12-12-2023

Toward the flow-centric detection of browser fingerprinting

Rui Zhao

Computers & Security 137 (2024) 103642

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Toward the flow-centric detection of browser fingerprinting

Rui Zhao

University of Nebraska at Omaha, Omaha, NE, USA

ARTICLE INFO

Keywords:
Security
Browser fingerprinting
Detection

ABSTRACT

Browser fingerprinting has become a prevalent technique employed by websites for advertising and analytics. It utilizes JavaScript objects and APIs to gather traditional and non-traditional browser attributes and creates unique identifiers for online user tracking. While previous research has examined the invocation of the browser's built-in JavaScript objects and APIs to retrieve browser attribute values, it overlooks the use and flow of those values within scripts.

In this paper, we define browser fingerprinting behavior as the aggregation of different types of browser attributes, and reduce the detection of browser fingerprinting to a joint analysis of the data flows of browser attribute values in JavaScript code. FProbe, our proposed framework for the flow-centric detection of browser fingerprinting, performs context-sensitive static data flow analysis on JavaScript code. Given the complexity and dynamic features of the JavaScript language, achieving soundness in static analysis of JavaScript code is extremely challenging or even impossible. Consequently, FProbe aims to be a practical and accurate tool for detecting browser fingerprinting. We implemented FProbe in Java and evaluated its performance using 4,296 fingerprinting scripts from recent work and 2,335,317 pieces of JavaScript code on 988,220 websites. FProbe achieved F-measures of 97.81% and 96.31% on these datasets, respectively. It identified browser fingerprinting behavior on 0.78% of the 988,220 websites, with the use of the toDataURL method observed in 4.26% of the fingerprinting scripts. Notably, only 72 fingerprinting scripts and 10 fingerprinting providers identified by FProbe were reported in previous work. These results highlight the effectiveness of FProbe in detecting browser fingerprinting and its complementarity to existing detection tools. Additionally, our comprehensive study demonstrates that fingerprinting with traditional browser attributes can achieve a 96.6% F-measure.

1. Introduction

Browser fingerprinting has become increasingly prevalent among websites for advertising and analytics over the last decade (Acar et al., 2014, 2013; Englehardt and Narayanan, 2016; Libert, 2015; Iqbal et al., 2021; Nikiforakis et al., 2013). It gathers the attributes of browsers, operating systems, and hardware (Schwarz et al., 2019) using the browser's built-in JavaScript objects and APIs such as those within window.navigator and window.screen, and then creates unique identifiers for browser instances to correlate browsing sessions across time and web domains. Studies have demonstrated that browser fingerprinting could identify over 80% of browser instances (Cao et al., 2017; Fifield and Egelman, 2015; Laperdrix et al., 2016; Mayer and Mitchell, 2012). Remarkably, even as the values of browser attributes evolve over time due to system upgrade, browser instances can still be accurately fingerprinted (Vastel et al., 2018).

To gain insights into the behavior of browser fingerprinting, we manually examined 100 unique browser fingerprinting scripts as reported in recent work (Iqbal et al., 2021) and 100 non-fingerprinting scripts. Fingerprinting scripts were found to collect and concatenate traditional and non-traditional browser attributes, utilizing objects such as navigator and screen, as well as employing techniques like canvas-based and audio-based fingerprinting. These attributes are then used to generate unique browser fingerprints, which are subsequently transmitted to the network or stored locally. This behavior is distinctly different from that observed in non-fingerprinting scripts, e.g., browser attributes relevant to the window's dimension are commonly used for adjusting the position and size of HTML DOM elements.

Inspired by these observations, we define browser fingerprinting behavior as the aggregation of different types of browser attributes, and reduce the detection of browser fingerprinting to a joint analysis of the data flows of browser attribute values in JavaScript code. Correspond-

E-mail address: ruizhao@unomaha.edu.

https://doi.org/10.1016/j.cose.2023.103642
Received 31 July 2023; Received in revised form 27 November 2023; Accepted 4 December 2023
Available online 7 December 2023

ingly, we introduce FProbe, a framework for the flow-centric detection of browser fingerprinting, since the flows of browser attribute values play the pivotal role in browser fingerprinting detection, as emphasized in Lerner et al. (2016). When provided with JavaScript code, FProbe identifies browser attributes as sources, meticulously examines the data flows originating from these sources, and computes the intersections of these data flows. It detects browser fingerprinting behavior by classifying the joint data flows associated with browser attribute values. FProbe adopts a context-sensitive static approach, maximizing its capability to detect browser fingerprinting in JavaScript code compared to dynamic approaches.

Several works for browser fingerprinting detection have been proposed (Acar et al., 2013; Nikiforakis et al., 2013; Englehardt and Narayanan, 2016; Lerner et al., 2016; Iqbal et al., 2021; Rizzo et al., 2021; Bahrami et al., 2022). They focus on the invocation of the browser's built-in JavaScript objects (e.g., canvas and audio) and APIs (e.g., navigator.appName and screen.width) to retrieve browser attribute values but overlook the use and flow of those values within scripts. Anther tool, EssentialFP (Sjösten et al., 2021), performs dynamic data flow analysis of browser attribute values but faces challenges such as incomplete coverage of code execution paths (Sabelfeld and Myers, 2003; Vogt et al., 2007), non-negligible performance overhead (Bandhakavi et al., 2010; Sjösten et al., 2021), and the requirement for software-specific instrumentation (Dhawan and Ganapathy, 2009). Su and Kapravelos (2023) also suffers from the limitations of dynamic analysis. To the best of our knowledge, this is the first work performing static data flow analysis to detect browser fingerprinting, complementary to existing browser fingerprinting detection methods. However, due to the complexity and dynamic features of the JavaScript language (e.g., runtime code generation and variadic functions), achieving soundness in static analysis of JavaScript code is extremely challenging or even impossible (Guarnieri and Livshits Gatekeeper, 2009; Madsen et al., 2013). Therefore, FProbe aims to be a practical and accurate tool.

To prevent browser fingerprinting, researchers have proposed three types of techniques that isolate or disable the execution of JavaScript, manipulate browser attributes, or fabricate browser fingerprints (Nikiforakis et al., 2015; Torres et al., 2015). However, browser fingerprinting prevention is not the focus of our work.

We implemented FProbe in Java and evaluated its performance using 4,296 fingerprinting scripts reported in recent work (Iqbal et al., 2021) and 2,335,317 pieces of JavaScript code collected from 988,220 websites. FProbe achieved a 97.81% F-measure on scripts from Iqbal et al. (2021), showing comparable performance to Iqbal et al. (2021). FProbe detected browser fingerprinting behavior in 10,570 scripts from 2,851 providers on 7,737 (0.78%) of the 988,220 websites, and achieved a 96.31% F-measure on 1,500 randomly sampled scripts. Among the 10,570 fingerprinting scripts, only 22.55% and 32.21% were recognized by EasyList Easylist (2021) and EasyPrivacy Easyprivacy (2021), respectively. Furthermore, only 72 fingerprinting scripts and 10 fingerprinting providers identified by FProbe were reported in Iqbal et al. (2021). Our findings also suggest that non-traditional browser fingerprinting techniques may not be widely used, consistent with findings in Nikiforakis et al. (2013); Englehardt and Narayanan (2016); Rizzo et al. (2021). The evaluation results highlight the effectiveness of FProbe in detecting browser fingerprinting.

To understand the capability of fingerprinting with traditional browser attributes, we conducted a comprehensive study in two phases. In the first phase, we designed a scheme for collecting traditional browser attributes and deployed it on a public website. We collected 3,073 unique traditional browser attributes during 3,814 visits from 1,790 visitors. In the second phase, we fingerprinted all the visitors using the collected browser attributes and our fingerprinting achieved a 96.6% F-measure.

The main contributions of this work include: (1) FProbe, a framework performing static data flow analysis to automatically detect browser fingerprinting in JavaScript code, (2) a comprehensive evaluation of FProbe with a comparison to existing work, (3) a large-scale measurement of browser fingerprinting on Alexa top one million websites, and (4) a comprehensive assessment of the capability of fingerprinting with traditional browser attributes.

```
function u() {
  sn = window.screen;
  return {
    size: {
      height: sn.height, width: sn.width
    },
    colorDepth: sn.colorDepth,
    pixelRatio: (t = 1, sn.systemXDPI > sn.logicalXDPI ? t = sn
        .systemXDPI / sn.logicalXDPI : t = window.
        devicePixelRatio, t)
  }; var t;
}
function a() {
  return (
    Object(function() {
      return -1 !== navigator.userAgent.indexOf("Chrome")
    })() ? t = "Chrome" : Object(function() {
      return -1 !== navigator.userAgent.indexOf("Firefox")
    })() ? t = "Firefox" : t = navigator.appName, t); var t;
}
function getfp() {
  return {
    screen: u(),
    browser: {
      browser: a(),
      mobile: /Mobile|Android|iP(ad|od|hone)/.test(navigator.
          appVersion),
    }, ...
  }
}
```

Fig. 1. Code excerpt from a real browser fingerprinting script.

The rest of this paper is organized as follows. Section 2 introduces FProbe and its design rationale. Section 3 provides implementation details. Sections 4 and 5 evaluate FProbe and measure browser fingerprinting in the wild. Section 6 explores the capability of fingerprinting with traditional browser attributes. Section 7 discusses the limitations and future work. Section 8 examines the related work. Section 9 concludes the paper.

2. Design of FProbe

In this section, we conduct a manual analysis of browser fingerprinting scripts reported in previous work and subsequently introduce FProbe along with its design rationale.

2.1. Manual analysis of browser fingerprinting scripts

We manually examined 100 unique browser fingerprinting scripts reported in recent work (Iqbal et al., 2021). Of these, 98 scripts collect and concatenate traditional browser attributes for generating fingerprints. These browser attributes are directly obtained from the browser's built-in JavaScript objects and APIs. They describe the properties of the browser (e.g., screen.width, navigator.appVersion, and navigator.language) and the machine (e.g., navigator.cpuClass and navigator.deviceMemory), as well as other information (e.g., Date.getTimezoneOffset). Meanwhile, all 100 fingerprinting scripts employ non-traditional browser fingerprinting techniques, such as canvas, font, WebRTC, performance, and audio-based methods.

We also examined 100 non-fingerprinting scripts. They often utilize individual browser attributes or similar types of attributes for rendering purposes. For instance, these scripts use browser attributes related to window.screen to adjust the size of HTML DOM elements, without incorporating attributes like navigator.cookieEnabled.

The code excerpt extracted from a real fingerprinting script, as shown in Fig. 1, illustrates browser fingerprinting behavior. Several

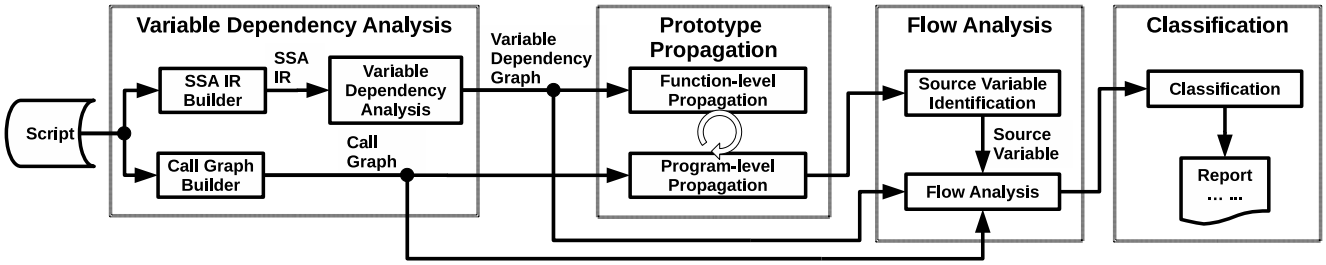

Fig. 2. The overall workflow of the browser fingerprinting detection framework, FProbe.

traditional browser attributes are combined in a returned JavaScript object within the entry function getfp(). Seven of them, obtained in the function u(), describe the properties of browser window; two of them, obtained in the function a(), describe the browser type; and others obtained in the function getfp() describe the browser version.

2.2. Design overview and rationale

Inspired by these observations, we define browser fingerprinting behavior as the aggregation of different types of browser attributes, and reduce the detection of browser fingerprinting to a joint analysis of the data flows of browser attribute values in JavaScript code. Correspondingly, we introduce FProbe for the flow-centric detection of browser fingerprinting. It comprises four phases, as illustrated in Fig. 2.

In the initial phase, *variable dependency analysis*, when provided with JavaScript code, the *call graph builder* and the *SSA IR builder* components statically construct a call graph and SSA IR,[1] respectively. Leveraging the SSA IR, the *variable dependency analysis* component then generates variable dependency graphs for all functions, illustrating value dependencies among variables. These dependencies capture the value flows among variables and operations during the flow (e.g., string, array, and arithmetic ones).

In the next phase, *prototype propagation*, based on the variable dependency graphs and the call graph, the *function-level propagation* and *program-level propagation* components iteratively propagate JavaScript values and prototypes over all variables within and across functions, respectively. This propagation process terminates once the possible values and prototypes of all variables are no longer subject to updates.

In the third phase, *flow analysis*, based on the propagated values and prototypes, the *source variable identification* component identifies source variables that receive browser attribute values obtained from fingerprinting-related JavaScript objects and APIs. Subsequently, the *flow analysis* component conducts a context-sensitive data flow analysis starting from the source variables and computes the intersections of these data flows throughout the entire program.

In the final phase, *classification*, based on the data flow joints of browser attribute values, FProbe generates a report when browser fingerprinting is detected. The report lists all the attributes involved in browser fingerprinting and details their flow paths in the JavaScript code.

FProbe adpots a pure static approach, maximizing its capability to detect browser fingerprinting in JavaScript code compared to dynamic approaches. It captures both explicit and implicit data flows[2] in a context-sensitive way. While many existing browser fingerprinting detectors concentrate on which browser attributes are referenced (Acar et al., 2013; Nikiforakis et al., 2013; Englehardt and Narayanan, 2016; Lerner et al., 2016; Iqbal et al., 2021; Rizzo et al., 2021), FProbe emphasizes the flows of browser attribute values within the code, which is essential for the reliable detection of browser fingerprinting (Lerner et al., 2016). Please refer to Section 8 for detailed discussions.

Due to the complexity and dynamic features of the JavaScript language (e.g., runtime code generation and variadic functions), achieving soundness in static analysis of JavaScript code is extremely difficult or even impossible (Guarnieri and Livshits Gatekeeper, 2009; Madsen et al., 2013), unlike most static analysis tools for typed programming languages (e.g., C and Java) choosing to be sound. Consequently, FProbe is designed to be a practical and accurate tool, complementary to existing browser fingerprinting detection methods (Acar et al., 2013; Nikiforakis et al., 2013; Englehardt and Narayanan, 2016; Lerner et al., 2016; Iqbal et al., 2021; Rizzo et al., 2021; Sjösten et al., 2021; Bahrami et al., 2022).

2.3. Variable dependency analysis

Call graphs serve as the foundation of program analysis (Grove et al., 1997; Weihl, 1980). FProbe statically constructs call graphs to ensure comprehensive coverage of code execution paths by resolving call sites to their respective targets. The constructed call graphs are directed graphs with nodes representing functions and edges representing call relations from caller functions to callee functions through call sites. It is important to note that accurately constructing call graphs for JavaScript programs is challenging due to the dynamic features of JavaScript, such as runtime code generation and variadic functions (Chugh et al., 2009; Just et al., 2011; Madsen et al., 2013; Nikiforakis et al., 2012; Richards et al., 2011, 2010; Yue and Wang, 2013). Although accurate call graphs can be dynamically constructed for JavaScript code, generating proper inputs to trigger all possible code execution paths while minimizing runtime overhead remains a challenging task.

Given the SSA IR (Cytron et al., 1991) of a function, the *variable dependency analysis* component extracts operands (i.e., value numbers) and operators from SSA instructions, and constructs a variable dependency graph to describe immediate value dependencies among value numbers. Converting the script to its SSA form enables accurate correlation between the definitions and uses of each value number, even when variables are reused in JavaScript code.

We classify operators in the SSA IR into two groups, *basic* and *global*, similar to the approach in Zhao et al. (2015). Basic operators are directly extracted from SSA instructions, representing value transformation and passing from right-hand-side value numbers to the left-hand-side value number in an SSA instruction within functions. As shown in Formula (1), the basic operators include prototype lookup (PROTOTYPE), unary/binary operation (UNARY and BINARY), read/write to object field (FIELD_GET and FIELD_PUT), conditional expression (CONDITION), object construction (CONSTRUCT), and Φ-function (PHI) (Cytron et al., 1991) in the SSA IR. For instance, a string concatenation corresponds to a BINARY_OP operation, while reading a field is a FIELD_GET operation. Any operation that cannot be recognized will be labeled as UNKNOWN.

$$Operator_{basic} = \{PROTOTYPE, UNARY, BINARY,$$
$$FIELD\_GET, FIELD\_PUT, CONDITION,$$

[1] Static Single Assignment form Intermediate Representation (Cytron et al., 1991).

[2] Implicit data flows often occur in conditional statements, JavaScript reflection, event/message handlers, and asynchronous function calls.

$$CONSTRUCT, PHI, UNKNOWN\} \quad (1)$$

We denote the variable dependency graph for function f as $D(f)$ in Formula (2), in which actual operations on variables are converted to their corresponding basic operators. The graph's nodes correspond to value numbers defined or used within the function, while edges represent operations propagating values from one value number to another, indicating dependency relations.

$$D(f) = \{x \xrightarrow{op} y \mid x, y \text{ are value numbers in function } f, \text{ where } y = op(x) \text{ and } op \in Operator_{basic}\} \quad (2)$$

2.4. Prototype propagation

In strongly typed programming languages such as C and Java, data types can be statically determined at variable declarations. Unlike them, the prototype of the value in a JavaScript variable cannot be easily inferred in static code analyses. However, prototypes play a crucial role in browser attribute identification. For example, in Fig. 1, the variable sn holds the reference to the screen object in line 2, and its field height is referenced in line 5. Without knowledge of the prototype of sn's value, it is impossible to identify the browser attribute screen.height in line 5. Therefore, the two components of this phase iteratively propagate JavaScript values and prototypes throughout the script, as illustrated in Fig. 3.

The propagation is an iterative process that is context-sensitive to the call stack and conditional expressions, as illustrated at the top of Fig. 3. It iterates through functions in a post-order traversal of the call graph G to ensure that callees can be processed before their callers whenever possible. The propagation terminates when there are no more updates to the propagated values and prototypes in the entire script. To prevent endless iteration, we have set a timeout for the propagation.

2.4.1. Function-level propagation

Utilizing the variable dependency graphs, depicted in the propagate-function-level of Fig. 3, the *function-level propagation* component iterates through all instructions in function f and propagates the value or prototype from right-hand-side value numbers to the left-hand-side value number of every instruction *stmt* intra-procedurally under the current call stack *stack*.

Specifically, for a statement which performs a prototype look-up on x (at line 3), the prototype of x will be saved to the left-hand-side value number y. For a statement which reads the field of an object x to a value number y (in line 5), the value of x's field will be saved to y. For a statement which writes a value number x to the field of an object y (in line 7), the value of x will be saved to y's field. The read and write operations on array elements are also covered in lines 5 and 7. If the index of the array element can be resolved, the propagation will be performed on such an element otherwise the entire array. For a statement which constructs an object y from a prototype x (in line 9), the prototype of y will be set to x. For a statement in which y receives one of multiple objects $x_1, \ldots, x_n$ (in line 11), the value of y will be the set of all possible values $x_1, \ldots, x_n$. We do not propagate anything in unary/binary operations and conditional statements since their computation results cannot be used to obtain browser attributes.

2.4.2. Program-level propagation

Building on the outcomes of function-level propagation and the call graph, illustrated in the propagate-program-level of Fig. 3, the *program-level propagation* component propagates values and prototypes from the definitions of global/lexical variables to their uses, from the arguments of callsites in caller functions to the parameters of callee functions, from the returned variables of callee functions to the receiving variables at the callsites in caller functions, and from event/message dispatchers to their handlers across function boundaries (inter-procedurally) under the current call stack. Corresponding dependencies are established across functions.

```
1  while true
2      for each function f ∈ call graph G do
3          propagate-value-function-level(f)
4          propagate-value-program-level(f, G)
5      if no value update then exit

function propagate-function-level(f)
// f: a function
1      for each statement stmt ∈ f do
2          switch stmt do
3              case y = prototype_of(x) do
4                  y ← x._proto_ | stack
5              case y = x.field do
6                  y ← x.field | stack
7              case y.field = x do
8                  y.field ← x | stack
9              case y = new x() do
10                 y._proto_ ← x | stack
11             case y = phi(x_1, ..., x_n) do
12                 y ← {x_1, ..., x_n} | stack

function propagate-program-level(f, G)
// f: a function
// G: the call graph
1      for each statement stmt ∈ f do
2          switch stmt do
3              case global||lexical : y = x do
4                  reads = find-read(stmt)
5                  for each [z = global||lexical : y ∈ function t] ∈ reads do
6                      z ∈ t ← x ∈ f | new stack
7              case y = invoke m arg_1, arg2, ... do
               case y = dispatch n.m, arg_1, arg2, ... do
8                  function t(para_1, para_2, ...) = find-call-target(stmt, G)
9                  para[i] ∈ t ← arg[i] ∈ f where i = 1, 2, ... | stack ← stmt
10             case return x do
11                 if stack.size == 0 then
12                     callsites = find-call-site(f, G)
13                 else
14                     frame r ← stack
15                     callsites = find-call-site(f, r)
16                 for each callsite : y = f(...) ∈ function t ∈ callsites do
17                     y ∈ t ← x ∈ f | stack
18             case y = dispatch *.[dispatchEvent,fireEvent] x_1, x_2, ... do
19                 ename = find-event-name(stmt)
20                 z_1, z_2, ... = find-event-receiver-arg(ename)
21                 z[i] ← x[i] where i = 1, 2, ... | new stack
```

Fig. 3. Value and prototype propagation algorithm.

As shown in Formula (3), global operators represent value passing across functions. We use GLOBAL_GET and LEXICAL_GET to denote reading from global and lexical variables, and use GLOBAL_PUT and LEXICAL_PUT to represent writing to global and lexical variables. We use INVOKE and DISPATCH to represent function calls and member method dispatches, respectively, and RETURN to denote value returns.

$$Operator_{global} = \{GLOBAL\_GET, GLOBAL\_PUT, LEXICAL\_GET, LEXICAL\_PUT, INVOKE, DISPACTH, RETURN\} \quad (3)$$

As shown in propagate-program-level of Fig. 3, for a statement which writes the value of variable x to a global/lexical variable y in function f (in line 3), we search for the statement reading y's value to another variable z in function t (find-read in line 4), and propagate the value from x to z under a new call stack (in line 6). It is worth noting that roughly propagating values from a write statement to all read statements with the same global/lexical variable could produce false positives. Therefore, based on the call graph, find-read in line 4 only searches for subsequent read statements prior to the next write statement for such a global/lexical variable.

For a statement which invokes a function or dispatches a member method in function f (in line 7), we first resolve the call target t based on the call graph G (find-call-target in line 8), and then propagate the value from each argument $arg[i]$ in the caller function f to its corresponding parameter $para[i]$ of the callee function t (in line 9). Meanwhile, a new stack frame is pushed to the top of the call stack $stack$. The stack frame includes the caller, the call site, arguments, and parameters.

For a statement which returns x in function f (in line 10), if the call stack is empty, we search the whole program for all call sites pointing to f (find-call-site in line 12); otherwise, we leverage the top frame of the call stack to find call sites pointing to f (in lines 14 and 15). We then propagate the value from the returned variable x in the callee function f to the receiving variable y at the call site in the caller function t (in line 17). Note that the propagation for return statements is a N-to-N mapping. It means several return statements can be identified for one call site, and several call sites can also be identified for one return statement when the call stack is empty.

For a statement which dispatches an event containing $x_1, x_2, \ldots$ (in line 18), we use the event name to search the whole program for its corresponding argument $z_1, z_2, \ldots$ at the receiving function (in lines 19 and 20). We then propagate every dispatched $x[i]$ to its corresponding argument $z[i]$ (in line 21).

2.5. Flow analysis

Based on the propagated values and prototypes, the *source variable identification* component identifies source variables that receive browser attribute values. The identification focuses on three types of JavaScript operations. The first two involve field references (e.g., navigator.userAgen) and method dispatches (e.g., navigator.javaEnabled). Identification in these cases relies on the prototypes of JavaScript objects and the names of their fields or methods. The third type is the *typeof* operation (e.g., typeof navigator.appName), commonly utilized to check the existence of JavaScript objects and APIs.

The *flow analysis* component conducts a context-sensitive interprocedural data flow analysis starting from all source variables and summarizes transitive relations among value numbers along the data flows. It computes locations within the program where the flows of browser attribute values converge (e.g., in an array or a JSON object). The algorithm for flow analysis, depicted in Fig. 4, is a recursive process based on variable def-use analysis. Given a variable x at the end of the flow $flow$ in function f, we search for all statements where variable x is used in f (find-use in line 1), analyze each of x's uses (starting from line 2), and record the transitive relation, as defined in Formula (4), for the flow.

$$\begin{aligned} T^{f_i,f_j}(x,y) = \{(x^{f_i}, op_1, v_1^{f_1}, \ldots, v_k^{f_k}, op_{k+1}, y^{f_j}) \mid \\ functions\ f_i, f_1, \ldots, f_k, f_j \in call\ graph\ G, \\ variables\ x^{f_i} \in f_i, v_1^{f_1} \in f_1, \ldots, v_k^{f_k} \in f_k, y^{f_j} \in f_j, \\ operations\ op_1, \ldots, op_{k+1} \in Operator_{basic} \cup Operator_{global}, \\ \exists\ x^{f_i} \xrightarrow{op_1} v_1^{f_1}, \ldots, v_k^{f_k} \xrightarrow{op_{k+1}} y^{f_j}\} \end{aligned} \tag{4}$$

For a statement which puts the value of x into the field of an object y (in line 4), we first look up all statements that read the value of y.field to another variable z (find-read in line 5). For each read statement, we add a flow segment from x to z via y.field to $flow$ (in line 7), and recursively analyze the use of z (in line 8). The operation on array elements is also covered here. For a write operation on an array element, the analysis will be performed on such an element if the element index can be resolved otherwise the entire array.

For a statement which writes the value of x to a global/lexical variable y (in line 9), we first look up all the statements which read the value of y to another variable z (find-read in line 10). For each read statement, we add a flow segment from x to z via y to $flow$ (in line 12), and recursively analyze the use of z (in line 13).

```
// s: a source variable
// d: the function where s is defined
// G: the call graph
1 flow-analysis(s, d, {}, G)

function flow-analysis(x, f, flow, G)
1     statements uses = find-use(x, f)
2     for each statement use ∈ uses do
3         switch use do
4             case y.field = x do
5                 reads = find-read(use)
6                 for each [z = y.field ∈ function t] ∈ reads do
7                     flow' = flow + z ∈ t ← y.field ← x ∈ f
8                     flow-analysis(z, t, flow') | new stack
9             case global‖lexical : y = x do
10                reads = find-read(use)
11                for each [z = global‖lexical : y ∈ function t] ∈ reads do
12                    flow' = flow + z ∈ t ← global‖lexical : y ← x ∈ f
13                    flow-analysis(z, t, flow') | new stack
14            case y = invoke m ..., x, ... do
              case y = dispatch n m, ..., x, ... do
15                function t(..., z, ...) = find-call-target(use, G)
16                flow' = flow + z ∈ t ← x ∈ f
17                flow-analysis(z, t, flow') | stack ← use
18            case return x do
19                if stack.size == 0 then
20                    callsites = find-call-site(f, G)
21                else
22                    frame r ← stack
23                    callsites = find-call-site(f, r)
24                for each callsite : y = f(...) ∈ function t ∈ callsites do
25                    flow' = flow + y ∈ t ← x ∈ f
26                    flow-analysis(y, t, flow') | stack
27            case y = dispatch *.[dispatchEvent,fireEvent] ..., x, ... do
28                ename = find-event-name(stmt)
29                z ∈ t = find-event-receiver-arg(ename)
30                flow' = flow + z ∈ t ← x ∈ f
31                flow-analysis(z, t, flow') | new stack
32            default y = op ..., x, ... do
33                flow' = flow + y ← x
34                flow-analysis(y, f, flow') | stack
```

Fig. 4. Flow analysis algorithm.

For a statement which invokes a function m or dispatches a method $m.n$ (in line 14), we first find the call target t (find-call-target in line 15), then add a flow segment from the argument x in f to its corresponding parameter z in t to $flow$ (in line 16), push the current call site use at the top of the call stack $stack$, and recursively analyze the use of z with the updated call stack (in line 17).

For a statement which returns x in function f (in line 18), if the call stack is empty, we search the whole program for all call sites pointing to f (find-call-site in line 20); otherwise, we leverage the top frame of the call stack to find call sites pointing to f (in lines 22 and 23). For every call site in function t, where variable y receives the returned x from f, we add a flow segment from x to y to $flow$ (in line 25), and recursively analyze the use of y in t (in line 26).

For a statement which dispatches an event containing x (in line 27), we use the event name to search the whole program for its corresponding argument z at the receiving function t (in lines 28 and 29). We then add a flow segment from the dispatched x to its corresponding receiving argument z to $flow$ (in line 30), and recursively analyze the use of z in t (in line 31).

For other statements in which x is used and a variable y is defined (e.g., a binary operation) (in line 32), we add a flow segment from x to y to $flow$ and head to the use analysis of y (in lines 33 and 34).

The log below illustrates the data flow of the browser attribute navigator.appName in the function a() as shown in Fig. 1. Its hierarchical structure depicts the transitive relationships among value numbers during the flow. At each step, the value numbers (e.g., 37 in line 1) and the instruction (e.g., 37 = getfield <appName> 6) are recorded. For

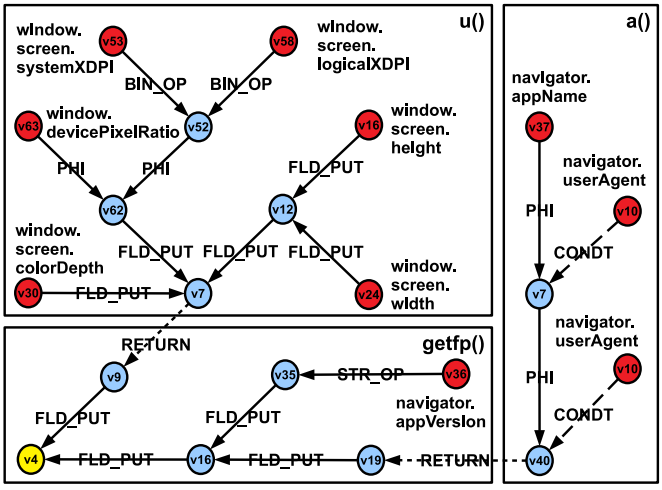

Fig. 5. Flows of browser attributes in Fig. 1.

the flow from a callee to its caller in line 4, we use two instructions: the return statement itself in the callee (e.g., "|-1|return 40") and its corresponding call site in the caller (e.g., "|19|19 = invoke 21@15 22 exception:23").

```
|37|37 = getfield <appName> 6
-|7|7 = phi 36,37
--|40|40 = phi 23,7
---|-1|return 40 & |19|19 = invoke 21@15 22 exception:23
----|16|fieldref 16.15 = 19
-----|4|fieldref 4.15 = 16
------|-1|return 4
```

The log below shows a flow joint of browser attribute values in Fig. 1. The first line indicates the instruction where the value flows of browser attributes converge, along with the total number of joined attributes (i.e., 9). The subsequent lines present the browser attributes and the corresponding code for retrieving their values.

```
JOINT|fieldref 4.15 = 16|9
-10 = getfield <userAgent> 8|navigator.userAgent
-37 = getfield <appName> 6|navigator.appName
-16 = getfield <height> 21|screen.height
-30 = getfield <colorDepth> 33|screen.colorDepth
-53 = getfield <systemXDPI> 50|screen.systemXDPI
-36 = getfield <appVersion> 26|navigator.appVersion
-58 = getfield <logicalXDPI> 57|screen.logicalXDPI
-63 = getfield <devicePixelRatio> 61| window.devicePixelRatio
-24 = getfield <width> 27|screen.width
```

Fig. 5 visually illustrates the flow analysis results of our code example, including browser attributes and the connections among the value flows of these attributes. The value number $v4$ in the function getfp() is the joint point for all the browser attributes. The edges in the figure are annotated with corresponding operators (e.g., BINARY_OP, FIELD_PUT, RETURN, and PHI). Notably, the label on the edge from $v36$ to $v35$ in the function getfp() has been updated to STR_OP for dispatching the method test() of a string value.

2.6. Classification

We observed four patterns related to the flow of browser attribute values in fingerprinting scripts during our manual analysis (Section 2.1). (1) Attribute values are aggregated together and then flow to a single sink point, as shown in Fig. 1. (2) Attribute values are individually sent to the same sink point, such as navigator.sendBeacon(). (3) Attribute values are aggregated in a JavaScript object without any sink point in the script, which is notable in browser fingerprinting libraries. (4) Attribute values are aggregated in several JavaScript objects and/or flow to several sink points.

Correspondingly, we define two types of browser fingerprinting behavior. The $Type_1$ behavior corresponds to the first three patterns discussed above. Over t_1 values from g types of browser attributes flow to one JavaScript object (e.g., an array or a string) or one dispatched method (e.g., XMLHttpRequest.send). The type of a browser attribute is determined by what browser property such an attribute describes. For example, window.devicePixelRatio and screen.width are of the same type, while navigator.language and navigator.userAgent are of different types. We use two threshold values, t_1 and g, to reduce false positives. A typical scenario we observed is that attributes under window.screen and window.document are often used together to calculate the size of DOM elements for rendering but not fingerprinting purposes.

The $Type_2$ behavior corresponds to the last pattern. Over t_2 values from g types of browser attributes are referenced but flow to multiple JavaScript objects and/or dispatched methods. For instance, some fingerprinting scripts send browser attribute values individually at different locations in the scripts.

Note that only browser attributes known to be effective for browser fingerprinting are counted toward the thresholds for both types of browser fingerprinting behavior. In Section 4, we will provide more details about the selection of browser attributes and the three threshold values.

3. Implementation

We implemented FProbe in Java, utilized the Closure compiler (Closure Compiler, 2021a) to identify JavaScript functions and resolve call sites, and used the WALA compiler (WALA Compiler, 2021b) to generate SSA IR. While various compilers for JavaScript call graph construction are available, they are often implemented in languages other than Java. Notably, researchers have found that the Closure compiler outperforms the WALA compiler regarding the call graph construction in aspects such as recursive calls, inline functions, and runtime performance (Antal et al., 2018). Throughout our implementation, we also observed that the WALA compiler tends to be over-conservative, thus easily missing nodes and edges in the call graph, and practically unusable for analyzing code at a scale, such as thousands of lines of code. Although the Closure compiler may occasionally use name-matching for resolving call sites and could introduce false or missing edges in the call graph, this does not contradict one of our design goals - FProbe aims to be a practical and accurate tool, rather than a sound one (Section 2.2).

To improve FProbe's runtime performance, we cached data dependencies and flow paths that had been explored in the *prototype propagation* phase and reused them in the *flow analysis* phase.

We used a configuration file to maintain the list of JavaScript APIs for browser attribute identification. For instance, we use simple expressions navigator.languages[*] and navigator.javaEnabled to match the reference to any element in the language array and the call to javaEnabled method, respectively. We use the compound expression such as navigator.mediaDevices.enumerateDevices()[*].deviceId to match the chained operations on JavaScript objects. The list of JavaScript APIs can be updated in a timely manner following the evolving browser fingerprinting techniques.

FProbe can also discover emerging browser fingerprinting techniques. For instance, we can use the expression navigator.* to analyze the flow of every object under the navigator and then identify new browser attributes used in conjunction with those already well-known for fingerprinting. We believe that our flow-centric approach is more accurate and reliable than (Bahrami et al., 2022).

4. FProbe with scripts from Iqbal et al. (2021)

We successfully downloaded 4,296 out of the 4,493 fingerprinting scripts reported in Iqbal et al. (2021) and excluded 2,164 of them that

were empty or had syntax errors to yield 1,347 unique ones, of which 148 were obfuscated. FProbe was then evaluated using these 1,347 unique scripts and compared with (Iqbal et al., 2021). We were unable to utilize results from other work on browser fingerprinting detection (Acar et al., 2013, 2014; Englehardt and Narayanan, 2016; Bahrami et al., 2022), due to the unavailability of their detection results or issues with their source code.

We manually examined the 1,347 unique scripts in our best effort, and labeled 1,024 of them as fingerprinting and 268 as non-fingerprinting. The manual analysis results served as the ground truth. We labeled a script as fingerprinting when it gathers browser attributes of different types not for rendering purposes or when it employed any non-traditional fingerprinting technique (e.g., canvas, font, WebRTC, and audio-based methods) - adhering to the criteria used in Iqbal et al. (2021). Additionally, we evaluated the script in the browser and modified runtime values of browser attributes like navigator.language and screen.width to assess their impact on web page rendering. When a script loaded and utilized a fingerprinting script,[3] we labeled it as non-fingerprinting.

During the manual analysis, we considered several commonly used obfuscation techniques, including eval and code and array packers like "JavaScript Obfuscator" and "Obfuscator.io". We failed to deobfuscate 55 scripts and labeled them as unknown. It is worth noting that we do not classify minified scripts as obfuscated since their code remains readable. The 148 obfuscated scripts were fed to FProbe in their original form.

In summary, FProbe achieved a 99.8% precision, a 95.9% recall, a 97.81% F-measure, and a 96.59% accuracy with a low false discovery rate on the 1,292 unique fingerprinting and non-fingerprinting scripts reported in Iqbal et al. (2021). FProbe also exhibited satisfactory performance on obfuscated fingerprinting scripts. Nevertheless, we strongly recommend FProbe to be used with JavaScript deobfuscators.

4.1. Effective browser attributes and threshold values

We identified 62 effective browser attributes for fingerprinting and grouped them into 12 categories based on their purposes, as detailed in Table 1. These attributes were compiled from prior research (Acar et al., 2013; Nikiforakis et al., 2013; Libert, 2015; Englehardt and Narayanan, 2016; Lerner et al., 2016; Laperdrix et al., 2016; Gómez-Boix et al., 2018; Iqbal et al., 2021; Rizzo et al., 2021; Sjösten et al., 2021; Bahrami et al., 2022) and the Browserize project (Browserize, 2022). These categories are about characters, languages, plugins, storage, do-not-track flags, screen and device properties, software details, Java and cookie support, timezone, and Canvas objects. Through a grid search with the 1,347 unique scripts and the 62 effective browser attributes, we determined the optimal values for thresholds t_1, t_2, and g in Section 2.6 as 6, 14, and 3, respectively.

4.2. Detection accuracy

Out of the 1,292 unique fingerprinting and non-fingerprinting scripts, FProbe identified 982 true positives, 2 false positives, 266 true negatives, and 42 false negatives. It achieved a 99.8% precision, a 95.9% recall, a 97.81% F-measure, and a 96.59% accuracy, as detailed in Table 2. FProbe's accuracy on scripts reported in Iqbal et al. (2021) is slightly lower than the results in Iqbal et al. (2021) with static features only, likely influenced by non-duplicated scripts and the limited number of non-fingerprinting scripts in our experiment. However, the analysis of scripts from the wild in Section 5 indicates that FProbe's accuracy is comparable to that reported in Iqbal et al. (2021). We define a true positive (TP) as fingerprinting behavior identified in a fingerprinting script, a false positive (FP) as fingerprinting behavior falsely identified in a non-fingerprinting script, a true negative (TN) as no fingerprinting behavior identified in a non-fingerprinting script, and a false negative (FN) as no fingerprinting behavior identified in a fingerprinting script. FProbe did not detect browser fingerprinting behavior in the 55 obfuscated scripts with unknown purposes.

Table 1
Attributes effective for browser fingerprinting.

Category	Browser Attribute	FP Sites
Character	document.characterSet	8.19%
	document.charset	4.19%
	document.charSet	4.19%
Language	navigator.languages[*]	1.40%
	navigator.language	52.58%
	navigator.browserLanguage	5.30%
	navigator.userLanguage	33.66%
	navigator.systemLanguage	4.25%
Plugins	navigator.mimeTypes.length	9.18%
	navigator.mimeTypes[*].description	0.78%
	navigator.mimeTypes[*].suffixes	0.62%
	navigator.mimeTypes[*].type	1.69%
	navigator.plugins.length	21.53%
	navigator.plugins[*].length	0.18%
	navigator.plugins[*].description	18.68%
	navigator.plugins[*].filename	1.93%
	navigator.plugins[*].name	8.84%
	navigator.plugins[*].version	1.65%
	navigator.plugins[*][*].description	0.10%
	navigator.plugins[*][*].suffixes	0.10%
	navigator.plugins[*][*].type	0.12%
Database	window.indexedDB	3.97%
	window.localStorage	32.30%
Do-not-Track	window.doNotTrack	5.42%
	navigator.doNotTrack	5.40%
	navigator.msDoNotTrack	0.93%
Touch Points	navigator.maxTouchPoints	6.19%
	navigator.msMaxTouchPoints	3.94%
Screen	window.devicePixelRatio	16.49%
	window.innerHeight	50.87%
	window.innerWidth	47.73%
	window.outerHeight	11.65%
	window.outerWidth	10.65%
	screen.availHeight	16.70%
	screen.availWidth	16.97%
	screen.height	63.91%
	screen.width	65.83%
	screen.colorDepth	31.65%
	screen.pixelDepth	7.99%
	visualViewport.height	0.09%
	visualViewport.width	0.09%
Devices	navigator.cpuClass	4.48%
	navigator.deviceMemory	1.76%
	navigator.hardwareConcurrency	2.49%
	navigator.oscpu	2.56%
	navigator.mediaDevices	0.34%
Software	window.name	82.06%
	navigator.appCodeName	1.10%
	navigator.appName	14.95%
	navigator.appVersion	8.22%
	navigator.buildID	0.40%
	navigator.platform	15.77%
	navigator.product	1.84%
	navigator.productSub	2.58%
	navigator.userAgent	76.22%
	navigator.vendor	5.79%
	navigator.vendorSub	0.12%
	navigator.version	0.43%
Java & Cookie	navigator.cookieEnabled	27.14%
	navigator.javaEnabled	18.70%
Timezone	Date.getTimezoneOffset	50.39%
DataURL	HTMLCanvasElement.toDataURL	4.30%

[3] An example is https://v2.clickguardian.app/track.js.

Table 2
FProbe's overall performance compared with (Iqbal et al., 2021). * is the results for 1,500 sampled scripts.

	Precision	Recall	F-Measure	Accuracy
FProbe with scripts from Iqbal et al. (2021)	99.8%	95.9%	97.81%	96.59%
FProbe in the wild *	94.23%	98.49%	96.31%	99%
Iqbal et al. (2021) static features	92.7%	85.5%	89%	99.8%
Iqbal et al. (2021) dynamic features	99.1%	96.7%	97.9%	99.9%
Iqbal et al. (2021) static and dynamic features	93.1%	93.8%	93.4%	93.1%

Table 3
FProbe's performance on $Type_1$ and $Type_2$ browser fingerprinting. * is the results for sampled scripts.

	Type	TPs	FPs	FDR
FProbe with scripts from Iqbal et al. (2021)	$Type_1$	964	2	0.2%
	$Type_2$	18	0	0%
FProbe in the wild *	$Type_1$	172	4	2.27%
	$Type_2$	24	8	25%

We further counted true and false positives in $Type_1$ and $Type_2$ browser fingerprinting and calculated their false discovery rates (FDR), as shown in Table 3. FProbe demonstrated excellent and comparable performance in detecting both types of browser fingerprinting in the scripts from Iqbal et al. (2021). Notably, $Type_1$ fingerprinting was identified in 964 (94.14%) of the 1,024 unique fingerprinting scripts from Iqbal et al. (2021), with the majority (98.17%) of our detected browser fingerprinting falling under $Type_1$. This indicates that a significant portion of fingerprinting scripts are indeed aggregating browser attributes.

4.3. False positives and false negatives

Two false positives are due to the overestimated data flows of lexical/global variables, a common issue in static code analysis due to the lack of runtime context, such as conditions and the call stack information (Guarnieri and Livshits Gatekeeper, 2009; Madsen et al., 2013). Achieving soundness in the static analysis of JavaScript code can thus be highly challenging.

We investigated the causes of 42 false negatives. 23 of them are due to code obfuscation; FProbe's limitation on obfuscated code will be studied later. Ten are due to the exclusive use of canvas-based or WebRTC-based fingerprinting, while JavaScript APIs for WebRTC-based fingerprinting were missing in FProbe's configuration. Six are due to the lack of runtime context during the analysis. Only two false negatives are due to FProbe's thresholds, as their scripts aggregate five browser attributes of two types, falling below our thresholds. It illustrates that our selected threshold values are neither underfitting nor overfitting. The code segment below depicts the script of the last false negative,[4] where it collects browser attributes by iterating through JavaScript objects, and FProbe failed to recognize it.

```
var FS = "";
for (var key in navigator) {
  if (key != "cookieEnabled") {
    if (typeof navigator[key] == "string") {
      FS += navigator[key]
    }
  } else {
    FS += false
  }
}
```

[4] http://www.childcare.go.kr/html/AnySign/AnySign4PC/ext/xcryptoCore_min.js?version=20190628.

Table 4
FProbe's performance on deobfuscated code.

	TP	FP	TN	FN
Completely Deobfuscated	6	0	5	4
Partially Deobfuscated	0	0	16	13

4.4. Obfuscated scripts

Among the 93 obfuscated fingerprinting and non-fingerprinting scripts, FProbe successfully resolved certain types of obfuscations and detected 49 true positives, 21 true negatives, and 23 false negatives. The following code snippet provides a simplified example of using arrays to pack values and later unpacking the arrays for use. Due to FProbe's prototype and value propagation, the packed values are correlated with their indexes in the array (i.e., 'userAgent' at the index 0 and 'width' at the index 1 in line 1). These values can then be accurately referenced by their indexes (i.e., 'userAgent' and 'width' are referenced by _0x5368[0] and _0x5368[1], respectively, in line 2).

```
var _0x5368 = ["userAgent", "width"];
var _0x42e7e9 = navigator[_0x5368[0]] + screen[_0x5368[1]]
```

To further investigate the impact of obfuscation, we ran FProbe on the deobfuscated code of 44 negatives, 29 of which were only partially deobfuscated due to unknown array/code packers. The 55 scripts we could not deobfuscate in the manual analysis were excluded. Table 4 demonstrates FProbe's performance on these 44 deobfuscated scripts.

The results indicate that FProbe can achieve the high precision when scripts are completely deobfuscated. Among the completely deobfuscated scripts, three false negatives occurred because their scripts exclusively use font-based browser fingerprinting. The last false negative is due to unresolved references to browser attributes. Unfortunately, FProbe struggled to accurately classify all 29 partially deobfuscated scripts. Therefore, we strongly recommend FProbe to be used with JavaScript deobfuscators.

5. FProbe with scripts from the wild

To detect browser fingerprinting in the wild, we downloaded 2,335,317 pieces of JavaScript code from 988,220 websites, according to Alexa's latest list of the top one million websites. This dataset includes inline scripts between <script> and </script> tags and scripts specified in the *src* attribute of <script> tags. We executed FProbe on all 969,678 unique scripts from the wild with the 62 effective browser attributes.

In summary, FProbe successfully identified browser fingerprinting behavior in 5,698 (0.59%) of all unique scripts with the high precision, recall, F-measure, and accuracy, along with an acceptable false discovery rate. It failed on 8,640 scripts, primarily due to syntax or runtime errors. Meanwhile, FProbe discovered 10,570 fingerprinting scripts from 2,851 providers on 0.78% of the 988,220 websites, with a minor overlap with the findings in Iqbal et al. (2021). We observed a noticeable number of websites, especially those related to news, hosting multiple fingerprinting scripts. Our results also indicated that some non-traditional browser fingerprinting techniques might not be as widely used as previously believed, aligning with observations in prior research

(Nikiforakis et al., 2013; Englehardt and Narayanan, 2016; Rizzo et al., 2021).

5.1. Detection accuracy

We randomly sampled 1,300 of the 969,678 unique scripts. Due to the highly imbalanced ratio of fingerprinting to non-fingerprinting scripts (8:1,292), we additionally sampled 200 scripts that were labeled as fingerprinting by FProbe. We manually examined the 1,500 sampled scripts and found 196 true positives, 12 false positives, 1289 true negatives, and 3 false negatives. The definitions of true positive, false positive, true negative, and false negative are consistent with those outlined in Section 4.

As shown in Table 2, on the 1,500 sampled scripts from the wild, FProbe achieved a 94.23% precision, suggesting that a significant portion of its identified fingerprinting behavior is accurate. It achieved a 98.49% recall, indicating its ability to identify a substantial portion of actual fingerprinting behavior in the wild. It achieved a 96.31% F-measure and a 99% accuracy with a low false positive rate of 0.92%. These results suggest the effectiveness of FProbe in the detection of browser fingerprinting.

True positives, false positives, and false discovery rate (FDR) for $Type_1$ and $Type_2$ fingerprinting among the 1,500 sampled scripts are shown in Table 3. The FDR for $Type_2$ browser fingerprinting is 25%, much higher than the FDR for $Type_1$ (2.27%). One possible reason is that $Type_2$ fingerprinting uses browser attributes in small groups or individually, resembling code behavior for rendering purposes. However, given that only 1.93% of fingerprinting scripts from Iqbal et al. (2021) and 5.74% of the sampled fingerprinting scripts from the wild fall into $Type_2$, the 25% FDR is unlikely to significantly impact FProbe's detection performance.

5.2. False positives and false negatives

The scripts of 8 false positives individually used more than 14 browser attributes for rendering purposes, misclassified as $Type_2$ fingerprinting. The scripts of other 4 false positives utilized attributes to verify specific browser properties (e.g., whether it is on an Apple device) for rendering purposes, while FProbe overestimated and falsely joined value flows of browser attributes. The causes of 3 false negatives include code obfuscation by array-based JavaScript packers, individual collection of browser attributes, and incomplete data flows due to unresolved call sites.

5.3. Dynamically loaded scripts and obfuscated scripts

The scripts of 273 (21.18%) true negatives were loading fingerprinting scripts from googletagmanager.com and google-analytics.com. We ran FProbe on their loaded scripts and FProbe detected browser fingerprinting on all of them.

The scripts of 2 true negatives and 1 false negative were obfuscated. While code obfuscation was observed in only 2% of the samples from the wild, it can significantly impact FProbe's detection performance, particularly the false negative rate. Therefore, it is strongly recommended to use FProbe with code deobfuscators.

5.4. Browser fingerprinting in the wild

On 7,737 (0.78%) of the 988,220 websites, FProbe detected 10,570 browser fingerprinting scripts; 9,963 (94.26%) of them were of $Type_1$, while 607 (5.74%) of them were of $Type_2$. It suggests that the dominant $Type_1$ browser fingerprinting demands more attention. Fig. 6 illustrates the percentages of the two types of traditional browser fingerprinting on the top N websites (in thousands). Browser fingerprinting is more frequently used on high-ranking websites. Of the 7,737 fingerprinting websites, 43.40% are among the top 10 K websites, 85.02% are among the top 20 K websites, and 94.42% are among the top 30 K websites.

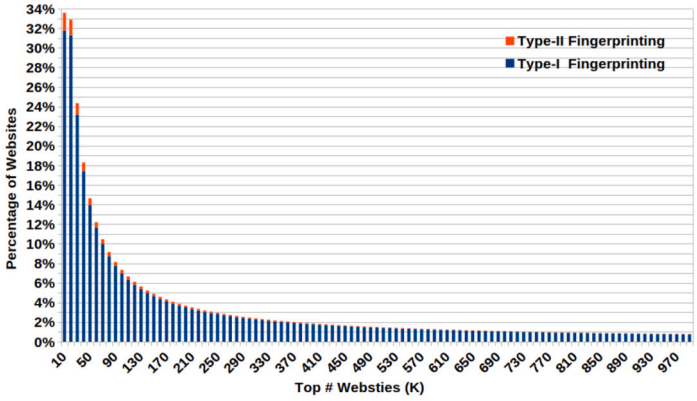

Fig. 6. Percentages of $Type_1$ and $Type_2$ browser fingerprinting among the Alexa top N websites (in thousands).

Table 5
Percentages of top 10 K, 50 K, and 100 K websites were detected with fingerprinting scripts by FProbe and FP-INSPECTOR (Iqbal et al., 2021).

	Top 10 K	Top 50 K	Top 100 K
FProbe	33.58%	14.65%	7.33%
FP-INSPECTOR (Iqbal et al., 2021)	22.76%	11.27%	10.18%

Table 5 shows the percentages of the top 10 K, 50 K, and 100 K websites where FProbe and FP-INSPECTOR (Iqbal et al., 2021) detected browser fingerprinting. Our dataset and FP-INSPECTOR's dataset shared only 3,963 fingerprinting websites and 89 fingerprinting scripts, possibly because our static crawler cannot capture dynamically loaded scripts like (Iqbal et al., 2021). Another possible reason is that Alexa's list of top one million websites and scripts on those websites have been updated since (Iqbal et al., 2021).

5.5. Browser fingerprinting scripts and providers

FProbe identified 2,851 fingerprinting providers on top one million websites compared to 519 and 2,349 providers reported in Englehardt and Narayanan (2016) and Iqbal et al. (2021). It identified 2,136 first-party fingerprinting scripts on 1,933 websites. It also identified 8,434 third-party fingerprinting scripts, from 963 third-party providers, on 6,328 first-party websites. These results, together with our observations on the dynamically loaded scripts, suggest that websites tend to deploy fingerprinting scripts offered by third parties, significantly reducing development and maintenance efforts while enabling user tracking across websites. We also observed that 45 domains served as both first-party and third-party providers for 524 websites. Half of these domains were for hosting shopping, finance, technology/internet, and news websites. Table 6 lists the domains of these providers, the number of websites they serve, and their categories.

Only 72 scripts and 10 providers (e.g., adsafeprotected.com, alicdn.com, yimg.com, and clickiocdn.com) identified by FProbe were reported in Iqbal et al. (2021). FProbe demonstrated complementary detection capabilities compared with other fingerprinting methods (Englehardt and Narayanan, 2016; Iqbal et al., 2021), attributed to its focus on a distinct aspect of browser fingerprinting behavior, i.e., the data flow joints of browser attribute values.

The left part of Table 7 presents the domains of the top 10 browser fingerprinting providers and the number of websites that have incorporated their browser fingerprinting scripts. Google has become the predominant fingerprinting provider since (Iqbal et al., 2021), potentially influenced by its initiative to block third-party cookies.

Table 6
Fingerprinting providers serving as both first-party and third-party.

Provider Domain	Served Website	Category
apple.com	6	Technology/Internet
jobstreet.com	5	Job Search/Careers
officedepot.com	5	Shopping
sony.com	5	Technology/Internet
tapatalk.com	5	Social Networking
archive.org	4	Reference
capitalone.com	4	Finance
discover.com	4	Finance
openbank.es	4	Finance
real.de	4	Shopping
southwest.com	4	Travel
tdameritrade.com	4	Brokerage/Trading
ziggo.nl	4	Technology/Internet
basecamp.com	3	Office/Business Applications
townhall.com	3	Political/Social Advocacy
aa.com	2	Travel
banorte.com	2	Finance
bbva.es	2	Finance
bestbuy.com	2	Shopping
brother-usa.com	2	Technology/Internet
castorama.fr	2	Business/Economy
diepresse.com	2	News
falabella.com	2	Shopping
glassdoor.com	2	Job Search/Careers
gq.com	2	Shopping
hotspotshield.com	2	Proxy Avoidance
imageshack.com	2	Media Sharing
imp.free.fr	2	Email
index.hu	2	News
kupivip.ru	2	Shopping
metacritic.com	2	Entertainment
nu.nl	2	News
orange.com	2	Business/Economy
oregon.gov	2	Government/Legal
pcmag.com	2	Technology/Internet
pingan.com	2	Finance
rghost.net	2	File Storage/Sharing
sap.com	2	Technology/Internet
segundamano.mx	2	Shopping
skrill.com	2	Finance
timeweb.com	2	Technology/Internet
uncrate.com	2	News
vesti.ru	2	News
worldstarhiphop.com	2	Entertainment
wunderground.com	2	Reference

Table 7
Top 10 fingerprinting providers and top 10 hosting domains of fingerprinting scripts.

Provider Domain	Websites
googletagmanager.com	2118
pagead2.googlesyndication.com	1633
s7.addthis.com	419
stats.wp.com	362
google-analytics.com	335
googleadservices.com	223
assets.adobedtm.com	162
widgets.outbrain.com	130
script.ioam.de	103
imasdk.googleapis.com	92

Hosting Domain	FP Scripts
zhcw.com	12
mrfood.com	7
silverdoctors.com	6
tolivelugu.com	6
addictinginfo.org	5
aksalser.com	5
blic.rs	5
cafef.vn	5
dantri.com.vn	5
epweike.com	5

Among the 7,737 websites where FProbe detected browser fingerprinting, 2,134 (27.58%) of them hosted at least two fingerprinting scripts, and 121 (1.56%) hosted at least four fingerprinting scripts. Among the 29 websites hosting at least five fingerprinting scripts, 15 were related to news, aligning with observations in Iqbal et al. (2021). The right part of Table 7 outlines the top 10 hosting domains and the number of hosted browser fingerprinting scripts.

5.6. EasyList and EasyPrivacy

We employed adblockparser (adblockparser, 2016) with EasyList (Easylist, 2021) and EasyPrivacy (Easyprivacy, 2021) to check the URLs of the identified fingerprinting scripts. EasyList and EasyPrivacy recognized the URLs of only 2,384 (22.55%) and 3,405 (32.21%) of the 10,570 fingerprinting script, respectively. It suggests that at least one-fifth of browser fingerprinting is used for online advertising and tracking, and browser fingerprinting is a useful indicator in their detection.

5.7. Browser attributes in fingerprinting

We examined the use of 62 effective browser attributes for fingerprinting across 7,737 websites, as detailed in Table 1. Strikingly, we discovered that approximately 50% of the websites used window.innerWidth or window.innerHeight for fingerprinting. These attributes, whose values change when the browser window is resized, are seemingly favored under the assumption that users are unlikely or reluctant to resize their browser windows. Conversely, certain attributes like navigator.oscpu and navigator.product are less frequently used. These findings underscore the maximal effort made by fingerprinting scripts to gather browser attributes.

Only 450 (4.26%) of the 10,570 fingerprinting scripts were using the toDataURL method. Similarly, only 333 (4.3%) of the 7,737 websites were using the toDataURL method. To address potential biases, we further examined all the 969,678 unique scripts from the wild and found toDataURL method calls in only 40,544 (4.18%) of them. Despite potential omissions of toDataURL method calls in dynamically loaded or obfuscated scripts due to the limitations of our evaluation, our observations strongly suggest that certain non-traditional browser fingerprinting techniques might not be as prevalent as previously assumed, consistent with prior research (Nikiforakis et al., 2013; Englehardt and Narayanan, 2016; Rizzo et al., 2021). More attention needs to be given to browser fingerprinting techniques that leverage traditional browser attributes.

5.8. Runtime performance

We measured FProbe's runtime performance on a desktop computer with an 8-core 2.6 GHz CPU, 32 GB of memory, a 64-bit Linux operating system, and Java Runtime Environment (JRE) 11. For analyzing one script, FProbe took 47 hours maximally, less than a second minimally, less than a second on the median, and 19 seconds on average. The running time is not linear with code size but is influenced by code complexity. Given FProbe's static approach, this runtime performance is considered acceptable.

The code structure below represents a simplified version of the script that consumed 47 hours in FProbe's analysis. Functions fa and fb call each other recursively. We omit stop conditions. FProbe overestimated data flows through the global variable g. These overestimated flows created a cyclic graph on which FProbe exhaustively explored all potential execution paths among lines $3 \to 4 \to 8 \to 9 \to 3$, $3 \to 4 \to 3$, and $8 \to 9 \to 8$. To address this issue, we plan to convert the cyclic graph into acyclic ones (DAGs) and then summarize transitive relations among variables within the acyclic graphs.

```
1 var g;
2 function fa() {
3   a = g;
4   g = a + ...;
5   fb();
6 }
7 function fb() {
8   b = g;
9   g = b + ...;
10  fa();
11 }
```

6. Capability of fingerprinting with traditional browser attributes

To assess the capability of fingerprinting with traditional browser attributes, we conducted a two-phase study. In the first phase, we designed a method to gather traditional browser attributes and deployed it

on a public website. We collected 3,073 unique traditional browser attributes across 3,814 visits involving 1,790 visitors. In the second phase, we fingerprinted all visitors using the collected browser attributes and achieved an impressive F-measure of 96.6%.

6.1. Collection of traditional browser attributes

For a given JavaScript object, we perform a recursive iteration of all its entries to collect traditional browser attributes. We use the entry name as the browser attribute's name. If the entry contains a primitive value, we directly extract it as the value of the browser attribute. If the entry is a (synchronous or asynchronous) function, we selectively evaluate it and obtain the returned value as the value of the browser attribute. When the entry contains another object, we recursively iterate through its entries and use the browser attributes from these entries as the value of the current browser attribute. We record the order of all iterated entries as the indexes of the collected browser attributes. Attributes obtained using advanced fingerprinting techniques (e.g., canvas, font, WebRTC, and audio-based methods) are outside the scope of our study, as they have been thoroughly studied in previous research (Acar et al., 2013; Englehardt and Narayanan, 2016; Lerner et al., 2016; Nikiforakis et al., 2013; Iqbal et al., 2021; Rizzo et al., 2021).

We excluded the following types of functions for evaluation during the attribute collection. (1) Functions do not return anything, such as window.sessionStorage.setItem() and window.dispatchEvent(). (2) Functions can disrupt browser attribute collection, such as window.alert(), window.open(), and window.close(). (3) Functions prompt users for confirmation. For instance, invoking navigator.geolocation.getCurrentPosition() will trigger a dialog asking the user to allow or decline the access to geolocation information. (4) Functions are restricted to the HTTPS protocol only. For example, navigator.geolocation.getCurrentPosition() does not return anything when invoked under the HTTP protocol. Our subsequent analysis demonstrates that our fingerprinting, even without using any values obtained from functions, can still achieve a high accuracy.

We deployed our scheme on a public website. The website randomly generates a unique identifier for every client during the first visit and saves the identifier in the cookie for associating visits from the same client. In every visit, the collected traditional browser attributes are sent back to our server together with the identifier. Nothing else was collected on our website. We eventually collected a total of 3,073 unique traditional browser attributes during 3,814 visits from 1,790 visitors. The number of browser attributes during a visit varies from 419 to 1258. The study is not human-subjected and has no privacy concern so it was exempt from the review by the Institutional Review Board.

6.2. Analysis of traditional browser attributes

Among all the 3,073 unique traditional browser attributes, 1,385 of them are with primitive values (i.e., 884 strings, 386 numbers, and 115 booleans), 1,014 of them are with values returned from functions, and 674 of them are with values derived from JavaScript objects. We divided them into 2,655 leaf attributes and 418 branch attributes. Fig. 7 shows a tree structure of selected browser attributes. Each node of the tree contains the browser attribute's index, name, type of value, and value. The node in index 23 corresponds to window.name, whose value is an empty string. The node in index 58 corresponds to navigator.javaEnabled function, which returns true. The node in index 87 corresponds to navigator.plugins object, which is an container of other objects. The node in index 656 corresponds to window.open function, which was skipped for evaluation, and its returned value is set to null.

We excluded 25 leaf attributes for browser fingerprinting because their values vary in browsing sessions from a same visitor. With the indexes, names, and values of the remaining 2,630 leaf attributes, we computed a fingerprint for every visit. We define a true positive as a re-visitor correctly recognized based on the fingerprint, a false positive as a new visitor falsely recognized as a re-visitor, a true negative as a new visitor correctly identified based on the fingerprint, and a false negative as a re-visitor falsely identified as a new visitor. To reduce false negatives caused by the evolving value of a browser attribute (e.g., the version number in navigator.userAgent), we calculated the value distance of that attribute across browsing sessions and compared the distance with a threshold during the fingerprinting.

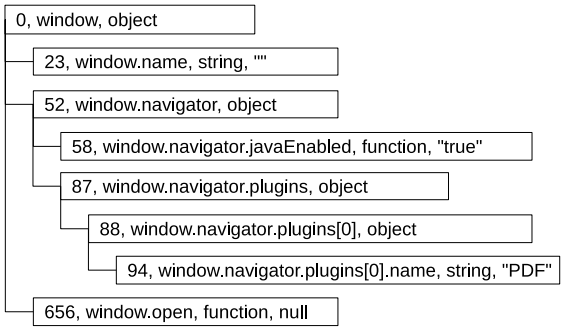

Fig. 7. An example of the tree of browser attributes.

Our fingerprinting with traditional browser attributes achieved the highest F-measure of 81% with window.navigator and an F-measure of 79.7% with the window object. The results were manually validated. We found that the indexes of leaf browser attributes play a significant role in the fingerprinting. For example, 376 possible values and 94 possible indexes can produce 574 unique attribute instances for clientInformation.userAgent. However, the indexes of branch attributes such as window.navigator do not improve the fingerprinting accuracy.

To reduce the number of attributes and simplify the scheme of their collection, while achieving a comparable performance, for every browser attribute, we calculated the distribution of all its possible values across all the browsing sessions. For attributes sharing the same distribution in terms of their values, we used only one of them for fingerprinting while excluding all the rest. The total number of traditional browser attributes for fingerprinting was then reduced to 476, while fingerprinting with the window object achieved a comparable F-measure of 80.8%.

We then performed fingerprinting with all possible combinations of the remaining 476 traditional browser attributes. With five leaf attributes only (i.e., window.innerHeight, navigator.getBattery, navigator.hardwareConcurrency, navigator.language, and navigator.userAgent), the browser fingerprinting achieved a high accuracy among 3,814 browsing sessions from 1,790 visitors with a 93.7% precision, a 99.7% recall, and a 96.6% F-measure. Note that the function navigator.getBattery was not executed during the attribute collection.

While research studies have explored the effectiveness of browser fingerprinting (Laperdrix et al., 2016; Gómez-Boix et al., 2018), our fingerprinting scheme distinguishes itself through its unique approach. It recursively iterates through JavaScript objects, additionally collects the indexes of browser attributes, and exclusively focuses on traditional browser attributes in fingerprinting. Despite these differences, fingerprinting with traditional browser attributes proves to be as effective as other browser fingerprinting techniques (e.g., canvas, font, WebRTC, audio, and even machine learning-based methods) and has been widely observed among fingerprinting scripts in our evaluation (Section 5). As a result, more attention needs to be given to browser fingerprinting techniques leveraging traditional browser attributes.

6.3. Browser fingerprinting capability

For all the websites with fingerprinting scripts discovered in Section 5, we estimated their fingerprinting capabilities with traditional browser attributes. The maximum, minimum, average, and median F-measure values for a website's fingerprinting capability were 99.06%,

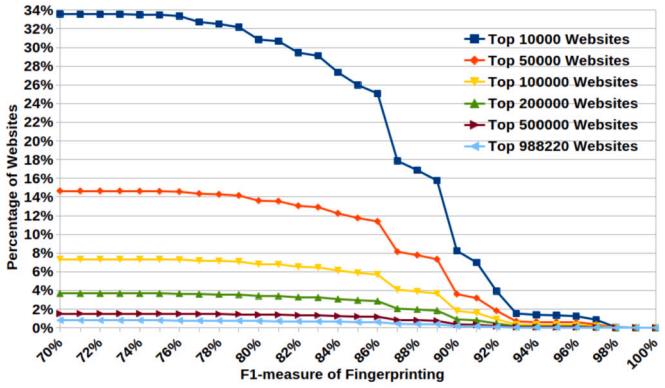

Fig. 8. Percentage of Alexa top N websites versus F-measure of browser fingerprinting.

70.59%, 87.59%, and 88.42%, respectively. Impressively, 90.6% of fingerprinting websites achieved an 80% F-measure, while 23.6% reached a 90% F-measure. Fig. 8 illustrates that fingerprinting with traditional browser attributes tends to be more effective among top-ranked websites.

7. Discussion

FProbe's performance is influenced by false and missing edges in call graphs (Antal et al., 2018), leading to false positives and false negatives. However, call graph construction is not our focus. Please refer to Antal et al. (2018) for discussions on various approaches regarding call graph construction for JavaScript programs.

FProbe suffers from false positives due to the overestimated code execution paths, especially those involving conditional statements, lexical/global variables, event/message handlers, and object fields, in comparison to dynamic (taint) analysis for detecting browser fingerprinting (Sjösten et al., 2021). (1) When it is impossible to resolve a conditional expression, false dependencies could be introduced from variables in the expression to those used in the condition's code blocks. (2) Excessive dependencies are created from the definition of a global/lexical variable to all its potential subsequent uses during our propagation and data flow analysis. (3) When it is impossible to resolve the event name at an event dispatcher, false dependencies could be introduced from the dispatchers to all their potential handlers. (4) When it is impossible to resolve the name of an object's field or the index of an array element, the propagation and flow analysis are performed on the entire object or array. However, the static analysis in FProbe avoids the incomplete coverage of code execution paths and the performance overhead often incurred in dynamic analysis (Bandhakavi et al., 2010; Dhawan and Ganapathy, 2009; Sabelfeld and Myers, 2003; Vogt et al., 2007; Xu et al., 2013).

False negatives may occur due to several reasons. (1) FProbe has limited capability in detecting browser fingerprinting in obfuscated scripts, as discussed in Section 4.4. Therefore, we recommend FProbe to be used with JavaScript deobfuscator to address this limitation to a certain extent. It is worth mentioning that FProbe can effectively analyze minified JavaScript files, where comments and unnecessary whitespaces are removed to reduce file size (Xu et al., 2013). (2) As it is impossible to achieve soundness in our propagation and flow analysis, browser attributes and their value flows may be omitted or incomplete. Consequently, some browser fingerprinting behavior may be overlooked. (3) FProbe cannot identify browser fingerprinting dispersed across scripts, similar to prior work (Iqbal et al., 2021; Su and Kapravelos, 2023). (4) FProbe cannot detect browser fingerprinting utilizing APIs not listed in Table 1. However, this limitation can be addressed by enhancing the API list.

We plan to (1) support the detection of other types of browser fingerprinting techniques such as those based on fonts, WebRTC, and audio, (2) integrate dynamic analysis techniques and address its performance issues, (3) reduce false positives and false negatives caused by false or missing data dependencies, (4) add modules for deobfuscating JavaScript code, and (5) improve browser fingerprinting classification using machine learning models. However, given the inherent nature of static and dynamic analyses, our tool aims to be practical and accurate rather than sound.

8. Related work

8.1. Browser fingerprinting detection

Only a few research studies targeted browser fingerprinting detection. In Nikiforakis et al. (2013), Nikiforakis et al. identified the prevalent use of flash objects in third-party commercial fingerprinting scripts. Analyzing JavaScript inclusion on the top 10,000 websites, they found that 40 websites included flash-based fingerprinting scripts. FPDetective was proposed in Acar et al. (2013). It drives an instrumented browser and monitors fingerprinting-related API calls during web browsing. It was evaluated on the top one million websites and identified 16 new fingerprinting-related scripts and flash objects. OpenWPM was introduced in Englehardt and Narayanan (2016) to measure both stateless and stateful web tracking techniques on the top one million websites. During web browsing, it simulates user behavior, records HTTP responses and cookies, and intercepts the use of specific JavaScript objects and APIs. OpenWPM identified canvas-based fingerprinting on 1.6% of the websites and font-based fingerprinting on less than 1% of the websites. It also identified fingerprinting techniques relying on WebRTC, AudioContext, and battery status. In Lerner et al. (2016), by examining the use of JavaScript APIs, Lernel et al. analyzed web tracking techniques (i.e., cookie-based web tracking and browser fingerprinting) on web page archives from 1996 to 2016. The results highlighted the increasing use of JavaScript-based fingerprinting over time.

Researchers have recently introduced machine learning-based approaches for detecting browser fingerprinting (Iqbal et al., 2021; Rizzo et al., 2021; Bahrami et al., 2022). In Iqbal et al. (2021), they trained a classifier with static and dynamic features extracted from the selected fingerprinting and non-fingerprinting scripts. The static features are derived from the abstract syntax tree of the source code. The dynamic features include the invoked JavaScript objects and APIs and JavaScript execution traces collected during web browsing. They detected browser fingerprinting behavior on 10.18% of Alexa top 100 K websites. The detection model achieved an F-measure of 89% with static features only, 97.9% with dynamic features only, and 93.4% with both static and dynamic features. Similar approaches have been proposed in Rizzo et al. (2021); Bahrami et al. (2022).

These heuristic approaches (Iqbal et al., 2021; Rizzo et al., 2021; Bahrami et al., 2022) and other studies (Acar et al., 2013; Nikiforakis et al., 2013; Englehardt and Narayanan, 2016; Lerner et al., 2016) concentrated on JavaScript objects and APIs invoked to retrieve browser attribute values. However, they overlooked the use and flow of browser attributes within JavaScript code, as emphasized in Lerner et al. (2016). In contrast, FProbe focuses on how browser attributes are aggregated in scripts for fingerprinting, since the value flows of browser attributes is crucial for browser fingerprinting detection, aligning with the definition of browser fingerprinting behavior. To the best of our knowledge, this is the first work that performs static data flow analysis to detect browser fingerprinting, and FProbe is complementary to existing fingerprinting detection methods.

Researchers have recently introduced EssentialFP (Sjösten et al., 2021), which performs dynamic analysis for browser fingerprinting detection. Dynamic analysis can explore code execution paths more precisely with runtime information. However, it often suffers incomplete coverage of code execution paths (Sabelfeld and Myers, 2003; Vogt et al., 2007) and non-negligible performance overhead (Bandhakavi et al., 2010; Sjösten et al., 2021). In our recent investigations, we observed

that the performance overhead of dynamic taint analysis in browsers could lead to the freezing of browsing tabs, preventing the loading of the scripts to be analyzed, a phenomenon also highlighted in Sjösten et al. (2021). Additionally, dynamic analysis often requires software-specific instrumentation (Dhawan and Ganapathy, 2009).

Static and dynamic analyses have been combined to uncover emerging browser fingerprinting techniques (Su and Kapravelos, 2023). They first execute scripts to gather the sequence of API executions and then employ static data flow analysis to identify emerging APIs that are concurrently executed with well-known browser fingerprinting APIs. However, this approach suffers from the limitations of dynamic analysis, as discussed above.

Although static code analysis suffers relatively low accuracy on predicting code execution paths and runtime values compared to dynamic analysis, FProbe demonstrated excellent performance in our evaluation. Importantly, it minimizes the risk of incomplete code path coverage and performance overhead (Bandhakavi et al., 2010; Dhawan and Ganapathy, 2009; Sabelfeld and Myers, 2003; Vogt et al., 2007; Xu et al., 2013), aligning with our design objective - a practical and accurate solution for maximally detecting browser fingerprinting. In contrast to tools like OpenWPM (Openwpm, 2021) used in Englehardt and Narayanan (2016); Iqbal et al. (2021) and JSFlow Jsflow (2021) used in Sjösten et al. (2021), FProbe does not require any browser-specific instrumentation. While FProbe has limited capability for obfuscated scripts, we believe it can be partially addressed by integrating JavaScript deobfuscators.

8.2. JavaScript code analysis

Research on the security analysis of JavaScript programs mainly focused on web attack and vulnerability detection. Different from them, FProbe focuses on a different research problem. It aims to be a practical and accurate tool for detecting browser fingerprinting.

Researchers have performed the static analysis of JavaScript to detect malware (Chugh et al., 2009; Curtsinger et al., 2011; Laskov and Šrndić, 2011), to explore vulnerabilities (e.g., injection and cross-site scripting) in web applications (Guarnieri et al., 2011; Guha et al., 2009; Jovanovic et al., 2006; Livshits and Lam, 2005; Wassermann and Su, 2008) and browser extensions (Bandhakavi et al., 2010), to protect applications (Huang et al., 2004), and to verify JavaScript behavior (Taly et al., 2011).

Dynamic analysis techniques have also been used to detect malicious JavaScript code (Cova et al., 2010; Livshits and Cui, 2008; Provos et al., 2007; Yin et al., 2007), to explore vulnerabilities in web applications (Saxena et al., 2010; Tripp et al., 2014), to identify information flows that violate privacy policies (e.g., cookie stealing and history sniffing) (Djeric and Goel, 2010; Dhawan and Ganapathy, 2009; Jang et al., 2010; Yin et al., 2007), and to enforce information flow security in JavaScript (Hedin and Sabelfeld, 2012; Zeng et al., 2010).

Static and dynamic analysis techniques have been combined to prevent cross-site scripting attacks (Tripp et al., 2014; Vogt et al., 2007), and to track information flow and injected code in web attacks (Just et al., 2011; Wei and Ryder, 2013).

Compared with existing tools for static data flow and taint analyses, FProbe is lightweight in terms of value and prototype propagation and data flow analysis. It focuses on values, variables, and flows associated with browser attributes, in contrast to others focusing on the entire program. Furthermore, the propagation of JavaScript values and prototypes improves the hit rate of source variable identification. FProbe computes data flow joints to identify the aggregation of browser attribute values, a feature that, to the best of our knowledge, has not yet been leveraged by other tools.

9. Conclusion

In this paper, we defined browser fingerprinting behavior as the aggregation of various browser attributes, and reduced browser fingerprinting detection to a joint analysis of the data flows of browser attribute values in JavaScript code. We introduced a flow-centric browser fingerprinting detection framework, FProbe, which performs context-sensitive static data flow analysis of JavaScript code. We evaluated FProbe using 4,296 fingerprinting scripts from recent work and 2,335,317 pieces of JavaScript code from 988,220 websites. FProbe achieved F-measures of 97.81% and 96.31% on these datasets, respectively. It identified browser fingerprinting behavior on 0.78% of the 988,220 websites. Only 72 browser fingerprinting scripts and 10 browser fingerprinting providers identified by FProbe were reported in prior research. These results demonstrate the effectiveness of FProbe in detecting browser fingerprinting. Additionally, we conducted a study to assess the capability of fingerprinting with traditional browser attributes.

CRediT authorship contribution statement

Rui Zhao: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

We thank the reviewers for their constructive comments to help us improve the paper. This work was supported by the University of Nebraska System Support Grant No. 20374.

References

Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., Preneel, B., 2013. Fpdetective: dusting the web for fingerprinters. In: Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, pp. 1129–1140.

Acar, G., Eubank, C., Englehardt, S., Juarez, M., Narayanan, A., Diaz, C., 2014. The web never forgets: persistent tracking mechanisms in the wild. In: Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, pp. 674–689.

adblockparser, 2016. https://pypi.org/project/adblockparser/.

Antal, G., Hegedus, P., Tóth, Z., Ferenc, R., Gyimóthy, T., 2018. Static javascript call graphs: a comparative study. In: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 177–186.

Bahrami, P.N., Iqbal, U., Shafiq Fp-radar, Z., 2022. Longitudinal measurement and early detection of browser fingerprinting. In: Proceedings on Privacy Enhancing Technologies 2022, vol. 2, pp. 557–577.

Bandhakavi, S., King, S.T., Madhusudan, P., Winslett, M., 2010. Vex: vetting browser extensions for security vulnerabilities. In: Proceedings of the USENIX Security Symposium, pp. 339–354.

Browserize - the browser characterization library. https://privacycheck.sec.lrz.de/.

Cao, Y., Li, S., Wijmans, E., 2017. (Cross-)browser fingerprinting via OS and hardware level features. In: Proceedings of the Annual Network and Distributed System Security Symposium.

Chugh, R., Meister, J.A., Jhala, R., Lerner, S., 2009. Staged information flow for javascript. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, pp. 50–62.

Closure Compiler, 2021a. https://developers.google.com/closure/compiler/.

Cova, M., Kruegel, C., Vigna, G., 2010. Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the International World Wide Web Conference, pp. 281–290.

Curtsinger, C., Livshits, B., Zorn, B.G., Seifert, C., 2011. Zozzle: fast and precise in-browser javascript malware detection. In: Proceedings of the USENIX Security Symposium, pp. 33–48.
Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K., 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13 (4), 451–490.
Dhawan, M., Ganapathy, V., 2009. Analyzing information flow in javascript-based browser extensions. In: Proceedings of the Annual Computer Security Applications Conference, pp. 382–391.
Djeric, V., Goel, A., 2010. Securing script-based extensibility in web browsers. In: Proceedings of the USENIX Security Symposium.
Easylist, 2021. https://easylist.to/easylist/easylist.txt.
Easyprivacy, 2021. https://easylist.to/easylist/easyprivacy.txt.
Englehardt, S., Narayanan, A., 2016. Online tracking: a 1-million-site measurement and analysis. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 1388–1401.
Fifield, D., Egelman, S., 2015. Fingerprinting web users through font metrics. In: International Conference on Financial Cryptography and Data Security, pp. 107–124.
Gómez-Boix, A., Laperdrix, P., Baudry, B., 2018. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In: Proceedings of the World Wide Web Conference, pp. 309–318.
Grove, D., DeFouw, G., Dean, J., Chambers, C., 1997. Call graph construction in object-oriented languages. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 108–124.
Guarnieri, S., Livshits Gatekeeper, B., 2009. Mostly static enforcement of security and reliability policies for javascript code. In: Proceedings of the USENIX Security Symposium, pp. 151–168.
Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R., 2011. Saving the world wide web from vulnerable javascript. In: Proceedings of the International Symposium on Software Testing and Analysis. ACM, pp. 177–187.
Guha, A., Krishnamurthi, S., Jim, T., 2009. Using static analysis for ajax intrusion detection. In: Proceedings of the International Conference on World Wide Web. ACM, pp. 561–570.
Hedin, D., Sabelfeld, A., 2012. Information-flow security for a core of javascript. In: Proceedings of the IEEE Computer Security Foundations Symposium. IEEE Computer Society, pp. 3–18.
Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., Kuo, S.-Y., 2004. Securing web application code by static analysis and runtime protection. In: Proceedings of the International World Wide Web Conference, pp. 40–52.
Iqbal, U., Englehardt, S., Shafiq, Z., 2021. Fingerprinting the fingerprinters: learning to detect browser fingerprinting behaviors. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 1143–1161.
Jang, D., Jhala, R., Lerner, S., Shacham, H., 2010. An empirical study of privacy-violating information flows in javascript web applications. In: Proceedings of the ACM Conference on Computer and Communications Security. ACM, pp. 270–283.
Jovanovic, N., Kruegel, C., Kirda, E., 2006. Pixy: a static analysis tool for detecting web application vulnerabilities. In: Proceedings of the IEEE Symposium on Security and Privacy.
Jsflow, 2021. https://www.jsflow.net/.
Just, S., Cleary, A., Shirley, B., Hammer, C., 2011. Information flow analysis for javascript. In: Proceedings of the ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, pp. 9–18.
Laperdrix, P., Rudametkin, W., Baudry, B., 2016. Beauty and the beast: diverting modern web browsers to build unique browser fingerprints. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 878–894.
Laskov, P., Šrndić, N., 2011. Static detection of malicious javascript-bearing pdf documents. In: Proceedings of the Annual Computer Security Applications Conference, pp. 373–382.
Lerner, A., Simpson, A.K., Kohno, T., Roesner, F., 2016. Internet Jones and the raiders of the lost trackers: an archaeological study of web tracking from 1996 to 2016. In: Proceedings of the USENIX Security Symposium.
Libert, T., 2015. Exposing the invisible web: An analysis of third-party http requests on 1 million websites. arXiv preprint. arXiv:1511.00619.
Livshits, B., Cui, W., 2008. Spectator: detection and containment of javascript worms. In: Proceedings of the USENIX Annual Technical Conference.
Livshits, B., Lam, M.S., 2005. Finding security vulnerabilities in Java applications with static analysis. In: Proceedings of the USENIX Security Symposium.
Madsen, M., Livshits, B., Fanning, M., 2013. Practical static analysis of javascript applications in the presence of frameworks and libraries. In: Proceedings of the Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 499–509.
Mayer, J.R., Mitchell, J.C., 2012. Third-party web tracking: policy and technology. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 413–427.
Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G., 2012. You are what you include: large-scale evaluation of remote javascript inclusions. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 736–747.
Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., Vigna, G., 2013. Cookieless monster: exploring the ecosystem of web-based device fingerprinting. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 541–555.
Nikiforakis, N., Joosen, W., Livshits, B., 2015. Privaricator: deceiving fingerprinters with little white lies. In: Proceedings of the International Conference on World Wide Web, pp. 820–830.
Openwpm, 2021. https://github.com/openwpm/OpenWPM/.
Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N., 2007. The ghost in the browser analysis of web-based malware. In: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets.
Richards, G., Lebresne, S., Burg, B., Vitek, J., 2010. An analysis of the dynamic behavior of javascript programs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 1–12.
Richards, G., Hammer, C., Burg, B., Vitek, J., 2011. The eval that men do - a large-scale study of the use of eval in javascript applications. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), pp. 52–78.
Rizzo, V., Traverso, S., Mellia, M., 2021. Unveiling web fingerprinting in the wild via code mining and machine learning. In: Proceedings on Privacy Enhancing Technologies, pp. 43–63.
Sabelfeld, A., Myers, A.C., 2003. Language-based information-flow security. IEEE J. Sel. Areas Commun. 21 (1), 5–19.
Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D., 2010. A symbolic execution framework for javascript. In: Proceedings of the IEEE Symposium on Security and Privacy. IEEE Computer Society, pp. 513–528.
Schwarz, M., Lackner, F., Gruss, D., 2019. Javascript template attacks: automatically inferring host information for targeted exploits. In: Proceedings of Network and Distributed System Security Symposium.
Sjösten, A., Hedin, D., Sabelfeld, A., 2021. Essentialfp: exposing the essence of browser fingerprinting. In: Proceedings of the IEEE European Symposium on Security and Privacy Workshops, pp. 32–48.
Su, J., Kapravelos, A., 2023. Automatic discovery of emerging browser fingerprinting techniques. In: Proceedings of the ACM Web Conference, pp. 2178–2188.
Taly, A., Erlingsson, U., Mitchell, J.C., Miller, M.S., Nagra, J., 2011. Automated analysis of security-critical JavaScript APIs. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 363–378.
Torres, C.F., Jonker, H., Mauw, S., 2015. Fp-block: usable web privacy by controlling browser fingerprinting. In: Proceedings of the European Symposium on Research in Computer Security, pp. 3–19.
Tripp, O., Ferrara, P., Pistoia, M., 2014. Hybrid security analysis of web javascript code via dynamic partial evaluation. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 49–59.
Vastel, A., Laperdrix, P., Rudametkin, W., Rouvoy, R., 2018. Fp-stalker: tracking browser fingerprint evolutions. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 1–14.
Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G., 2007. Cross site scripting prevention with dynamic data tainting and static analysis. In: Proceedings of Network and Distributed System Security Symposium.
WALA Compiler, 2021b. http://wala.sourceforge.net/wiki/index.php/Main_Page.
Wassermann, G., Su, Z., 2008. Static detection of cross-site scripting vulnerabilities. In: Proceedings of the ICSE Workshop on Dynamic Analysis, pp. 171–180.
Wei, S., Ryder, B.G., 2013. Practical blended taint analysis for javascript. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 336–346.
Weihl, W.E., 1980. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 83–94.
Xu, W., Zhang, F., Zhu, S., 2013. Jstill: mostly static detection of obfuscated malicious javascript code. In: Proceedings of the ACM Conference on Data and Application Security and Privacy, pp. 117–128.
Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E., 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the ACM Conference on Computer and Communications Security. ACM, pp. 116–127.
Yue, C., Wang, H., 2013. A measurement study of insecure javascript practices on the web. ACM Trans. Web 7 (2), 7.
Zeng, P., Sun, J., Chen, H., 2010. Insecure javascript detection and analysis with browser-enforced embedded rules. In: Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 393–398.
Zhao, R., Yue, C., Yi, Q., 2015. Automatic detection of information leakage vulnerabilities in browser extensions. In: Proceedings of the International Conference on World Wide Web, pp. 1384–1394.

Rui Zhao received the Ph.D. degree in computer science from the Colorado School of Mines, Golden, CO, USA, in 2016. He is currently an Assistant Professor with the School of Interdisciplinary Informatics, University of Nebraska at Omaha, Omaha, NE, USA. His research interests include web security and privacy, and system security. He is a member of ACM.