Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 207 (2022) 867–876

Procedia Computer Science

www.elsevier.com/locate/procedia

26th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2022)

Looking for Criminal Intents in JavaScript Obfuscated Code

Federico Cerutti*, Daniele Barattieri di San Pietro, Francesco Gringoli, Gianfranco Lamperti

Department of Information Engineering, University of Brescia, 25123 Brescia, Italy

Abstract

The majority of websites incorporate *JavaScript* for client-side execution in a supposedly protected environment. Unfortunately, *JavaScript* has also proven to be a critical attack vector for both independent and state-sponsored groups of hackers. On the one hand, defenders need to analyze scripts to ensure that no threat is delivered and to respond to potential security incidents. On the other, attackers aim to obfuscate the source code in order to disorient the defenders or even to make code analysis practically impossible. Since code obfuscation may also be adopted by companies for legitimate intellectual-property protection, a dilemma remains on whether a script is harmless or malignant, if not criminal. To help analysts deal with such a dilemma, a methodology is proposed, called *JACOB*, which is based on five steps, namely: (1) *source code parsing*, (2) *control flow graph recovery*, (3) *region identification*, (4) *code structuring*, and (5) *partial evaluation*. These steps implement a sort of decompilation for control flow flattened code, which is progressively transformed into something that is close to the original *JavaScript* source, thereby making eventual code analysis possible. Most relevantly, *JACOB* has been successfully applied to uncover unwanted user tracking and fingerprinting in e-commerce websites operated by a well-known Chinese company.

© 2022 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0)
Peer-review under responsibility of the scientific committee of the 26th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2022)

Keywords: JavaScript; web; cybercrime; code obfuscation; code deobfuscation; control flow flattening; reverse engineering; e-commerce; user tracking; fingerprinting

1. Introduction

JavaScript is a high-level programming language conforming to the *ECMAScript* standard. Designed in 1993 as a scripting language for the *Netscape Navigator* browser, its original purpose was to enrich webpages with minimal client-side scripting features, such as form validation. The role of *JavaScript* remained minor until *AJAX* became the standard for dynamic webpages. Afterwards, support for *just-in-time* compilation in Google's *V8* engine contributed to a dramatic performance improvement, thereby paving the way to *JavaScript* further. While *JavaScript* engines implement a protected environment, new vulnerabilities are constantly found in web browsers [3, 11, 14, 20]. Advanced time-dependent attacks have also been ported to the web [8, 17], as well as more traditional attacks such as formjackers

* Federico Cerutti. Tel.: +39-334-702-7876
E-mail address: federico@ceres-c.it

1877-0509 © 2022 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0)
Peer-review under responsibility of the scientific committee of the 26th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2022)
10.1016/j.procs.2022.09.142

(credit card skimming) [5]. In these scenarios, *JavaScript* has proven to be a critical attack vector for both independent and state-sponsored groups. Consequently, the defenders need to analyze scripts to ensure that no threat is delivered and to respond to potential security incidents. In principle, as the standard delivery form of web applications is source code, *JavaScript* analysis should be less labor-intensive than binary reverse engineering. Unfortunately, *code obfuscation*, the operation of generating (either manually or automatically) purposefully unreadable code, can prevent an analyst from understanding the behavior of a piece of code. Although obfuscation may be legitimately adopted by companies for source code protection, it is often pursued to hide malicious (and possibly criminal) software, thereby making it difficult to distinguish legitimate intellectual property protection from criminal intents without in-depth analysis. To help analysts deal with such a dilemma, a methodology for deobfuscation of potentially threatening *JavaScript* code is presented in this paper, called *JACOB* (*JAvascript COde Bulldozer*). Most notably, *JACOB* has been successfully exercised to uncover unwanted user tracking and fingerprinting in a pool of e-commerce websites operated by a well-known Chinese company.[1]

2. Code Obfuscation

JavaScript code is omnipresent and often distributed as a plain text `.js` file, inducing a recurring need for developers to find a way to protect the source code. To meet this demand, multiple open source and commercial obfuscation tools exist for *JavaScript*; automated code obfuscators rely on established techniques [6, 13], among which:

- *Control logic obfuscation*. The logic ruling branching and decision-making in the code is obfuscated by different techniques: (*a*) *control flow flattening*, where the control flow does not develop in depth, because every basic block[2] is redirected to a main dispatcher, which steers execution from one basic block to the subsequent (cf. Figure 1); (*b*) *bogus control flow*, where useless control flow paths that cannot possibly be followed are deliberately inserted into the code, thereby undermining a purely static analysis, owing to a possibly overwhelming magnified code; the complexity can be increased even further, via *opaque predicates*, which may prevent both software and humans from inferring the actual execution flow; and (*c*) *probabilistic execution*, where blocks with different syntax, but equivalent behavior, are executed in an apparently arbitrary way.
- *Layout obfuscation*. The layout of the code is obfuscated while retaining the original syntax: (*a*) *identifier replacement*, where original variable identifiers are replaced with meaningless names such as one-letter identifiers or purposefully confusing names (characters like `O`, `o`, `0`); to generate even more confusion, identifiers may be reused in different scopes; (*b*) *dead code injection*, where useless statements are inserted; and (*c*) *symbol stripping*, where unnecessary symbols, including debugging information, are removed.[3]
- *Data obfuscation*. The referenced data is obfuscated: (*a*) *data encoding*, where data is encoded with a compression or encryption algorithm and decoded at runtime; (*b*) *data splitting*, where contiguous data is split and relocated appropriately in order to appear scrambled; and (*c*) *literal to function-call conversion*, where literals are replaced with calls to a function that returns the literal, given some specific arguments.
- *Anti debugging*, where the software, noticing it is being analyzed with a debugging tool, refuses to execute or takes specific code paths hiding relevant information.

The actual aim of code obfuscation, however, be it harmless or malignant, remains unknown until the obfuscated code is manipulated to aid comprehension. Since deobfuscation is generally hard to achieve manually, an automatic technique for deobfuscating *JavaScript* code has been designed and implemented, based on a methodology outlined in Section 3.

[1] Specifically, the *Alibaba Group* company, operating (among others) the *Alibaba*, *AliExpress*, *Taobao*, and *Tmall* websites.

[2] A *basic block* is a maximal sequence of statements with a single entry and a single exit such that no statement inside the sequence is the destination of a jump and only the last statement can cause the program to execute code in a different basic block.

[3] This approach is commonly used by the *JavaScript* developers to shrink the size of files pushed to users, thereby reducing download times and improving user experience.

3. *JACOB*: A Methodology for Deobfuscating JavaScript Code

JACOB has been specifically conceived to cope with control flow flattening obfuscation by regenerating control structures, such as `if-then`, `if-else`, and `while` loops. To this end, the control flow graph (CFG)[4] is restructured, by aggregating basic blocks and enriching data to infer branching logic, while retaining semantic equivalence, a task generally performed by binary decompilers.

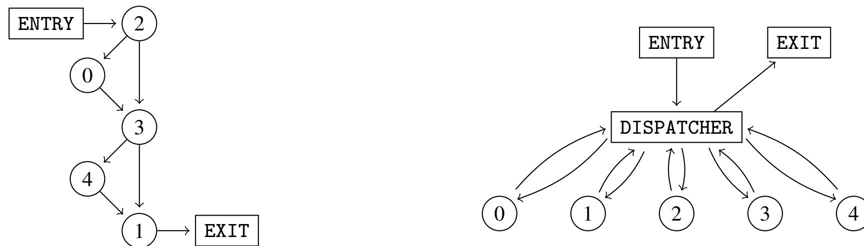


Figure 1: A normal control flow graph (left), and its flattened equivalent (right).

As a result of obfuscation by control flow flattening, all the basic blocks are arranged at the same level, with the execution flow being managed by a *dispatcher* node, as shown in Figure 1.[5] While jump-like statements may seem mandatory to implement control flow flattening, their effects can be emulated in languages lacking them, such as *JavaScript*, with elaborate solutions such as the one described in Section 4.3. *JACOB* consists of five steps, namely (1) *source code parsing*, (2) *control flow graph recovery*, (3) *region identification*, (4) *code structuring*, and (5) *partial evaluation*, which are outlined below.

3.1. Source code parsing

The first step is to identify code sections implementing application logic, ignoring ancillary structures introduced by the obfuscation process. In many languages, the execution flow can be redirected at will via jump-like statements. Since *JavaScript* lacks similar unconditional branching instructions, they need to be emulated by the authors of the obfuscator. Of course, thanks to the flexibility of the language, many solutions can be designed to achieve this goal, and implementations can drastically vary among different obfuscators, requiring ad-hoc parsing.

This phase aims to create a standardized interface allowing the subsequent steps to easily retrieve basic blocks as abstract syntax tree chunks which can be easily queried and traversed, while retaining important information about code structure.

3.2. Control flow graph recovery

The control flow graph of a control-flow flattened program is *flat* (cf. Figure 1, right), and the execution flow returns to the dispatcher after every basic block. To allow further analysis and code reconstruction, the original CFG must be recovered, restoring direct edges between nodes and removing the intermediary dispatcher. This process is called *unflattening* and is described in multiple sources [10, 16, 21].

The dispatcher is connected to every node, but has no knowledge about the application logic: determining how execution shall proceed is demanded to basic blocks. Once the specific dispatcher implementation has been understood, it is possible to reconnect consequent basic blocks. From a practical standpoint, this process could be as simple as tracking write accesses to a specific variable which controls the execution flow (e.g. a pseudo *Program Counter*). Or it

[4] A Control Flow Graph is a directed graph where nodes are basic blocks, which represents all the paths that might be traversed during the execution of the program.

[5] While not detrimental, is would not be useful to apply *JACOB* to unflattened control flow code, since there is no point in recovering control structures that were not lost.

might even be necessary to deal with some kind of dynamic execution routing which cannot be inferred statically, thus requiring a theorem prover[6] to identify the next successor.

3.3. Region identification

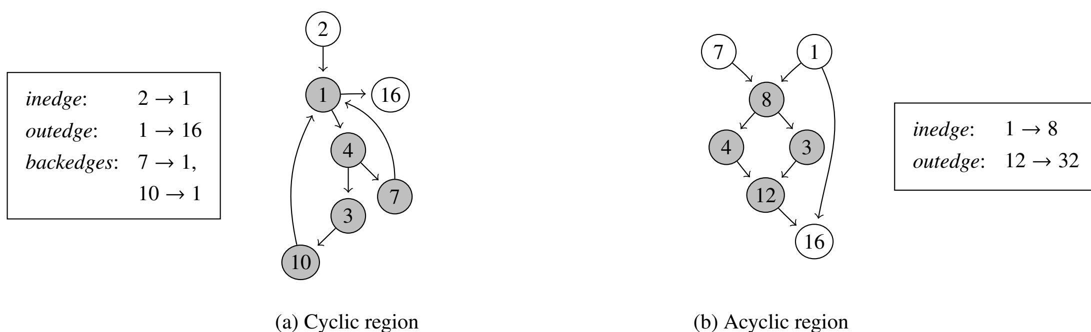
JavaScript is a block-structured programming language, denoting that control structures rely on source code being organized in lexical *blocks*. Considering an `if-else` structure, each of the two branches is delimited by separators defining a block (curly brackets). The separators identify execution boundaries: statements in a block are executed until the block is consumed.

In source code, *blocks* are explicitly organized hierarchically; for example, an `if-then` can be nested within a `while` loop. On the other hand, *basic blocks* are minimal structures which can not contain, per definition, other basic blocks: this means they can not be organized hierarchically.

When going from a structured to an unstructured code representation (like when compiling C source into Assembly code), the hierarchical structure is lost. Nonetheless, the CFG of the unstructured program still contains, implicitly, information on the original block hierarchy. To recover the original control structures, basic blocks are gathered into groups, called *regions*, pertaining to any of the three classes proposed in [2]:

- Iteration
- Selection
- Sequence

JACOB takes into account only two types of regions: *cyclic* (iteration) and *acyclic* (selection, sequence), as exemplified in Figure 2.



(a) Cyclic region (b) Acyclic region

Figure 2: Different region types. Gray nodes denote a region

A *single-entry, single-exit* property is enforced, that is all the edges entering a region (*inedges*) must end at the same node *inside* the region and all the edges leaving a region (*outedges*) must end on the same node *outside* the region. Based on this property, once a region is identified, it can be extracted from the CFG and replaced with a single node.

3.3.1. Cyclic Region

Loops are control structures designed to iterate multiple times over a section of code. The content of a loop (its body) is executed until a given testing condition becomes false, then the program can proceed. In a *structured loop*, as defined in most programming languages, an iteration can start only at the first statement of the body, while exiting the loop moves the execution to the first statement after the loop. An iteration can be interrupted prematurely, either returning to the first statement of the body (`continue`) or exiting the loop (`break`).

In a CFG, loops are strongly connected components[7] of the graph, and can be identified by *backedges*. Backedges are fundamental to loops because, as the name suggests, they allow execution to return to a previous node. The last

[6] A theorem prover is a software built to automatically prove mathematical theorems. Theorem provers are often used by reverse engineering tools (including *angr*, the *Miasm*, and *ExpoSE*) to evaluate reaching conditions of a node in a CFG.

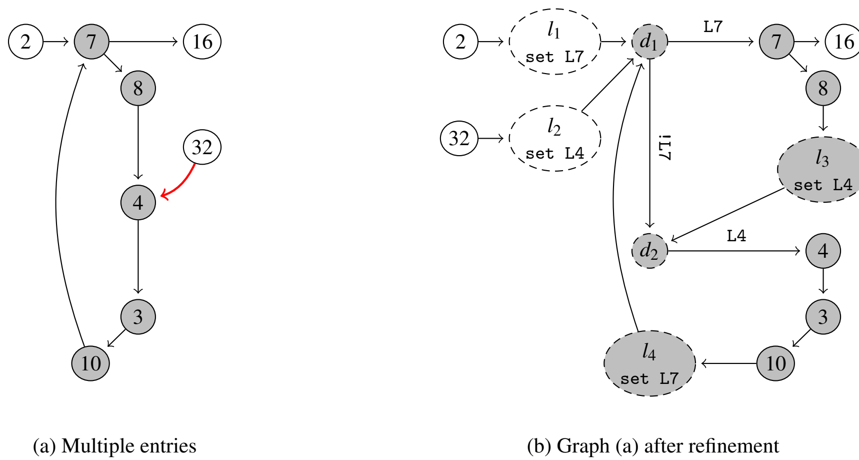
[7] A directed graph is strongly connected if every vertex is reachable from every other vertex. The strongly connected components of a graph are the parts of the graph fulfilling the strong connection property.

basic block in the loop is then required to have a backedge, `continue` statements will result in additional backedges, and `break` statements generate more outedges.

Abnormal Cyclic Region. As mentioned above, source code compilation results in loss of explicit information on hierarchical code organization; loops, for example, are translated into Assembly jump statements. Thanks to their high flexibility, jumps also allow the execution to enter loops at different points of the body, as well as to leave via different exits. However, this behavior poses a challenge while trying to recover control structures since there is no direct correspondent to such a control flow in structured loops. Decompilers of binary programs often overcome this issue by generating *pseudo-C* code with `goto` statements.

JavaScript control flow flattened code might include non-standard loops as well. Considering that *JavaScript* has no jump-like statements, this issue can't be ignored. Preserving code behavior requires *JACOB* to implement a sophisticated `goto`-free solution. This problem is solved by a labelling technique: new nodes are added both inside and outside the cyclic region to set virtual labels and steer the control flow appropriately. Strategically placing these nodes in the graph allows enforcing the *single-entry, single-exit* property of regions, while retaining the semantics.[8]

Abnormal Entry. A loop is said to have abnormal entries when at least two of its nodes are target of an entry edge; in this case the first iteration could bypass part of the loop body. With reference to Figure 3a, if the loop is entered through node 32, nodes 7 and 8 are skipped at first, and executed only on the second iteration. As shown in Figure 3b, all the entries are moved to a newly created decision node *inside* the loop (d_1), which controls execution and, based on previously set labels, decides where the loop should start. After this change, the region has only one entry node, so an equivalent structured loop exists, but it also retains the possibility to execute only part of the nodes on the first iteration. The second iteration will correctly execute all the nodes of the CFG. An indefinite number of abnormal entries can be addressed by chaining multiple decision nodes.



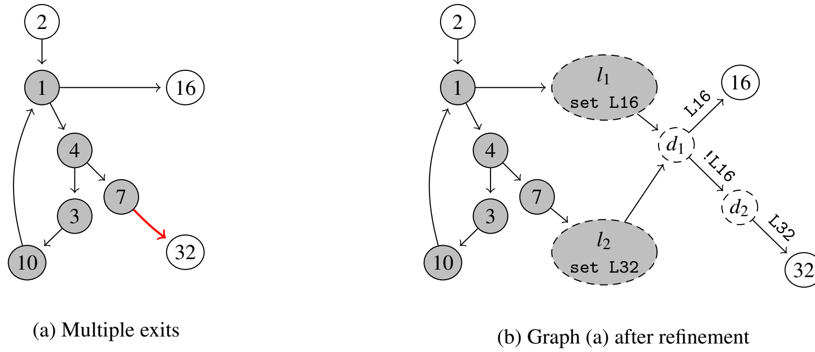
(a) Multiple entries

(b) Graph (a) after refinement

Figure 3: In red: abnormal entry in candidate region

Abnormal Exit. A loop is said to have abnormal exits when not all its exists point to the same external node. This scenario can be visualized as the combination of a `break` with a `goto`, which means the last iteration terminates abruptly, skipping a portion of the loop and jumping to a different part of the program. With reference to Figure 4a, when node 7 is executed, execution continues to node 32, which is outside the loop. The remaining nodes of the body are then not executed. As shown in Figure 4b, similarly to the previous paragraph, every exit is moved to a newly created decision node (d_1) *outside* the loop, to which multiple decision nodes can be chained if required.

[8] As a matter of fact, albeit this technique has been developed independently, a similar result can be obtained by combining [9, 22].



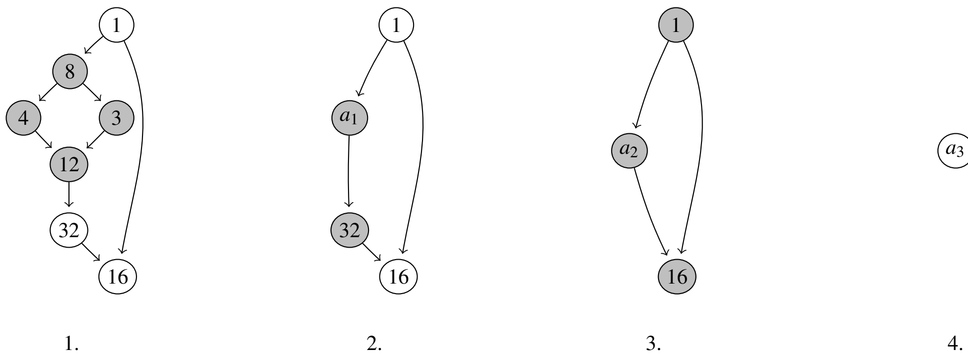
(a) Multiple exits

(b) Graph (a) after refinement

Figure 4: In red: abnormal exit from candidate region

3.3.2. Acyclic Region

An acyclic region is defined as a set of CFG nodes included between a node and its immediate postdominator[9], where the *single-entry, single-exit* property holds. As shown in Figure 5 and stated at the beginning of this section, the region identification process recursively replaces graph components with individual nodes. Once the innermost region is replaced with a single node a_1, it is possible to identify the region a_2, and the process continues until no other acyclic region can be identified.



1. 2. 3. 4.

Figure 5: Successive steps in acyclic region analysis, nodes in gray will be encapsulated

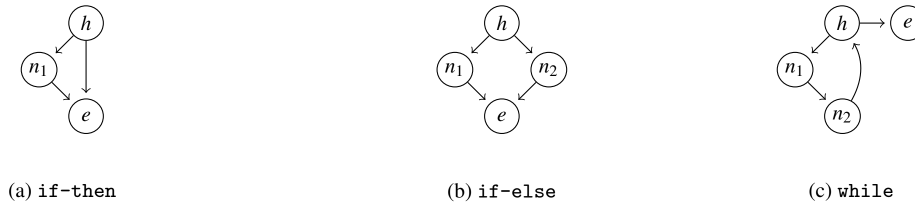
3.4. Code structuring

The region identification process carried out in the previous step inferred hierarchical information from the CFG, highlighting logical code blocks boundaries. At this point, for example, the graph of an `if` clause body will be a region by itself, while the body of the `else` branch will be in a different region. This partitioning allows code generation via a simple substitution process, where every region is replaced with a node containing the equivalent *JavaScript* code, already wrapped in the appropriate control structure.

The code structuring process addresses the reconstruction of different control flow structures, including `if-then`, `if-else`, and `while` loops. Owing to the previously achieved hierarchical scheme, at every level the correct control structure depends solely on the topological shape of the current graph (cf. Figure 6). It is possible to discern these three structures thanks to their characteristic shape, which directly correlates to a specific meaning:

[9] In a directed graph, a node a dominates another node b if, when traversing the graph from its entry, every path from the entry to b passes through a. Similarly, a node b postdominates another node a if every path to the exit passes through b. Finally, a node b immediately postdominates another node a if b does not postdominate any other postdominator of a.

- The `if-then` structure implies the body of the `if` *might* be executed before reaching the end node. The end might as well be executed directly, skipping the `if` body. This semantics can be represented by a triangle, where the head is connected to the body, which is then connected to the end, and directly to the end (cf. Figure 6a).
- The `if-else` structure implies that *either* side of the branch *has* to be executed, and only then the end can be reached. This means that the head must be connected with the two candidate code blocks, which are then connected with the same end node. These connections generate a diamond-shaped graph (cfr. Figure 6b).
- The `while` structure, as previously mentioned, is characterized by a *backedge* aiming at the head of the region, which can leave the region. Loops are cyclic regions of the graph (cfr. Figure 6c).



(a) `if-then` (b) `if-else` (c) `while`

Figure 6: CFG topological shapes of different control structures

Much like region identification, the code structuring process “bubbles up” a graph-to-code conversion, starting from the innermost region. The final result is a single node embodying *unflattened* and structured code for the whole program. The output is valid, `goto`-free *JavaScript* code semantically equivalent to the original obfuscated input.

3.5. Partial evaluation

The output of code structuring (Section 3.4) is guaranteed to be consequential code, which means basic blocks to be executed one after another are sequential, thereby allowing a human reader to follow the execution flow easily. Still, any obfuscation technique outlined in Section 2 other than control flow flattening has not been addressed yet. *Data splitting*, for example, often accompanies and greatly benefits from control flow flattening: scattering basic blocks through the codebase makes it difficult to deduce the value of a variable at a specific code location when its final value is dynamically generated. Moreover, *JavaScript* programs heavily rely on strings to perform their tasks, making string splitting actively used.

JACOB involves then a final phase to attempt data reconstruction via *partial evaluation*, a technique for program optimization through specialization. In theory, a partial evaluator accepts in input a program and outputs a specialized version that is equivalent in behavior, where all the static inputs have been precomputed. Conceived for optimization purposes, a partial evaluator elegantly defeats *data splitting* as well. Reconstructing data at runtime is inherently inefficient, thus optimizing the code (by precomputing static inputs) results in a program where data reconstruction code is replaced with the final value of data itself.

4. *JACOB* at Work: Uncovering Unwanted User Tracking and Fingerprinting in Online Shopping

JACOB has been successfully exercised for deobfuscating a specific script deployed by a pool of e-commerce websites operated by the *Alibaba Group*. A suspicious script was noticed, thanks to the anti-fingerprinting features of the *Mozilla Firefox* browser, while one of the authors of this paper was logging in to the *AliExpress* website, at which point he was notified of an ongoing canvas tracking attempt. Appearing in a supposedly simple webpage, the *Firefox* message piqued the author’s curiosity, so initial analysis of the webpage was undertaken to pinpoint which action was triggering the fingerprinting and how pervasive it was.

4.1. User tracking and fingerprinting

In general, information about user activities are collected during navigation by websites operators and third parties to infer preferences and serve targeted content. Services like *Netflix* and *Spotify* offer personalized content based on

customer's interests, while *Google* finalizes search results in light of previously visited links. Most typically, tracking can be performed for commercial purposes, including advertisement targeting and rating of user's trustworthiness. It is even possible to track users through their entire online life. For example, *Google* ubiquitous *Ads* and *Facebook* iconic *Like* button, being embedded in each and every website, allow the two companies to track which pages a user is visiting, even outside their domain. Tracking is powered by several techniques, which may fall in three main categories:

- *Cookies*. A cookie is a small chunk of information saved by websites among other browser data, that is used to store information about user preferences or session data. Cookies can, and often do, store a unique user identifier.
- *Fingerprinting*. A digital fingerprint is a set of attributes relevant to a machine or operating system installation, such as screen size/resolution, installed fonts, hardware components, driver behavior and more. By putting all these apparently insignificant information together, it is possible to identify a specific machine [7].
- *Behavioral*. By exploiting the imperceptible differences in human behavior, it is possible to identify a visitor by their interactions with webpages. Studies have shown the possibility to manage authentication via keypress analysis [1], or to unlock a phone with sensors and speaker data alongside usage statistics [12].

The above techniques greatly differ in pervasiveness, however. While cookies, being saved on the user's machine, can be deleted or ignored via specific browser features (private/incognito browsing), it is hard to mask a computer's hardware and its quirks: fingerprint data might be sent to servers alongside required information and processed opaquely by the back-end without the user even noticing.

4.2. Collina, the suspect

Following the message of the *Firefox* browser notifying a tracking attempt by the *AliExpress* webpage, the attention was driven to a self-contained script, named `collina.js` (hereafter simply referred to as *collina*), operating independently of the rest of the front-end. It was discovered that, when fingerprinting is deemed necessary, another script in the webpage requests *collina* to the server, executes it, then waits for *collina* to be completely loaded; finally the script can retrieve the machine's fingerprint via a function exposed in the HTML DOM[10] by *collina*. This fingerprint is then sent to the server as an HTTP request header named `ua`, alongside strictly necessary information.

4.3. Collina deobfuscation

The original *collina* file, as downloaded from the *Alibaba* servers, consists of two long lines of incomprehensible *JavaScript* code. The result of mere beautification[11] on *collina* shows that this code is composed of a large anonymous function, which includes several nested functions. The body of the principal function consists of a `for` loop that embodies several nested `switch` control structures, where each bottom-most `case` represents a basic block.

The switch cascade is controlled by a series of variables derived from a single global value in an intricate way. Given how assigning a single value can control the execution flow, it can be assimilated to the program counter of a CPU, and every assignment to that global variable can be considered analogous to a `goto`. This is an example of the dispatcher mentioned in Section 3.

Beyond control flow flattening, the analysis of *collina* highlights additional obfuscation techniques, including *function-wide variable reuse*, *string obfuscation*, and usage of a *virtual stack*. All the variables (as many as 350) are declared at the beginning of every function, and there are few (if any) declarations in the body; to reduce intelligibility further, every variable is reused multiple times to store different data types. Strings are split, reversed, and encoded[12] to be reconstructed at runtime. An additional string decoding system relies on a virtual global stack to mimic a stack-based calling convention[13] for passing input data to a custom decoding function.

[10] The HTML DOM is a *JavaScript* application program interface that allows *JavaScript* to add/remove/manipulate HTML elements and react to HTML events, such as `click`, `mouse over an element`, `keypress`, `element change`, and so forth.

[11] Beautification is the process of formatting a source code in order to graphically facilitate the comprehension.

[12] Encoding by incrementing/decrementing/xor-ing every character with a predefined value.

[13] A calling convention defines how arguments are passed to functions and how the return value will be passed to the caller in Assembly language.

4.4. After examination of evidence

After an accurate reading of the script generated by means of *JACOB*, the doubts advanced in Section 4.2 have been confirmed: *extensive user tracking and fingerprinting are performed by collina*. The structure of the program revolves around a single function, which takes up 99.5% of the lines of code in the output file. This function performs a variety of different tasks and is likely the result of multiple inlining operations, given the amount of duplicated code. The behavior of this function changes according to its arguments: of its five parameters, the first one is always present, while the others are optional. This function even autonomously generates an HTML `<script>` object which is injected into the webpage to expose its own external interface.

As soon as *collina* is loaded in the webpage, an initialization process is undertaken to retrieve many browser features and quirks, which allow for precise environment version pinpointing. Given that *JavaScript* Application Program Interfaces (APIs) often vary, with new ones being added to browsers and other being deprecated, different versions of the same browser will support different features, just like different browsers would. Checking for support of a variety of different APIs allows *collina* to validate whether the information reported in the user-agent[14] are coherent with the browser and environment (e.g. mobile or desktop) in use, preventing the user from forging its own user-agent.

Also, *collina* registers itself as an event listener for a series of different events, such as `touchstart`, `touchend`, `mousemove`, `deviceorientation`, and `keyup`. All these events allow for tracking all the interactions with the webpage, following the user's movements to fingerprint it. Specific detection is performed for testing frameworks, like *Selenium* [18] and *Puppeteer* [15], which are commonly used for web browser automation. Moreover, several anti tampering features are added, such as a duplicate timestamp: the execution timestamp is saved in a variable, then the result of the integer division by 2^{32} is saved in another variable alongside the remainder of the division. This prevents the tampering of fingerprinting data without a complete understanding of the inner workings of the software.

5. Conclusion

JACOB is based on established concepts, which are fundamental to binary decompilers and deobfuscators. Porting these techniques to a high-level language such as *JavaScript* brought new complexities, but also made the task easier. For example, the need to output valid *JavaScript* code imposed strict limits on the synthesizable structures (cf. Paragraph 3.3.1) and required devising new solutions to modify the input CFG. On the other hand, not handling Assembly code, many complex parts of a decompiler were not needed, such as function identification, type inference, and calling convention guessing. Albeit conceived for *JavaScript*, the *JACOB* methodology could be easily adapted to other source-level obfuscated code that is written in a different programming language.

The *JACOB* approach to analyze control flow flattened *JavaScript* code has been implemented in the open source project `bulldozer`. Notably, although several tools for recovering control structures from binary code exist (decompilers), to our knowledge, `bulldozer` is the first performing such a task on *JavaScript* source code. Developing `bulldozer` was a non-trivial task due to the vertical integration of multiple tasks in a single software. Binary decompilers do not handle CFG recovery, while binary control-flow unflattening deobfuscators leverage decompilers for region identification and code structuring. Due to the lack of similar tools targeting *JavaScript* code, `bulldozer` was written completely from scratch, except for the region identification algorithm (Section 3.3), which is closely tied to *angr* [19], a binary analysis framework. The source code of `bulldozer` and a relevant bachelor thesis [4] explaining the implementation details, along with a deobfuscated and commented version of the *collina* code, can be found at: https://github.com/ceres-c/bulldozer

Although extensive analysis of deobfuscated *collina* code shows no evidence of any explicit known attack against the user, the fact remains that fingerprinting unequivocally violates the user's privacy, a conclusion that would have been hardly possible without *JACOB*. We stress once again that, although code obfuscation can be legitimately adopted for intellectual-property protection, it can also be exploited for illegal/unwanted actions (fingerprinting included), as well as for hiding criminal intents. As many other things, code obfuscation is a double-edged sword, so it is paramount to know which edge of the sword is actually being wielded: *JACOB* was designed precisely to serve this purpose.

[14] The user-agent is the HTTP request header containing a string which reports information about the browser, such as product name and version.

References

[1] Bilan, S., Bilan, M., Bilan, A., 2020. Interactive biometric identification system based on the keystroke dynamic, in: Biometric Identification Technologies Based on Modern Data Mining Methods. Springer International Publishing, pp. 39–58. doi:10.1007/978-3-030-48378-4_3.

[2] Böhm, C., Jacopini, G., 1966. Flow diagrams, turing machines and languages with only two formation rules. Commun. ACM 9, 366–371. URL: https://doi.org/10.1145/355592.365646, doi:10.1145/355592.365646.

[3] Brown, F., Narayan, S., Wahby, R., Engler, D., Jhala, R., Stefan, D., 2017. Finding and preventing bugs in javascript bindings, pp. 559–578. doi:10.1109/SP.2017.68.

[4] Cerutti, F., 2021. Design and implementation of Bulldozer, a decompiler for JavaScript "binaries".

[5] Chen, J., Yan, T., Wang, T., Fu, Y., 2020. A closer look at the web skimmer. URL: https://unit42.paloaltonetworks.com/web-skimmer/.

[6] Collberg, C., Thomborson, C., Low, D., 1997. A taxonomy of obfuscating transformations.

[7] EFF, . Cover your tracks. URL: https://coveryourtracks.eff.org/.

[8] Gruss, D., Maurice, C., Mangard, S., 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript, in: Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, pp. 300–321. doi:10.1007/978-3-319-40667-1_15.

[9] Gussoni, A., Di Federico, A., Fezzardi, P., Agosta, G., 2020. A comb for decompiled c code, in: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. pp. 637–651. URL: https://doi.org/10.1145/3320269.3384766, doi:10.1145/3320269.3384766.

[10] Kan, Z., Wang, H., Wu, L., Guo, Y., Xu, G., 2019. Deobfuscating android native binary code, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 322–323. doi:10.1109/ICSE-Companion.2019.00135.

[11] Kang, Z., 2021. A review on javascript engine vulnerability mining. Journal of Physics: Conference Series 1744, 042197. doi:10.1088/1742-6596/1744/4/042197.

[12] López, J.M.E., Celdrán, A.H., Marín-Blázquez, J.G., Esquembre, F., Pérez, G.M., 2021. S3: An AI-enabled user continuous authentication for smartphones based on sensors, statistics and speaker information. Sensors 21, 3765. doi:10.3390/s21113765.

[13] Madou, M., Van Put, L., De Bosschere, K., 2006. Understanding obfuscated code, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), pp. 268–274. doi:10.1109/ICPC.2006.49.

[14] Park, S., Xu, W., Yun, I., Jang, D., Kim, T., 2020. Fuzzing javascript engines with aspect-preserving mutation, pp. 1629–1642. doi:10.1109/SP40000.2020.00067.

[15] Puppeteer devs, . puppeteer. URL: https://github.com/puppeteer/puppeteer.

[16] Rinsma, T., 2017. Seeing through obfuscation: interactive detection and removal of opaque predicates. Master's thesis. Radboud University. Nijmegen, The Netherlands.

[17] Röttger, S., Janc, A., 2021. leaky.page. URL: https://leaky.page/.

[18] Selenium devs, . selenium. URL: https://github.com/SeleniumHQ/selenium.

[19] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G., 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis, in: IEEE Symposium on Security and Privacy, pp. 138–157. doi:10.1109/SP.2016.17.

[20] Tian, Y., Qin, X., Gan, S., 2021. Research on Fuzzing Technology for JavaScript Engines. Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/3487075.3487107.

[21] Udupa, S., Debray, S., Madou, M., 2005. Deobfuscation: reverse engineering obfuscated code, in: 12th Working Conference on Reverse Engineering (WCRE'05), pp. 10 pp.–54. doi:10.1109/WCRE.2005.13.

[22] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., Smith, M., 2015. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations, in: NDSS. doi:10.14722/ndss.2015.23185.