graph that stores the business control flow in the candidates. Figure 3 shows how the system will convert the business control flow of the source code of the candidate showAd to its BCFG.



```
 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  }
```

Figure 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**

After the above steps, the system has collected all the necessary information. It is time to generate tampering proposals for testing. The system will first rank functions using a learning-based method. Then the BCFGs of ranked functions are traversed to derive tampering proposals.

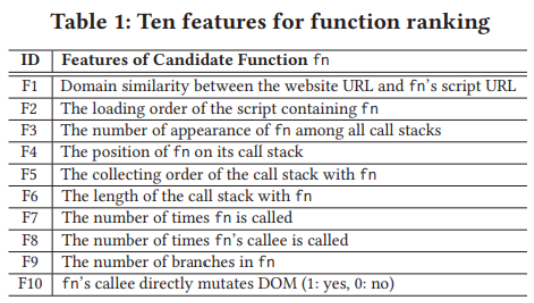
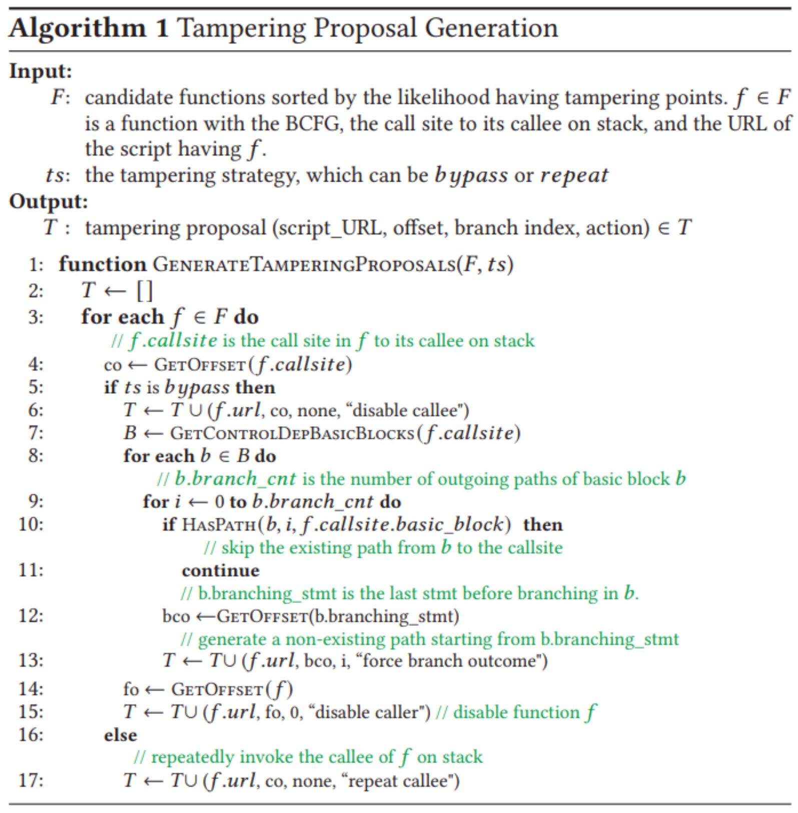


Table 1: Ten features for function ranking

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)

The system will evaluate the importance of 10 features presented in Table 1 using tampering locations that are learned from a small number of web applications and using ANOVA F-test to refine these features. Then, it will choose highly important features for the classification task. The classification task uses a classifier that is learned from the training data using the Balanced Random Forest, the weighted-SVM, and the weighted-Logistic Regression. After classification, the tampering proposals are generated by Algorithm 1 shown in the following figure:



Algorithm 1 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) $\in T$

```
function GenerateTamperingProposals($F, ts$)
  $T \leftarrow []$
  for each $f \in F$ do
     // $f.callsite$ is the call site in $f$ to its callee on stack
    co ← GetOffset($f.callsite$)
    if $ts$ is $bypass$ then
      $T \leftarrow T \cup (f.url$, co, none, "disable callee")
      $B \leftarrow$ GetControlDepBasicBlocks($f.callsite$)
      for each $b \in B$ do
          // $b.branch\_cnt$ is the number of outgoing paths of basic block $b$
        for $i \leftarrow 0$ to $b.branch\_cnt$ do
          if HasPath($b, i, f.callsite.basic\_block$) then
              // skip the existing path from $b$ to the callsite
            continue
            // b.branching_stmt is the last stmt before branching in $b$.
          bco ←GetOffset(b.branching_stmt)
            // generate a non-existing path starting from b.branching_stmt
          $T \leftarrow T \cup (f.url$, bco, i, "force branch outcome")
      fo ← GetOffset($f$)
      $T \leftarrow T \cup (f.url$, fo, 0, "disable caller") // disable function $f$
    else
        // repeatedly invoke the callee of $f$ on stack
      $T \leftarrow T \cup (f.url$, co, none, "repeat callee")
```

The algorithm requires ranked candidate functions and the tampering strategy, which is *bypass* or *repeat*, as inputs, and output a list of tampering proposals. A tampering proposal consists of the URL of the script containing the candidate function, the source code offset of the tampering point, the branch index, and the tampering action. For each candidate function f, the algorithm first locates the locations where f invokes its callees observed on the stack (line 4). When the tampering strategy is *bypass*, the algorithm generates a proposal that skips the invocation of the callee function (line 6) and obtains the basic blocks that the call site control-depends on (line 7). Then, the algorithm checks whether the calling site can be reached via a specific path. Among them, the algorithm skips the path connecting the predecessor block and the call site (line 11). Since the algorithm explores the remaining paths even when the path conditions are not met, the algorithm generates tampering proposals for such paths (line 13) so that the branch outcome can be enforced.

After generated the tampering proposals, the system will use the automated scripts to load the target website. When the modified JS engine obtains the script, the modification proposal specifies that the bytecode IR is changed dynamically according to 4 kinds of tampering actions in the proposal: disable callee, disable caller, forced branching, and repeat callee. After that, the system needs human testers to check if the tampering proposal successfully alters the original business in an intended way. To minimize manual work, the method uses similarity-based clustering techniques to group the collected test results. For each DOM tree, the method uses the Structural Similarity Index Method on its screenshot and the Tree Edit Distance algorithm on its HTML files to compute the structural similarity between DOM trees. DOM tree similarity metric would reduce the number of clusters of test results. In this way, the tester only needs to check one test result in each cluster instead of every result.

After finishing tester checking, the analysis is finished, and a vulnerability report is generated.

Evaluation

Subjects:

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.

To evaluate their approach, they propose 5 research questions:

RQ1. How much overhead does their system introduce? (Performance Overhead)

RQ2. How many real-world business flow tampering vulnerabilities can their system find? (Effectiveness in Finding Vulnerability)

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

RQ4. How effective are tampering testing and result screening? (Effectiveness of Tampering Testing and Result Screening)

RQ5. How effective is their system on reducing search space? (Effectiveness in Reducing Search Space)

Tool used:

The testing module leverages the Puppeteer library, which provides high-level APIs to control Chromium over the DevTools Protocol. To collect dynamic data and perform the business flow tampering testing, instrument the target JS code by modifying V8 engine 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. They 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, they modified the JS engine because it brings in additional benefits.

Results:

- RQ1. Performance Overhead:

Fig. 4 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.

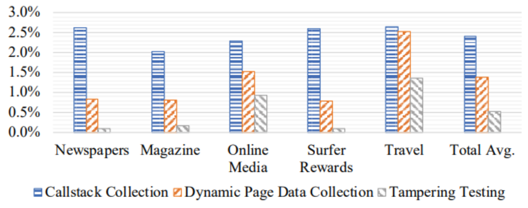


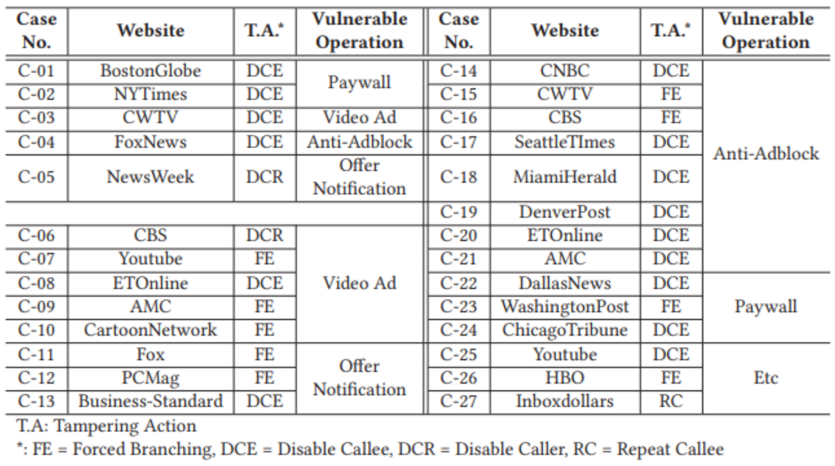
Figure 4: Normalized execution overhead

The overhead for the call stack collection step is 2.41% (90ms) on average. They 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 the dynamic page data collection step is 1.39% (70ms). From the cases that terminate normally, the average overhead is 0.53% (4ms) which is lower than dynamic page data collection.

- RQ2. Effectiveness in Finding Vulnerability:

Their technique discovers 27 vulnerable cases from 23 popular websites as shown in Table 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.

Table 3: Result of our testing on 200 websites



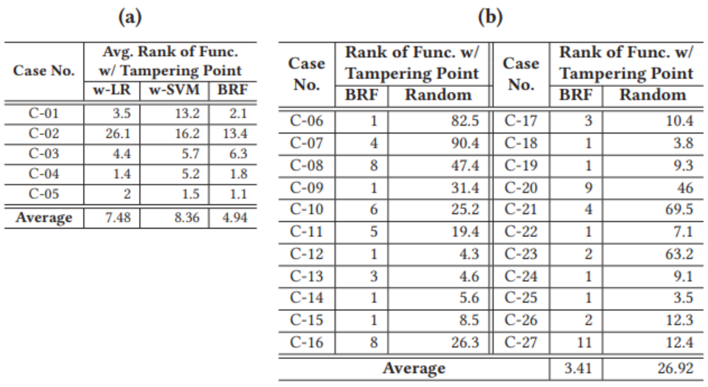
Case No.	Website	T.A.*	Vulnerable Operation	Case No.	Website	T.A.*	Vulnerable Operation
C-01	BostonGlobe	DCE	Paywall	C-14	CNBC	DCE	Anti-Adblock
C-02	NYTimes	DCE		C-15	CWTV	FE	
C-03	CWTV	DCE	Video Ad	C-16	CBS	FE	
C-04	FoxNews	DCE	Anti-Adblock	C-17	SeattleTImes	DCE	
C-05	NewsWeek	DCR	Offer Notification	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

- RQ3. Feature selection and learning algorithm:

Table 4a describes the results of the 3 classifiers, as a result, BRF shows the best average rank value. Table 4b shows the function ranks of the 22 successful cases they find after they apply the trained classifiers (BRF). As we can see in the table, the rank values with the classifier show significantly better performance than the random method.

Table 4: Function ranking with classifiers



(a)

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

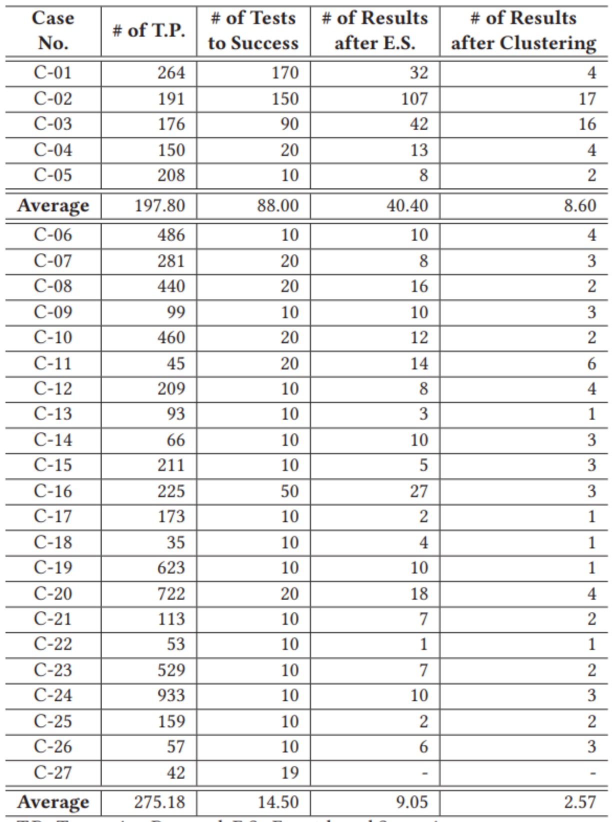
(b)

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

- RQ4. Effectiveness of Tampering Testing and Result Screening:

Table 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 they 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.

Table 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

On average, they test around 50% of the tampering proposals. The clustering and screening significantly reduce manual efforts such that testers only need to check 8 results on average, in comparison to the hundreds of 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.

- RQ5. Effectiveness in Reducing Search Space:

To evaluate the effectiveness of their filtering method, they collect statistics for the 27 successful cases. Specifically, they collect the total number of 3 types of data (JS files, functions, and branches) they have to consider before and after filtering. Fig. 5 shows the normalized numbers of each data type from 27 cases, and the last bar denotes the average number.

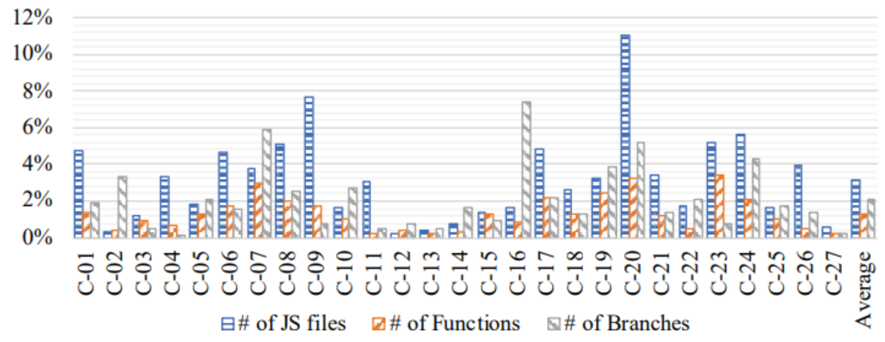


Figure 5: Effectiveness in reducing search space

On average, they 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.

<u>Reflection</u>

The paper presents a novel dynamic analysis approach that scans web applications and automatically detects client-side business flow tampering vulnerabilities. The proposed technique directly applies to web applications. The paper provides solid proof that this technique has small overhead, and discovers 27 unique vulnerabilities in popular websites, which allow an adversary to interrupt the business logic of web applications such as skipping ads at the beginning of videos, bypassing ad blocking checking, and even illicitly earning reward points. This formal guarantee is the major strength of this paper.

The technique shares some similarities with recent work to explore execution paths by forcing program execution on JS programs, native binary programs, mobile apps, and kernel rootkits. However, the 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 have also been proposed to analyze JS programs.

There are some techniques to test web applications for business logic vulnerabilities. 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. Compare with those business logic vulnerability detectors, this technique can dynamically analyze the whole client-side of web applications, instead of only analyzing the APIs of web applications.