Byte by Byte: Unmasking Browser Fingerprinting at the Function Level Using V8 Bytecode Transformers

Pouneh Nikkhah Bahrami*
University of California, Davis
Davis, California, USA
pnikkhah@ucdavis.edu

Dylan Cutler
Google
Cambridge, Massachusetts, USA
dylancutler@google.com

Igor Bilogrevic
Google
Zurich, Switzerland
ibilogrevic@google.com

Abstract

Browser fingerprinting enables persistent cross-site user tracking via subtle techniques that often evade conventional defenses or cause website breakage when script-level blocking countermeasures are applied. Addressing these challenges requires detection methods offering both function-level precision to minimize breakage and inherent robustness against code obfuscation and URL manipulation.

We introduce ByteDefender, the first system leveraging V8 engine bytecode to detect fingerprinting operations specifically at the JavaScript function level. A Transformer-based classifier, trained offline on bytecode sequences, accurately identifies functions exhibiting fingerprinting behavior. We develop and evaluate lightweight signatures derived from this model to enable low-overhead, on-device matching against function bytecode during compilation but prior to execution, which only adds a 4% (average) latency to the page load time. This mechanism facilitates targeted, real-time prevention of fingerprinting function execution, thereby preserving legitimate script functionality. Operating directly on bytecode ensures inherent resilience against common code obfuscation and URL-based evasion. Our evaluation on the top 100k websites demonstrates high detection accuracy at both function- and script-level, with substantial improvements over state-of-the-art AST-based methods, particularly in robustness against obfuscation. ByteDefender offers a practical framework for effective, precise, and robust fingerprinting mitigation.

CCS Concepts

• **Security and privacy → Privacy protections**.

Keywords

Browser Fingerprinting, Bytecode Analysis, Function-level Detection, Transformer Models, Obfuscation Resilience

ACM Reference Format:
Pouneh Nikkhah Bahrami, Dylan Cutler, and Igor Bilogrevic. 2025. Byte by Byte: Unmasking Browser Fingerprinting at the Function Level Using V8 Bytecode Transformers. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3719027.3765158

*Also with Google.

1 Introduction

Modern JavaScript APIs grant web applications powerful capabilities, from interactive graphics to real-time communication but also expose system details that can be exploited for *browser fingerprinting*. This technique involves gathering a combination of device and browser characteristics, often through subtle variations in API responses or rendering outputs (e.g., Canvas [37], AudioContext [10], fonts [8, 57]), to create stable and quasi-unique identifiers. While sites may use these techniques for safety-oriented purposes (e.g. anti-fraud, anti-abuse, bot detection) [5, 15, 30, 33, 49] and for enhancing security by identifying returning devices without traditional authentication, they can also be re-purposed for pervasive, cross-site user tracking [16, 17, 27, 34, 43, 57]. This tracking capability, often operating statelessly and without clear user awareness or control [7, 39], raises significant privacy concerns since it can bypass standard privacy-enhancing tools and user preferences [27].

Mitigating the privacy risks associated with fingerprinting, *without* undermining its safety-oriented uses or breaking website functionality, remains an open challenge. Existing countermeasures often fall short, as fine-grained defenses (e.g. adding noise to API outputs [11, 52], normalizing values [4, 8, 54], or restricting API access [59, 61]) could inadvertently disrupt critical fingerprinting applications (e.g., security checks) or break essential website features [57]. Furthermore, scripts may circumvent such defenses [57] or embrace newly developed fingerprinting vectors [3].

Coarse-grained methods, primarily URL-based filter lists [14, 17, 42], struggle to differentiate between scripts using fingerprinting for cross-site tracking versus non-tracking purposes. They also suffer from delays in updates and are vulnerable against common evasion techniques like CNAME cloaking [9, 13], URL randomization [60], and code obfuscation [36, 38, 50]. Critically, blocking entire scripts often leads to web breakage, especially problematic when scripts mix essential functions with fingerprinting capabilities [2], and is ineffective against inline scripts [27].

Machine learning-based methods aim to automate the identification of fingerprinting behaviors by leveraging both static and dynamic features of JavaScript code. Static analysis approaches using ASTs [27, 46, 56] have demonstrated effectiveness but remain susceptible to obfuscation techniques [38, 50]. Recent work by Ghasemisharif and Polakis [22] utilized V8 bytecode to detect general *tracking* scripts at the *script-level*, showing resilience against obfuscation. Amjad et al. [2] porposed a *function-level* detection framework for *tracking* behavior using dynamic execution context graphs. However, these approaches encounter three key limitations in addressing the fingerprinting dilemma. First, script-level analysis [22, 27] is too coarse to isolate tracking behaviors embedded within multi-purpose scripts, risking disruption of core functionality or failing to block trackers. Second, dynamic context analysis [2], while finer-grained, can be complex and can introduce runtime overhead, making efficient *pre-execution* intervention difficult. Third, neither [2, 22] specifically isolates fingerprinting behaviors; instead, both conflate stateless (fingerprinting) and stateful (traditional) tracking techniques. Yet, these two forms of tracking differ fundamentally in their mechanisms and objectives, and a unified detection approach is likely to exhibit mixed effectiveness.

We argue that addressing the privacy risks of fingerprinting requires a solution that is precise (targeting individual functions), robust (resilient to evasion), and proactive (acting *before* execution). We introduce ByteDefender, the first system using V8 engine bytecode for function-level classification specifically aimed at identifying fingerprinting functions prior to execution. By analyzing the bytecode of individual functions during JavaScript compilation, ByteDefender achieves fine granularity and inherits robustness against source-level obfuscation [38, 47].

Distinct from dynamic context methods [2, 27], ByteDefender focuses on the static bytecode patterns within functions, enabling lightweight, pre-execution detection. We employ a Transformer-based classifier [12, 31, 58], trained offline to recognize specific bytecode behaviors associated with fingerprinting techniques [17, 27]. ByteDefender can identify fingerprinting functions for efficient on-device matching against known fingerprinting signatures, enabling it to preemptively block or alter the execution of fingerprinting functions and mitigate potential privacy leakage without disrupting entire scripts. This targeted approach minimizes web breakage compared to script-level methods [22, 27] and allows core website functionality to proceed without disruption.

ByteDefender provides a practical pathway to effectively mitigate the privacy risks associated with browser fingerprinting while better preserving web compatibility. Our main contributions are:

(1) Introduction of the first machine-learning method using V8 engine bytecode for browser fingerprinting detection specifically at the JavaScript function level. Our approach, utilizing a Transformer architecture [58] to classify static bytecode sequences, achieves high accuracy (98.9%), precision (84.0%), and recall (85.1%) without reliance on dynamic execution context [2] or predefined API lists [27], while demonstrating robustness against evasion techniques.
(2) Large-scale evaluation on 100,000 websites, showing high accuracy (99.7%) and significant improvements over AST-based methods [27, 46, 56] for script-level fingerprinting detection, especially on obfuscated JavaScript code [36, 50] and origin/URL manipulation [9, 13, 60].
(3) Development of a practical, low-overhead mechanism that only adds 4% latency (on average) to the page load time for on-device detection and prevention of fingerprinting functions during JavaScript compilation, *before* execution occurs.
(4) Public release of the ByteDefender, implementation, and codebase to facilitate reproducibility and further research in bytecode-based fingerprinting mitigation.

The remainder of this paper is structured as follows. Section 2 presents background on JavaScript execution, browser fingerprinting, existing countermeasures, and related work. Section 3 details the design and implementation of ByteDefender, data collection, feature representation, and classifier architecture. Section 4 describes the experimental setup, dataset characteristics, and implementation of our classifier. Section 5 presents our evaluation results, including accuracy metrics, robustness analysis against obfuscation, comparison with state-of-the-art methods, and the performance overhead of our model in the browser. Section 6 discusses limitations of our approach. Section 7 concludes the paper.

2 Background and Related Work

This section provides background on the V8 JavaScript execution pipeline, browser fingerprinting techniques and countermeasures, and positions our work with respect to prior research in browser fingerprinting detection.

2.1 JavaScript Execution and V8 Bytecode

JavaScript execution in modern browsers follows a multi-stage pipeline implemented within the V8 engine [26, 53]. As illustrated in Figure 1, raw JavaScript source code is first parsed into an Abstract Syntax Tree (AST), which represents the code's syntactic structure. V8's interpreter, Ignition, then compiles this AST into bytecode – a platform-independent, intermediate representation optimized for quick generation and execution. This bytecode serves as the input for further stages, including interpretation or Just-In-Time (JIT) compilation into optimized machine code by compilers (e.g. TurboFan) for frequently executed functions. Notably, V8 often employs lazy compilation, generating bytecode for functions only when they are needed [2].

The bytecode generated by Ignition is compact and not directly human-readable. However, built-in tools allow disassembly of the bytecode into a sequence of readable instructions such as `LdaConstant`, `CallProperty1`, and `JumpIfFalse`. ByteDefender analyzes such a sequence of bytecode instructions produced by Ignition, prior to execution or further optimization.

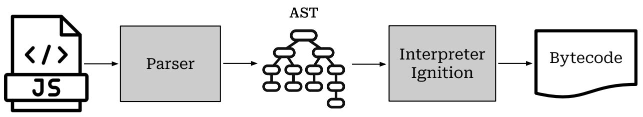


Figure 1: Simplified V8's compilation pipeline

2.2 Browser Fingerprinting

Browser fingerprinting is a technique for gathering device and browser-specific information through various APIs and configurations to create a quasi-unique identifier. Unlike traditional stateful tracking via cookies, fingerprinting is typically stateless and less apparent to users. Common vectors include probing the Canvas API [37], AudioContext [10], font availability and rendering [8, 57], WebGL capabilities [37, 63], WebRTC interfaces [19, 45], and querying numerous 'navigator' properties or system settings. A combination of these attributes can derive an identifier with high uniqueness and stability over time.

While acknowledged by browser vendors [7, 20, 62] and standard bodies [39] as posing privacy risks when used for cross-site tracking [17], fingerprinting techniques also have safety-oriented applications in areas such as fraud detection and device authentication [5, 15, 30, 33, 49]. This dual-use nature makes mitigation complex, as overly broad countermeasures can break such security and anti-abuse mechanisms.

Existing mitigation techniques struggle to effectively address the diverse implementation strategies of fingerprinting. Fine-grained API defenses, such as randomization [11, 52], normalization [4, 8, 54], or blocking [61], could break functionality or can be circumvented [3, 11, 57]. Coarse-grained filter lists [14, 17, 42] are slow to update, are easily evaded by CNAME cloaking [9, 13], URL randomization [60], or obfuscation [36, 38, 50], and cause breakage in mixed scripts [2, 27]. Machine learning offers automation but faces trade-offs.

Static analysis using Abstract Syntax Trees (ASTs), explored in [46, 56] and adopted by FP-Inspector [27], analyzes the structural representation of JavaScript code. These methods extract features such as specific API calls (e.g., `toDataURL`, `measureText`), structural patterns (e.g., sensitive APIs invoked within loops) and various complexity metrics. Although ASTs offer a structured view of code semantics, they are highly sensitive to syntactic changes, making them vulnerable to even minor obfuscation. Code obfuscation techniques [36, 38, 50], including renaming variables/functions, inserting dead code, or flattening control flow, drastically alter the AST structure. Moreover, ASTs require extensive post-processing to extract usable features and are not directly suitable as input to learning models. These limitations make AST-based techniques ill-suited for real-time, proactive fingerprinting detection and hinder their ability to generalize to obfuscated or novel scripts.

Dynamic analysis addresses some of these issues by observing runtime behavior. FP-Inspector [27] supplemented its static analysis with dynamic features obtained by executing scripts. More recently, NoT.js [2] employed extensive dynamic analysis using browser instrumentation (via the DevTools Protocol) to build detailed execution graphs. These graphs capture the sequence of function calls (call stack) and the data accessible at each point (scope chain) when network requests or other specific events occur. NoT.js uses features derived from this graph (e.g., node types, connectivity) to train a classifier identifying tracking functions, focusing on their role in initiating tracking requests, and subsequently generates surrogate scripts. While powerful for understanding runtime execution flow and achieving function-level granularity for tracking, this approach requires significant browser instrumentation, potentially incurring runtime overhead. Crucially, detection happens during or after execution, making preemptive blocking difficult. Furthermore, its features are derived from the effects of execution (e.g., API calls made, network requests sent), whereas ByteDefender analyzes the intrinsic structure (bytecode) of the function itself before execution to predict its fingerprinting potential based on learned patterns corresponding to fingerprinting API usage.

Script-level bytecode analysis, introduced by Ghasemisharif and Polakis [22] inspired by [47] for general tracking/ad detection, established V8 bytecode as a robust feature source resilient to obfuscation. They treated entire JavaScript files as documents and applied text classification techniques (e.g., DPCNN and FastText) to the concatenated bytecode sequences of all functions within a script. While demonstrating bytecode's advantages over ASTs, this script-level granularity introduces two major limitations. First, it conflates fundamentally distinct forms of tracking: stateless and stateful, resulting in mixed detection effectiveness. Second, labeling the entire script as tracking can lead to coarse blocking, mirroring the breakage issues seen with filter lists [27] when legitimate and tracking functions coexist in a single script.

ByteDefender integrates the strengths of these related areas while overcoming their key limitations for the specific task of fingerprinting detection. It adopts V8 bytecode as the core feature, inheriting the demonstrated obfuscation resilience [22, 47] that AST-based methods lack [27, 46, 56]. Crucially, unlike script-level bytecode analysis [22, 27], ByteDefender operates at the function-level. By analyzing the bytecode sequence of each individual function and complementing it with execution traces of the function, the approach constructs a novel function-level ground truth that provides the fine granularity needed to identify specific fingerprinting behaviors within mixed-purpose scripts, thereby minimizing the risk of web breakage associated with blocking entire files. Distinct from dynamic analysis approaches such as NoT.js [2], ByteDefender performs static analysis on the function's bytecode during compilation and before execution. This allows for lightweight, proactive intervention, preventing potential information leakage before the fingerprinting function runs, a significant advantage over methods requiring runtime observation. Instead of relying on dynamic call graphs or execution context, ByteDefender trains a Transformer model [58] to directly recognize the intrinsic bytecode patterns and instruction sequences to identify signatures for compilation-time blocking. This focus on the code's static representation at the bytecode level, combined with function-level granularity and pre-execution timing, provides a unique approach specifically tailored to proactively mitigating fingerprinting risks.

3 ByteDefender: Function-Level Fingerprinting Detection

In this section we detail the design and implementation of ByteDefender. The core methodology involves collecting function-level bytecode and corresponding execution traces, using these traces to establish ground truth labels for fingerprinting activity, mapping these labels to the corresponding bytecodes, and training a Transformer-based classifier on the labeled bytecode sequences. Figure 2 provides a high-level overview of ByteDefender.

A primary challenge in this work is the lack of pre-existing ground truth labels for function bytecode as it relates to fingerprinting. To overcome this, we leverage execution trace analysis to infer labeling at the function level. Prior research confirms that fingerprinting techniques rely on specific patterns of Web API usage [10, 17, 27]. For instance, canvas fingerprinting typically involves calls to `fillText()` followed by `toDataURL()` or `getImageData()` [37]. Similarly, AudioContext fingerprinting uses a characteristic sequence including methods such as `createOscillator` and `startRendering` [10].

By identifying known API interaction patterns in execution traces captured during web crawls, we can heuristically label functions that exhibit fingerprinting behavior. This labeled trace data then serves as the basis for training our bytecode classifier.

Our approach comprises several key steps: (1) instrumenting the V8 engine to extract detailed function-level bytecode along with necessary metadata (script URL, script Id, function name); (2) using a browser extension to capture fine-grained execution traces, focusing on high-entropy API calls; (3) collecting bytecode and trace data via automated web crawling; (4) applying heuristics based on known fingerprinting techniques to label functions in the trace data; (5) mapping these labels to their corresponding function bytecode sequences; and (6) training and evaluating classifiers on the resulting labeled bytecode dataset.

3.1 Instrumented V8 Bytecode Collection

As we discussed in Section 2.1, V8 converts JavaScript source code into bytecode via the Ignition interpreter. While standard V8 tools (such as d8 and Node.js with the `-print-bytecode` flag) expose bytecode, they do not provide essential metadata that links the bytecode to its originating script URL, script ID, or function name. Listing 2 displays the original bytecode for the JavaScript code found in Listing 1, which was generated by V8 during the compilation process. As shown in Listing 2, there is a lack of detailed information about the script and function being compiled. This metadata is crucial for mapping ground-truth labels derived from execution traces (which contain script/function context) to the corresponding bytecode sequences.

Listing 1: A Script source code which performs browser fingerprinting

```
function gatherFingerprint() {
    var fingerprint = {
        userAgent: navigator.userAgent,
        language: navigator.language,
        platform: navigator.platform,
        screenResolution: `${screen.width}x${screen.
             height}`
    };
    return fingerprint;
}
gatherFingerprint();
```

Listing 2: A truncated and original bytecode generated by V8 for the script shown in Listing 1

```
[generated bytecode (0x16ce00098fb5 <SharedFunctionInfo>)]
Bytecode length: 79
Parameter count 1
Register count 4
Frame size 32
0x7ff2001400c8 @ 0 :  85 00  CreateObjectLiteral [0], [0]
0x7ff2001400cc @ 4 :  cd     Star1
0x7ff2001400cd @ 5 :  23 01  LdaGlobal [1], [1]
0x7ff2001400d0 @ 8 :  cc     Star2
0x7ff2001400d1 @ 9 :  33 f7  GetNamedProperty r2, [2]
0x7ff2001400d5 @ 13 : 3a f8 DefineNamedOwnProperty r1
0x7ff2001400d9 @ 17 : 23 01 LdaGlobal [1], [1]
0x7ff2001400dd @ 21 : 33 f7 GetNamedProperty r2, [3], [7]
...
```

To overcome this limitation, we instrument the V8 source code. Specifically, we modify the `DoFinalizeJobImpl()` function within `interpreter.cc` to access and output the `scriptId`, `scriptURL`, and `functionName` associated with the `SharedFunctionInfo` object, alongside the generated bytecode instructions by `BytecodeArray` class. We call `BytecodeArray::Disassemble()` to convert the binary format of raw bytecodes to the corresponding mnemonic name that represent each bytecode instruction.

Furthermore, to manage the verbosity of bytecode output for large scripts, we modify V8's bytecode printing functions (`NameForRuntimeId`, `Decode` in `bytecode-decoder.cc`, and `Disassemble` in `bytecode-array.cc`) to log only the sequence of bytecode instruction names, omitting memory addresses, operands, comments, and bytecode offsets.

While this choice simplifies feature extraction and reduces noise, it also introduces a potential limitation: the loss of fine-grained semantic context, including constants, property names, or literal values (e.g., "userAgent", "canvas", or "toDataURL"), which may help in distinguishing between non-fingerprinting and fingerprinting behavior. However, this design decision is intentional and aligned with our goal of achieving generalizable, obfuscation-resilient fingerprinting detection. Many common obfuscation techniques (e.g., string encoding, variable renaming, and control flow flattening) target operands and syntactic constructs, but tend to preserve the underlying operational structure captured in the bytecode instruction sequences. By focusing solely on opcode patterns and abstracting away operands, our model learns behavioral representations that are more invariant to obfuscation. Listing 3 shows the output format from our instrumented V8 for the script in Listing 1, now including the necessary metadata.

To handle concurrent compilation tasks within V8, which uses multiple processes and threads, we ensure unique output filenames for each compilation unit by incorporating thread ID, process ID, and a timestamp into the filename, preventing concurrent write conflicts or file overwrites.

Listing 3: Bytecode generated by our instrumented V8 for the script in Listing 1

```
Script URL: \url{https://example.com/fpjs.js}
Script ID: 3
Function name: gatherFingerprint
Bytecode:
Parameter count 1
Register count 4
Frame size 32
[CreateObjectLiteral,Star1,LdaGlobal,Star,GetNamedProprty,
DefineNamedOwnProperty,LdaGlobal,Star,GetNamedProperty,
DefineNamedOwnProperty,LdaGlobal,Star2,GetNamedProperty,
DefineNamedOwnProperty,LdaGlobal,Star3,GetNamedProperty,
ToString,Star2,LdaConstant,Add,Star2,LdaGlobal,Star3,
GetNamedProperty,ToString,Add,DefineNamedOwnProperty,Mov,
Ldar,Return]
```

3.2 Execution Trace Collection via Browser Extension

Complementary to bytecode collection, we capture dynamic execution traces using a custom Chrome browser extension. This extension employs the `Tracing` domain of Chrome DevTools Protocol (CDP) [55] which gathers fine-grained events that provide a detailed timeline of browser activities such as function calls during JavaScript execution. This data is captured as JSON objects.

The extension operates by attaching to the active browser tab and initiating a CDP tracing session. Crucially, for the purpose of generating ground truth labels via established heuristics (Section 3.4), this tracing session is specifically configured to capture

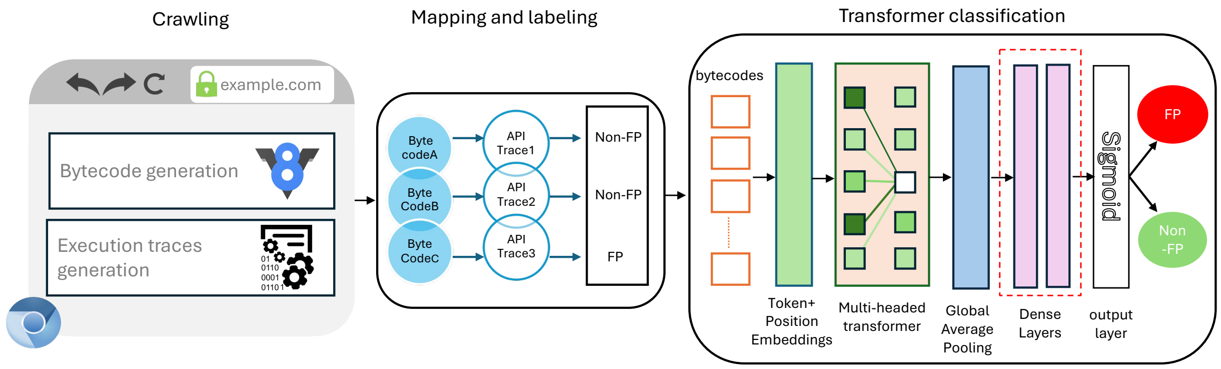


Figure 2: ByteDefender: (1) Web crawling is performed using a Chromium browser with an instrumented V8 engine to extract function-level bytecodes and execution traces via a custom extension. (2) Execution traces of functions are mapped to their corresponding function bytecodes and labeled as Fingerprinting (FP) or Non-Fingerprinting (Non-FP). (3) A transformer-based classifier is trained on labeled bytecodes.

events associated only with "High Entropy APIs" as designated by Chromium [25]. These APIs (Appendix A, Table 5) expose potentially identifying information and are natively instrumented within Chrome, which makes them detectable via the `Tracing` domain. Consequently, traces are collected only for functions that invoke at least one of these specific APIs. This focus on high-entropy API calls allows us to effectively apply heuristics from prior research [10, 17, 27] for labeling purposes.

During page execution, the extension listens for `Tracing.dataCollected` events corresponding to these high-entropy API calls. It parses these events to extract relevant details for each call, including the API identifier, arguments, the URL and function name of the calling script, line/column numbers, and the top-level page URL. Traces are filtered to exclude noise from invalid sources (e.g., internal browser pages starting with `chrome:`, extension URLs, `file:` URLs) or DevTools activity itself).

The filtered, structured trace data is then serialized to JSON and transmitted to our local server for further analysis and labeling. When scripts are parsed, the extension uses the CDP `Debugger` domain to retrieve and send the raw script source code, along with the script's unique identifier and URL, to the server for processing.

3.3 Data Collection Framework

We integrate the instrumented V8 browser (Section 3.1) and the tracing extension (Section 3.2) into an automated crawling framework built using Selenium WebDriver [48]. The framework systematically navigates to the homepages of target websites (Section 4). For each visited page, the instrumented V8 logs function-level bytecode with metadata, while the extension concurrently captures and filters execution traces and script sources. To allow sufficient time for dynamic scripts to execute, the crawler interacts with each page for a fixed duration (15 seconds) before proceeding. The crawler finally transmits the bytecode logs and serialized trace data to our local analysis servers.

3.4 Ground Truth Generation via Heuristic Labeling

As ground truth labels for fingerprinting behavior in bytecode are not available, we generate them heuristically by analyzing the collected execution traces, as described in Section 3.2. We leverage the fact that specific fingerprinting techniques are characterized by distinct patterns of high-entropy API calls [17, 27]. Based on established research, we define heuristics for four common fingerprinting categories: Canvas, Canvas Font, Audio Context, and WebRTC. Applying these heuristics to the captured traces allows us to label function execution instances as fingerprinting or non-fingerprinting and provides the ground truth for training our bytecode classifiers. Below we list fingerprinting techniques along with detection heuristics we used to build our initial ground truth of fingerprinting and non-fingerprinting functions.

Canvas Fingerprinting Heuristic. Inspired by [27, 37], this technique exploits differences in graphics rendering. To detect functions employing this technique, all of the following criteria must be met in the execution trace:

- Call to `fillText()` of `CanvasRenderingContext2D` or `OffscreenCanvasRenderingContext2D`.
- Call to the `toDataURL` method within the same function to export the Canvas content.
- The length of text written via `fillText()` is at least 10 characters.
- The function trace avoids invoking the `save`, `restore`, or `addEventListener` methods of the rendering context.

Canvas Font Fingerprinting Heuristic. This approach infers installed fonts by measuring text rendering dimensions [27], rather than direct browser's font list enumeration. To detect functions employing this technique, all of the following criteria must be met:

- Calls to `CanvasRenderingContext2D.measureText` or `OffscreenCanvasRenderingContext2D.measureText` occur at least 20 times within the function trace.
- Calls setting the font property (e.g., `CanvasRenderingContext2D.font.set`) occur for more than 20 different font values within the function trace.

Audio Context Fingerprinting Heuristic. This technique leverages subtle variations in audio signal processing due to hardware or browser differences [10]. To detect functions employing this technique, all of the following criteria must be met:

- Calls to characteristic AudioContext API methods for audio creation and manipulation are present (e.g., `BaseAudioContext.createOscillator`, `BaseAudioContext.createDynamicsCompressor`, `OfflineAudioContext.startRendering`, `AudioNode.connect`).
- A subsequent call to `AudioBuffer.getChannelData` occurs within the function trace to extract the processed audio data.

WebRTC Fingerprinting Heuristic. Exploits the exposure of low-level network details via `RTCPeerConnection` [19, 27, 45]. To detect functions employing this technique, all of the following criteria must be met:

- Call to `RTCPeerConnection.createDataChannel` or `RTCPeerConnection.createOffer` to initiate SDP generation.
- Call to `RTCPeerConnection.setLocalDescription` to access SDP details, potentially containing local IP addresses.

3.5 Mapping Labels to Bytecode Sequences

The crucial step is mapping the fingerprinting/non-fingerprinting labels derived from execution traces (Section 3.4) to the corresponding bytecode sequences collected by the instrumented V8 (Section 3.1). We do so by joining the trace data and the bytecode log data using the unique combination of *script URL*, *script ID*, and *function name* present in both datasets thanks to our instrumentation. To ensure a consistent dataset, we discard records associated with anonymous functions, as they lack a reliable name for joining. If a unique combination of *script URL*, *script ID*, and *function name* for a function's bytecode does not have a corresponding record in the execution traces, the function is labeled as non-fingerprinting. Since traces were only captured for functions invoking high-entropy APIs, the absence of a trace implies the function made no such calls, and therefore failed to trigger any of our fingerprinting detection heuristics. We also discard records with invalid or non-web URLs (e.g., internal V8 metrics, browser extension paths starting with `chrome-extensions::`, `v8/`, `chrome:`, `file:`) which are not valid URLs. This mapping process yields our final dataset: Sequences of function-level bytecode labeled as either fingerprinting or non-fingerprinting. We use this labeled dataset to train our classifiers.

3.6 Bytecode Representation via Word Embeddings

To leverage machine learning for classifying function behavior based on bytecode, we must first transform the discrete sequences of bytecode instructions (obtained as described in Section 3.1) into numerical vector representations suitable for model input. This process, known as embedding, maps each unique bytecode instruction (our "word" or "token") into a continuous vector space, ideally positioning instructions with similar semantic roles closer together. The choice of embedding technique impacts how contextual information and instruction relationships are captured. We explore established methods, adapting them for the unique characteristics of V8 bytecode.

Static Embeddings (Word2Vec and FastText). These methods generate a single, fixed vector representation for each unique bytecode instruction, learned from a large corpus of bytecode sequences in an unsupervised way.

- **Word2Vec** [35]: We employ the Skip-gram architecture, which learns embeddings by training a shallow neural network to predict the context bytecode instruction (neighboring bytecodes within a defined window) given a target bytecode instruction. For example, given the bytecode sequence '..., LdaGlobal, GetNamedProperty, Star, ...', the model learns vectors such that the embedding for 'GetNamedProperty' helps predict 'LdaGlobal' and 'Star' if they fall within the context window. The weights from the resulting hidden layer are used to form the embedding matrix. Key hyperparameters include the embedding dimension (vector size, e.g. 100) and the context window size (e.g. 3 or 5 bytecode instructions before and after). While computationally efficient, Word2Vec primarily captures local co-occurrence patterns and assigns the same vector regardless of the instruction's context in a specific sequence [23].
- **FastText** [28]: FastText enhances Word2Vec by representing each bytecode instruction not just as a whole unit but also as a bag of character n-grams. For a hypothetical bytecode instruction 'LdaConstant', FastText might consider n-grams like '<Ld', 'Lda', 'daC', 'aCo', ..., 'ant>', along with the full instruction '<LdaConstant>'. This allows the model to share representations between instructions with similar character patterns (e.g., 'LdaConstant' and 'LdaGlobal') and potentially generate better embeddings for rare instructions not frequently seen during training. Like Word2Vec, FastText produces static embeddings, but it requires more memory due to storing n-gram vectors.

These static embeddings serve as input for our baseline Random Forest classifier (Section 3.7). We determine optimal hyperparameters (vector dimension, window size, training epochs) through experimentation (details in Section 4).

Contextual Embeddings (Learned by Transformer). Our primary approach utilizes the Transformer architecture itself to learn dynamic, context-aware embeddings. Unlike static methods, the embedding representation for a bytecode instruction changes based on its position and the surrounding bytecodes within a specific function's bytecode sequence.

- **Input Representation:** We treat each unique V8 bytecode instruction (e.g., 'CreateObjectLiteral', 'LdaGlobal', 'GetNamedProperty', 'Return') as a token in our vocabulary. Since pretrained Transformer models are trained on natural language (e.g. BERT [12, 31], GPT [44]), their vocabularies and learned representations are unsuitable for bytecode instructions. Therefore, we initialize a token embedding matrix ($\mathbf{E}_{token} \in \mathbb{R}^{V \times d}$) randomly, where V is the vocabulary size (number of unique bytecode instructions) and d is the embedding dimension.
- **Embedding Calculation:** For a bytecode instruction i at position j in a sequence, its input vector is calculated as $\mathbf{x}_j = \mathbf{E}_{token}[i]$. We supplement these embeddings with sinusoidal position encoding [58].

- **Contextualization via Self-Attention:** This sequence of input vectors $(\mathbf{x}_1, ..., \mathbf{x}_L)$ is fed into a Transformer block [58]. The multi-head self-attention mechanism within the block allows each bytecode instruction's representation to be updated based on its relationship with all other bytecode instructions in the sequence. For example, the final representation for a 'GetNamedProperty' bytecode instruction will differ depending on whether it follows an 'LdaGlobal' (potentially accessing a global object property) or an 'LdaSmi' (an unlikely sequence, perhaps indicating unusual code). These learned embeddings are optimized end-to-end specifically for the fingerprinting classification task.

This approach allows the model to capture complex dependencies and nuances specific to bytecode sequences used in fingerprinting, offering a richer representation than static embeddings.

3.7 Fingerprinting Classification Models

Using the labeled bytecode sequences and their vector representations, we train classifiers to predict whether a given function exhibits fingerprinting behavior. We evaluate two distinct model architectures:

Random Forest (Baseline). We utilize Random Forest (RF) as a strong traditional machine learning baseline. RF is an ensemble method constructing multiple decision trees during training and making a final classification based on which class is predicted by the majority of the trees [40].

- **Input Features:** RF requires a fixed-size feature vector for each sample (function). We generate this by taking the static embeddings (Word2Vec or FastText, as described in Section 3.6) learned for each bytecode instruction and computing the element-wise average of the embedding vectors for all tokens present in a given function's bytecode sequence. This results in a single d-dimensional vector per function, where d is the embedding dimension.
- **Training:** The RF model is trained on these averaged vectors and their corresponding fingerprinting/non-fingerprinting labels. Key hyperparameters, such as the number of trees, maximum depth of each tree, criteria for splitting nodes (e.g., 'gini' or 'entropy'), and the number of features considered at each split, are tuned using techniques (e.g., GridSearchCV [40]) on a validation set. RF models offer relatively fast training and some level of interpretability through feature importance analysis, but the averaging process inherently loses sequential information present in the bytecode.

Transformer Classifier. Our primary model is a custom Transformer classifier, leveraging the architecture's proven success in sequence modeling [12, 58]. This deep learning approach processes the entire bytecode sequence directly, preserving order and context.

- **Architecture:** As illustrated in Figure 2 (right panel) and detailed below, our model follows a standard Transformer encoder structure adapted for classification:
  a) *Input Embeddings:* Summed learned token embeddings along with positional encoding for the bytecode sequence (Section 3.6).
  b) *Transformer Encoder Layer:* A multi-head self-attention sub-layer followed by a position-wise fully connected feed-forward network sub-layer. Residual connections and layer normalization are applied around each sub-layer [58]. This allows the model to progressively build richer contextual representations of the sequence.
  c) *Pooling:* A Global Average Pooling layer takes the output sequence from the final Transformer layer ($H \in \mathbb{R}^{L\times d}$) and computes the mean across the sequence length dimension, resulting in a single fixed-size vector ($h \in \mathbb{R}^{d}$) that summarizes the entire function's bytecode context.
  d) *Classification Head:* The pooled vector h is passed through one or more dense (fully connected) layers with ReLU activation functions and dropout [51] for regularization, before a final output layer consisting of a single neuron with a sigmoid activation function.

This end-to-end architecture (Figure 2, right panel) is trained specifically to optimize fingerprinting classification accuracy based on bytecode patterns. Performance comparisons are detailed in Section 5.

4 Experimental Setup and Dataset

This section details the experimental environment, the large-scale data collection process used to gather function-level bytecode and execution traces, the data cleaning and labeling pipeline, and the resulting dataset characteristics. We also describe the model architectures and hyperparameter tuning strategies employed for our classifiers.

4.1 Hardware and Crawling Infrastructure

All data collection was done on an Ubuntu 22.04 server equipped with an Intel Xeon Silver 4216 CPU and 376GB of RAM. Model training and testing was done with eight TPU v4 units, each with 16 GB of memory. Data collection utilized an automated web crawler built with Selenium WebDriver, controlling an instrumented version of the Chromium browser (version 123.0.6284.0, Section 3.1) augmented with our custom tracing extension (Section 3.2).

Crawling was conducted from a US-based university network IP address in June 2024. The crawler visited the homepages of the top 100,000 websites listed in the Tranco top-sites list [41], interacting with each page for 15 seconds to capture dynamic behavior before sending collected data (bytecode logs and serialized traces) to local servers.

4.2 Dataset Construction and Cleaning

The instrumented V8 engine generates bytecode sequences at the function-level, reflecting the compilation order and lazy execution semantics. Concurrently, our extension captures API execution traces.

Initial Data Collection. The raw data collection yielded bytecode and traces corresponding to approximately 21.1 million script instances across the 100k websites, comprising roughly 658 million function instances. This initial dataset contained significant noise, including functions from scripts with invalid URLs, anonymous functions lacking names, and functions executed using `eval`. Analysis of the raw execution traces showed that about 9.6M function instances (1.5% of total) invoked at least one high-entropy API potentially related to fingerprinting. Of these, 6.6M were anonymous functions (including 2,905 eventually labeled fingerprinting) and

3.1M were named functions (including 4,690 eventually labeled fingerprinting).

Data Cleaning and Label Mapping. To construct a clean dataset suitable for training and evaluation and to map execution trace labels to bytecode sequences, we performed several filtering steps using the methodology in Section 3.5:

(1) *URL Filtering:* Records associated with scripts having invalid or empty URLs were removed. This reduced the dataset to 9.4M scripts and 656M functions (7,595 fingerprinting functions).
(2) *Anonymous and Eval-Loaded Function Filtering:* As described in Section 3.5, our labeling process requires mapping dynamic execution traces to their corresponding static bytecode sequences using a key composed of script URL, script ID, and function name. Anonymous and eval-loaded functions lack a stable name, making reliable trace-to-bytecode mapping infeasible. We therefore exclude them from training dataset. This further reduced the dataset to 4.5M scripts (4,666 fingerprinting) and 407M functions (4,690 fingerprinting). Notably, this step removed a significant portion (~38%) of both potential fingerprinting and non-fingerprinting functions. While this exclusion limits the diversity of our training set, it does not impact deployment: ByteDefender relies solely on bytecode structure and does not require function names. As a result, it can still detect fingerprinting in anonymous or eval-loaded functions if their bytecode exhibits patterns learned from named functions during training.
(3) *Duplicate Removal:* Duplicate function records (i.e., bytecode sequences) were removed to prevent data contamination between training and testing sets. That is, for the function-level evaluation (Section 5.1) we removed all duplicate function bytecode sequences, whereas for the script-level evaluation (Section 5.2) we removed all duplicate script bytecode sequences. This resulted in the final function count.

The final processed dataset statistics are presented in Table 1. Each record contains the website domain, script ID, script URL, function name, the generated bytecode sequence, and its assigned fingerprinting/non-fingerprinting label derived from trace heuristics (Section 3.4).

Table 1: Distribution of non-fingerprinting (non-FP) and fingerprinting(FP) scripts and functions across the dataset after successive cleaning steps.

Cleaning Step	# Scripts		# Functions	
	FP	non-FP	FP	non-FP
Removing invalid/empty scripts	7,539	9,363,941	7,595	655,930,885
Removing anonymous functions	4,666	4,456,889	4,690	407,310,091
Removing repeated rows	4,666	4,456,889	4,670	404,877,008

Dataset Imbalance and Sampling Strategy. The final processed dataset exhibits a significant class imbalance, with non-fingerprinting (non-FP) functions vastly outnumbering fingerprinting (FP) functions (4,670 FP vs. approx. 405M non-FP), as shown in Table 1. This imbalance is anticipated, given that fingerprinting behavior, particularly involving the specific high-entropy APIs targeted by our labeling heuristics, is relatively infrequent compared to the bulk of general web script functionality. Training directly on such an imbalanced dataset would likely result in a classifier heavily biased towards the majority (non-FP) class, leading to poor detection of the minority (FP) class.

To address this imbalance and create a more effective training set for our supervised models, we employ a combined sampling strategy. We retain all identified positive examples (the 4,670 unique fingerprinting functions after cleaning and deduplication) to ensure the model learns from all available instances of fingerprinting behavior. To balance the classes for training, we then randomly undersample the vastly larger set of negative (non-FP) functions. Specifically, we select approximately 20 times the number of FP functions from the non-FP function pool (resulting in roughly 93,400 non-FP function samples). This creates the final training dataset with a FP:non-FP ratio of approximately 1:20. The full, original distribution (after cleaning and deduplication, but before training set balancing) is used for the test set to evaluate the model's performance under realistic conditions reflecting real-world class prevalence.

4.3 Dataset Analysis

To understand the characteristics of fingerprinting versus non-fingerprinting scripts and functions in our cleaned dataset, we analyze function counts per script and bytecode sequence lengths.

Functions per Script. Figure 3 shows the Cumulative Distribution Function (CDF) of the number of functions per script, comparing scripts labeled as fingerprinting (containing at least one fingerprinting function) versus non-fingerprinting scripts. A vast majority (90%) of non-fingerprinting scripts contain fewer than 100 functions. In contrast, scripts containing fingerprinting functions typically have significantly more functions, with the CDF rising sharply between 100 and 1,000 functions. This suggests that fingerprinting logic is often embedded within larger, more complex scripts that likely perform other tasks, reinforcing the need for function-level detection to avoid breaking legitimate functionality contained within the same script.

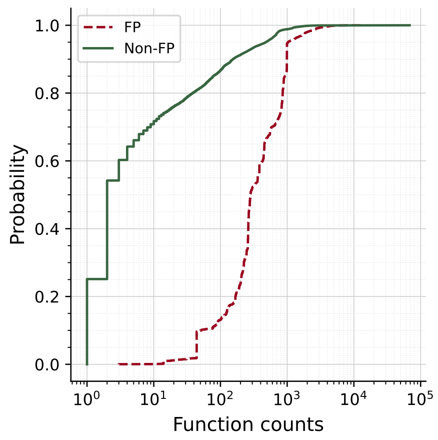


Figure 3: Cumulative distribution of the number of functions in fingerprinting (FP) versus non-fingerprinting (non-FP) scripts.

Mixed Scrips. We investigate the prevalence of mixed scripts in our dataset, where both legitimate functional and fingerprinting scripts are combined within a single script. Our analysis reveals that all 7,539 fingerprinting scripts identified are mixed scripts. Surprisingly, not a single script is composed entirely of functions labeled as fingerprinting. Therefore, blocking entire scripts based solely on the presence of one fingerprinting function would likely result in widespread disruption.

Bytecode Length. We also analyze the distribution of bytecode sequence lengths (number of bytecode instructions) for individual functions and concatenated script bytecode, as shown in Figure 4. The bytecode sequence of a script is constructed by concatenating the bytecode sequences of individual functions of the script together. A clear difference emerges: around 90% of non-fingerprinting functions have fewer than 100 bytecode instructions. Conversely, 90% of fingerprinting functions have bytecode lengths ranging between approximately 100 and 1,000 instructions. A similar, though less pronounced, trend holds for entire scripts. This indicates that functions performing fingerprinting tend to be computationally more complex or involve more operations than average non-fingerprinting functions, providing a potential signal for classification.

4.4 Implementation Details

Static Embeddings (Word2Vec/FastText). For baseline models requiring static embeddings, we explored Word2Vec [35] and FastText [28]. Recognizing that bytecode sequences are long but use a limited vocabulary compared to natural language, we prioritized computational efficiency by testing lower embedding dimensions (50, 100, 200) rather than the 300 often used for NLP [35]. We experimented with context window sizes (3 and 5) and training epochs (100 and 150) on the bytecode corpus derived from our training split. The optimal configuration for Word2Vec, based on downstream performance with the Random Forest classifier, includes an embedding dimension of 100 and a window size of 3. For FastText, these include an embedding dimension of 50 and a window size of 3.

Random Forest Configuration. We tuned the Random Forest classifier using GridSearchCV [40] on the validation set (derived from the balanced training data). Tested hyperparameters included split criterion ('gini', 'entropy'), max depth (10, 20, 30), max features ('sqrt', 'log2'), minimum samples per split (5, 10), and the number of estimator (100, 200). The optimal configuration for this classifier includes the following settings: criterion='entropy', max_depth=30, max_features='sqrt', min_samples_split=5, and n_estimators=200.

Transformer Architecture. Our custom Transformer model is implemented using JAX/TensorFlow [1, 21] and inspired by the original design [58] but adapted for bytecode classification. We determined the final architecture through experimentation on the validation set. The selected model for function classification employs:

- Input embedding dimension: 256.
- Transformer blocks: 1 layer.
- Attention heads: 4 heads per layer.
- Feed-forward dimension: 512 neurons in the hidden layer of the position-wise feed-forward networks within each block.
- Classification head: Global Average Pooling followed by two dense layers with ReLU activation, interspersed with dropout layers (rate 0.1), and a final sigmoid output neuron.

The selected model for script classification employs:

- Input embedding dimension: 128.
- 1D Convolutional Layer (Optional): A 1D convolution with kernel size and stride of 2 along the sequence axis, this reduces the sequence length for attention and saves memory.
- Transformer blocks: 1 layer.
- Attention heads: 4 heads per layer.
- Feed-forward dimension: 256 neurons in the hidden layer of the position-wise feed-forward networks within each block.
- Classification head: Same as function model.

These configurations provided the best trade-off between performance and complexity in our experiments.

Transformer Training Details: For function classification, the Transformer model was trained for 16 epochs on the balanced training set using the Adam optimizer [32] with binary cross-entropy loss. We employed a batch size of 128.

For script classification, which has more memory requirements due to the longer sequence length, we employed a batch size of 8 and 16. We trained the model for 10 epochs, since the script classification dataset was larger. We found these hyperparameters empirically to offer a good balance between model complexity and generalization based on validation performance [51].

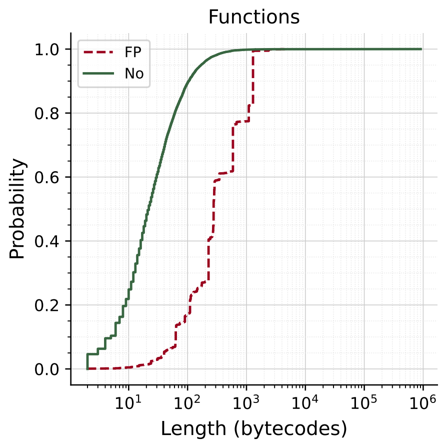
5 Evaluation

In this section, we evaluate the effectiveness of ByteDefender in detecting fingerprinting functions at the JavaScript function level, and its overhead when browsing the web. We assess our primary Transformer-based classifier and compare it against a Random Forest baseline, using the dataset and methodologies detailed in Sections 4 and 3. Specifically, we compare the models according to their accuracy, precision, and recall. Furthermore, we evaluate ByteDefender's script-level robustness against code obfuscation.

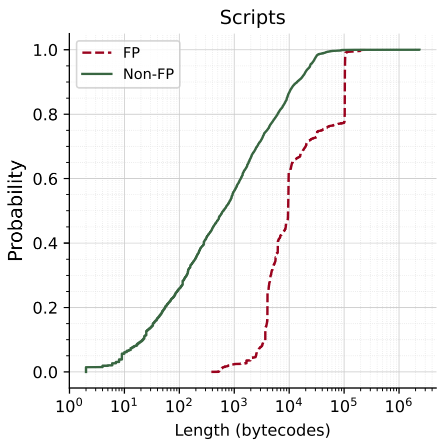
5.1 Function-Level Classification Performance

Experimental Setup: We used the processed dataset described in Section 4, derived from crawling 100k websites. The data was split into training (90%) and testing (10%) sets, ensuring no duplicate function bytecode sequences were found across splits. Due to class imbalance, the training set was balanced by undersampling the non-fingerprinting class to a 1:10 ratio (FP:non-FP) after oversampling positive examples. For the Random Forest baseline, Word2Vec and FastText embeddings were trained on the training set's bytecode corpus, and the resulting averaged function vectors were used for training the RF model. For the Transformer classifier, embeddings were learned end-to-end during training on the bytecode sequences.

Results. Table 2 presents the core classification results on the test set. Our Transformer classifier significantly outperforms the Random Forest baselines across most metrics, achieving an accuracy of 98.9%, precision of 84.0%, and recall of 85.1%. While Random Forest with FastText embeddings achieves high precision (94.8%), its recall is considerably lower (66.3%), indicating it misses a substantial portion of fingerprinting functions. Word2Vec embeddings yield even lower performance for the Random Forest model. The



(a) Functions



(b) Scripts

Figure 4: Comparative distribution of bytecode lengths for functions (a) and scripts (b).

Transformer's superior performance, especially in recall, highlights the benefit of processing the full bytecode sequence and leveraging contextual embeddings learned end-to-end, compared to using averaged static embeddings which lose sequential information.

The choice between precision and recall depends on the deployment goal. High recall, as achieved by the Transformer, is crucial if the priority is to minimize missed fingerprinting instances (false negatives), even at the cost of some false positives. High precision, where RF+FastText excels, is important if minimizing disruption to non-fingerprinting functions (false positives) is paramount. The high overall accuracy of the Transformer suggests it provides the best balance for effectively detecting the majority of fingerprinting functions present in the wild.

Table 2: Model performance for function classification using different classifiers and embeddings on the test set.

Classifier	**Embed. Model**	**Acc. (%)**	**Prec. (%)**	**Recall (%)**	**ROC AUC (%)**	**PR AUC (%)**
Transformer	Transf. (learned)	98.9	84.0	85.1	93.3	81.6
Random Forest	Word2Vec	93.6	80.0	60.0	79.0	60.7
	FastText	94.8	94.8	66.3	83.1	67.6

5.2 Comparison to State-of-the-Art (Script-Level)

To compare ByteDefender with prior work that operates at the script-level, we adapted our approach. As existing methods primarily detect fingerprinting based on entire scripts [27, 46, 56], we created a script-level version of our dataset by concatenating the bytecode of all functions within a script and labeling the script as FP if it contained at least one fingerprinting function. We then trained our Transformer architecture on this script-level data.

We compare this against a reimplementation of the static AST-based approach, representative of methods used in studies by Iqbal et al. [27], Rizzo [46], and van Zalingen and Haanen [56]. Reconstructing this model involved using the script source code collected during our crawls (Section 3.3) and employing the Esprima library's 'parseScript' function [18] to convert each script's source code into its corresponding AST. We then traversed these ASTs hierarchically, segmenting them into pairs of parent and child nodes. In this representation, parent nodes capture contextual structures (such as 'for' loops, 'try' statements, or 'if' conditions), while child nodes represent the specific functions or operations executed within those contexts (for instance, API calls such as `createElement`, `toDataURL`, or `measureText`). To create input vectors for classification, we encoded the presence of these parent-child pairs using one-hot vectors, where each vector position corresponds to a unique potential parent-child combination observed across the dataset. Within a script's vector, a value of '1' at a specific position indicates that the corresponding parent-child combination exists in its AST, while a '0' signifies its absence. Finally, following the methodology described for the static component in FP-Inspector [27], we used these one-hot feature vectors representing the script's AST patterns as input to train a standard decision tree classifier.

Table 3 shows the results of this script-level comparison using the test set derived from our dataset. ByteDefender's Transformer, adapted to the script-level, demonstrates exceptionally high performance across all metrics: 99.7% accuracy, 92.1% precision, and 96.9% recall. The reimplemented AST-based classifier achieves considerably lower performance, with 97.5% accuracy, 85.0% precision, and only 80.0% recall. The substantial difference, particularly the nearly 17% higher recall achieved by ByteDefender, strongly underscores the superiority of the bytecode-based approach compared to AST analysis for script-level fingerprinting detection. Bytecode captures semantics closer to execution and is less susceptible to syntactic variations and obfuscation, leading to significantly better identification of true positive fingerprinting scripts.

Table 3: Comparative performance of ByteDefender (Transformer, script-level) and representative AST-based classifiers in script classification.

Method	Acc. (%)	Prec. (%)	Recall (%)	ROC AUC (%)	PR AUC (%)
ByteDefender (Transformer, script-level)	99.7	92.1	96.9	99.5	92.3
AST-based classifiers from prior work (Random-Forest, script-level)	97.5	85.0	80.0	90.0	68.0

5.3 Robustness Evaluation

A key advantage claimed for bytecode analysis is its robustness against common evasion techniques. We evaluate this against URL manipulation and code obfuscation.

Robustness against URL manipulations. As ByteDefender analyzes function bytecode content directly and does not rely on the script's origin URL or domain for its classification decision (unlike filter lists or some features in [27]), it is inherently robust against URL-based evasion techniques such as CNAME cloaking [9, 13] or path/parameter randomization [60]. This allows ByteDefender to detect fingerprinting behavior regardless of how or where the script is hosted.

Robustness against code obfuscation. Fingerprinting scripts often employ obfuscation techniques to complicate static inspection and evade detection [50]. As we discussed in Section 3.1, our model operates on bytecode sequences that retain only the opcode instructions, omitting auxiliary details such as memory addresses, operands, and bytecode offsets. This abstraction aligns with our goal of building a generalizable and obfuscation-resilient detection approach. Common obfuscation strategies—such as string encoding, variable renaming, and control flow flattening—primarily target operands and syntactic constructs, while typically preserving the underlying operational behavior reflected in opcode instructions [50]. At the function-level, V8 bytecode is relatively robust to obfuscation, since the compiled bytecode retains semantic structure even when the source code is heavily transformed. However, at the script-level—where bytecode sequences represent full-page execution contexts—robustness degrades due to non-deterministic factors such as script loading order, lazy compilation, and network timing [6]. These factors can cause significant variation even across semantically identical scripts.

To evaluate the impact of obfuscation at the script level, we assess ByteDefender's performance—which is trained on real-world scripts consisting of a mixture of unobfuscated and obfuscated code (Section 5.2)—on re-obfuscated scripts. To generate obfuscated versions of collected scripts, we use two distinct and publicly available tools: JavaScript-Obfuscator [29], which applies heavy transformations such as variable renaming, string encoding, and control flow flattening; and the Google Closure Compiler [24], which focuses on code optimization through renaming and simplification. We then use our instrumented V8 engine (Section 3.1) to extract bytecode sequences from these obfuscated scripts. The evaluation includes 47,000 JavaScript-Obfuscator samples and 11,000 Google Closure obfuscated samples, each preserving a 1:20 ratio of fingerprinting to non-fingerprinting scripts. As shown in the first row of Table 4, ByteDefender, when trained only on real-world scripts, performs poorly on re-obfuscated inputs. For example, when obfuscating with JavaScript Obfuscator, precision remains moderate (60.5%), but recall falls to 0.1%, indicating a near-total failure to detect fingerprinting scripts. A similar trend is observed with the Google Closure Compiler, where recall is only 2.8%.

To mitigate this vulnerability, we train ByteDefender's script-level model on a an augmented dataset that combines real-world and re-obfuscated scripts. Our final training set consists of approximately 100,000 scripts with a 1:20 fingerprinting-to-non-fingerprinting ratio, augmented with the aforementioned obfuscated samples. The model is then evaluated on a balanced set of previously unseen real-world and re-obfuscated samples. As shown in the last two rows of Table 4, augmented ByteDefender significantly improves generalization. For instance, recall jumps from 0.1% to 92.1% with JavaScript Obfuscator and from 2.8% to 78.0% with Google Closure. These results demonstrate that exposing the model to diverse obfuscation styles during training enables it to generalize effectively and detect fingerprinting behavior in obfuscated scripts.

Table 4: Evaluation of script-level ByteDefender and its augmented variant on scripts obfuscated using various techniques.

Classifier	Obfuscator	Acc. (%)	Prec. (%)	Recall (%)
ByteDefender	JavaScript Obfuscator	55.0	60.5	0.1
	Google Closure	56.8	93.8	2.8
Augmented ByteDefender	JavaScript Obfuscator	89.1	85.0	92.1
	Google Closure	89.9	82.9	78.0

5.4 Real-Time Detection Overhead in the Browser

Next, we assess the runtime overhead introduced by enabling on-device signature matching against JavaScript function bytecodes.

To detect fingerprinting behaviors, we first generate a list of non-cryptographic hashes corresponding to known fingerprinting function bytecodes and embed these signatures into the V8 engine source code. Within `interpreter.cc`, we call the `Disassembly()` method from the `BytecodeArray` class to translate raw binary bytecode into its symbolic instruction format. To manage the verbosity of the output, we apply the same selective instrumentation approach detailed in Section 3.1. We further modify V8 to compute the hash of each function's bytecode sequence at runtime and compare it against the embedded signature list to detect fingerprinting functions during execution.

To quantify the overhead, we crawl 1,000 websites using our modified Chromium browser with lightweight signature matching enabled and compare its performance against an unmodified baseline Chromium build. Figure 5 illustrates the distribution of page load times for both the baseline and instrumented Chromium across 1,000 websites. As shown, the CDF curves for the two versions are closely aligned. Our measurements show that the signature matching mechanism introduces an average additional latency of 158.74ms, which represents only a 4% increase of the overall median page load time in our sample. The distribution of overhead is tightly bounded, with the 5th percentile at 117.54ms, 25th percentile at 146.22ms, median at 158.62ms, 75th percentile at 170.30ms, and 95th percentile at 199.21ms. These results demonstrate that the runtime cost of on-device bytecode matching remains consistently low, introducing negligible impact on overall browser performance while preventing fingerprinting functions—and *not* entire scripts—from being executed.

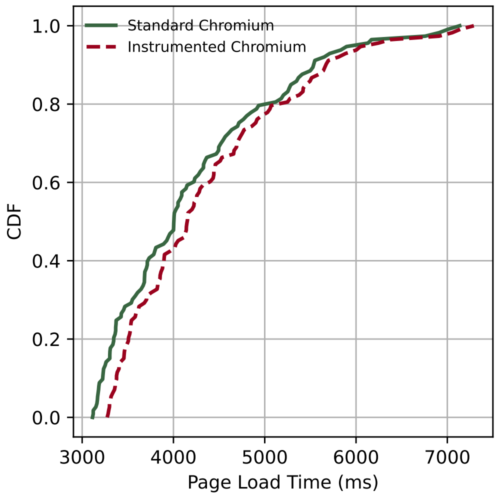


Figure 5: CDF plot comparing page load times of the baseline (standard Chromium) and our instrumented Chromium with signature matching, across 1,000 websites.

6 Limitations and Future Work

While ByteDefender demonstrates strong performance and robustness against common evasion techniques, we acknowledge several limitations inherent in our current methodology and dataset construction, alongside potential avenues for future research.

6.1 Impact of Anonymous Functions and Scripts

Our approach to generating ground truth labels necessitates mapping function execution traces to their corresponding bytecode sequences using script URL, script ID, and function name (Section 3.5). This requirement presented a significant challenge, as anonymous functions (lacking names) or scripts without canonical URLs (e.g., those loaded via ‘eval‘ or complex dynamic means) could not be reliably mapped and were consequently excluded during the data cleaning phase (Section 4.2). As shown in Table 1, this filtering step removed a substantial number of functions, including a large fraction of potential fingerprinting instances initially identified by high-entropy API usage. This necessary exclusion limits the diversity of functions represented in our training set, potentially affecting the model's ability to generalize to websites heavily reliant on anonymous functions or unconventional script loading patterns.

However, it is important to clarify the scope of this limitation. It primarily affects the generation of labeled training data. The ByteDefender Transformer model itself, once trained, analyzes bytecode structure and does not inherently require function names or script URLs for classification. It learns bytecode patterns associated with fingerprinting. Therefore, the model may still successfully classify anonymous functions or inline scripts during deployment if their bytecode exhibits patterns similar to those learned from the named functions in the training set. Techniques like code inlining (embedding scripts directly in HTML) are not necessarily an evasion strategy against ByteDefender's core detection capability, as the V8 engine still generates bytecode for these inline script blocks which can then be analyzed. However, we acknowledge that ByteDefender's function-level classifier has not been explicitly evaluated on anonymous or eval-loaded functions, and its generalization to such cases remains unverified.

Future work could focus on developing alternative ground truth generation methods less reliant on explicit identifiers, perhaps incorporating dynamic analysis techniques for associating traces with anonymous functions or employing code similarity metrics on bytecode to propagate labels.

6.2 Reliance on Heuristic-Based Ground Truth

We established the ground truth for this study using heuristics based on well-documented fingerprinting techniques involving specific high-entropy API call patterns (Section 3.4), drawing from established research [17, 27]. This approach was necessary due to the infeasibility of manually labeling the millions of functions collected, and it provides a high-precision dataset for training, aligning with practices in related work [17, 27]. Our evaluation (Section 5) confirms that models trained on this data achieve high accuracy for known fingerprinting methods. However, this reliance on predefined heuristics introduces an inherent bias: the heuristics are tailored to known fingerprinting techniques and prioritize precision. As a result, they may fail to capture novel or evolving fingerprinting behaviors that deviate from documented patterns or utilize different API sequences [3]. Addressing this requires ongoing research. Exploring semi-supervised or unsupervised learning methods capable of identifying anomalous or suspicious bytecode patterns directly, without depending on predefined API lists, could enable the detection of emerging fingerprinting strategies and represents a promising direction for future work.

6.3 Browser and Engine Dependency

ByteDefender's current implementation is specifically tailored to Chromium's V8 engine. We leverage V8's bytecode instruction set as the feature representation for our classifiers. Because JavaScript bytecode is not standardized across different browser engines (such as SpiderMonkey in Firefox or JavaScriptCore in Safari), each having its own distinct instruction set and compilation pipeline, a model trained on V8 bytecode is unlikely to function correctly on bytecode generated by other engines *without significant adaptation.* Extending ByteDefender's approach to achieve cross-browser compatibility would necessitate instrumenting and *collecting data from each target engine and training engine-specific models.*

Furthermore, bytecode instruction sets can evolve even within the same engine (V8) across different versions, driven by new JavaScript language features or internal optimizations. This implies that maintaining optimal performance for ByteDefenderover time may require periodic retraining using bytecode collected from updated V8 versions. While the core methodology remains valid, this potential need for retraining represents an ongoing maintenance consideration.

Future research should explore techniques for cross-engine bytecode translation or the identification of higher-level, potentially engine-invariant features derivable from bytecode to enhance model portability across browsers and versions.

7 Conclusion

Browser fingerprinting presents a persistent and evolving challenge to user privacy, leveraging subtle browser and device characteristics to enable cross-site tracking often immune to conventional defenses. Existing countermeasures frequently fall short, either lacking precision and causing website breakage (e.g., script-level blocking) or proving vulnerable to common evasion tactics like code obfuscation and URL manipulation (e.g., filter lists, AST-based analysis). Addressing this requires a robust, precise, and proactive detection mechanism.

In this paper, we introduced ByteDefender, the first system to utilize V8 engine bytecode specifically for detecting fingerprinting behaviors at the JavaScript function level. By analyzing the intrinsic bytecode patterns of individual functions during compilation, ByteDefender achieves fine-grained detection before execution occurs. We demonstrated the efficacy of a Transformer-based classifier trained on function-level bytecode sequences derived from a large-scale crawl of 100k websites, where ground truth was established using heuristic analysis of execution traces.

Our evaluation confirms the advantages of this approach. ByteDefender achieves high detection accuracy (99.7% at the script level), precision (92.1%), and recall (96.9%), significantly outperforming representative AST-based methods, especially demonstrating superior robustness against obfuscated JavaScript code. Its function-level granularity is crucial, given our finding that fingerprinting scripts are typically mixed-purpose, mitigating the web breakage associated with coarser script-level blocking. We also demonstrated the feasibility of efficient on-device deployment through preliminary measurements showing that lightweight signature matching introduces only negligible (average 4%) page load latency.

ByteDefender offers a practical and effective framework for fingerprinting mitigation. It moves beyond the limitations of existing methods by providing a robust, precise, function-level analysis grounded in the code representation used by the JavaScript engine itself. By enabling targeted, pre-execution intervention, it enhances user privacy with minimal impact on web compatibility. Future work includes refining signature generation for minimal overhead and exploring cross-browser compatibility.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems.

[2] Abdul Haddi Amjad, Shaoor Munir, Zubair Shafiq, and Muhammad Ali Gulzar. 2024. Blocking Tracking JavaScript at the Function Granularity. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2177–2191.

[3] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2021. Fp-radar: Longitudinal measurement and early detection of browser fingerprinting. *arXiv preprint arXiv:2112.01662* (2021).

[4] bravemitigations 2018. Fingerprinting Protection Mode. https://github.com/brave/browser-laptop/wiki/Fingerprinting-Protection-Mode.

[5] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. 2016. Picasso: Lightweight device class fingerprinting for web clients. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. 93–102.

[6] Javier Cabrera Arteaga, Martin Monperrus, and Benoit Baudry. 2019. Scalable comparison of JavaScript V8 bytecode traces. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 22–31.

[7] Dell Cameron. 2018. Apple Declares War on Browser Fingerprinting, the Sneaky Tactic That Tracks You in Incognito Mode. https://gizmodo.com/apple-declareswar-on-browser-fingerprinting-the-sneak-1826549108. Accessed: 2025.

[8] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-) browser fingerprinting via OS and hardware level features. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society.

[9] Ha Dao, Johan Mazel, and Kensuke Fukuda. 2021. CNAME cloaking-based tracking on the web: Characterization, detection, and protection. *IEEE Transactions on Network and Service Management* 18, 3 (2021), 3873–3888.

[10] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1515–1532.

[11] Amit Datta, Jianan Lu, and Michael Carl Tschantz. 2018. The effectiveness of privacy enhancing technologies against fingerprinting. *arXiv preprint arXiv:1812.03920* (2018).

[12] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[13] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom van Goethem. 2021. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *Proceedings on Privacy Enhancing Technologies* 2021 (2021), 394 – 412. https://api.semanticscholar.org/CorpusID:231951672

[14] Disconnect 2024. Disconnect tracking protection lists. https://disconnect.me/trackerprotection. Accessed: 2025.

[15] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2021. FP-Redemption: Studying browser fingerprinting adoption for the sake of web security. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th International Conference, DIMVA 2021, Virtual Event, July 14–16, 2021, Proceedings 18*. Springer, 237–257.

[16] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.

[17] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1388–1401.

[18] esprima_parsing [n. d.]. Syntactic Analysis (Parsing). https://docs.esprima.org/en/latest/syntactic-analysis.html. Accessed: 2025.

[19] David Fifield and Mia Gil Epner. 2016. Fingerprintability of webrtc. *arXiv preprint arXiv:1605.08805* (2016).

[20] firefoxAntiFingerpriting [n. d.]. How to block fingerprinting with Firefox. https://blog.mozilla.org/firefox/how-to-block-fingerprinting-with-firefox. Accessed: 2025.

[21] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[22] Mohammad Ghasemisharif and Jason Polakis. 2023. Read between the lines: Detecting tracking javascript with bytecode classification. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3475–3489.

[23] Yoav Goldberg. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722* (2014).

[24] google_closure [n. d.]. Google Closure compiler. https://developers.google.com/closure/compiler. Accessed: 2025.

[25] high_entropy_apis [n. d.]. High Entropy APIs flagged by chromium. https://github.com/chromium/chromium/blob/aae7191b27cef1f097b23e7742afb4895ec6a9d3/docs/privacy_budget/privacy_budget_instrumentation.md?plain=1#L196. Accessed: 2025.

[26] Franziska Hinkelmann. 2017. Understanding V8's Bytecode. https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775. Accessed: 2025.

[27] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1143–1161.

[28] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759* (2016).

[29] Timofey Kachalov. [n. d.]. Javascript obfuscator tool. https://obfuscator.io/. Accessed: 2025.
[30] Faezeh Kalantari, Mehrnoosh Zaeifi, Yeganeh Safaei, Marzieh Bitaab, Adam Oest, Gianluca Stringhini, Yan Shoshitaishvili, and Adam Doupé. 2024. Browser Polygraph: Efficient Deployment of Coarse-Grained Browser Fingerprints for Web-Scale Detection of Fraud Browsers. In *Proceedings of the 2024 ACM on Internet Measurement Conference*. 681–703.
[31] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, Vol. 1.
[32] Diederik P Kingma. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[33] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)* 14, 2 (2020), 1–33.
[34] Zengrui Liu, Jimmy Dani, Yinzhi Cao, Shujiang Wu, and Nitesh Saxena. 2025. The First Early Evidence of the Use of Browser Fingerprinting for Online Tracking. In *THE WEB CONFERENCE 2025*.
[35] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[36] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically detecting javascript obfuscation and minification techniques in the wild. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 569–580.
[37] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *Proceedings of W2SP* 2012 (2012).
[38] Ray Ngan, Surya Konkimalla, and Zubair Shafiq. 2022. Nowhere to hide: Detecting obfuscated fingerprinting scripts. *arXiv preprint arXiv:2206.13599* (2022).
[39] Tom Ritter Nick Doty. 2025. W3C Fingerprinting Guidance. https://w3c.github.io/fingerprinting-guidance/. Accessed: 2025.
[40] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
[41] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2018. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156* (2018).
[42] privacybadger [n. d.]. Privacy Badger. https://www.eff.org/privacybadger. Accessed: 2025.
[43] Gaston Pugliese, Christian Riess, Freya Gassmann, and Zinaida Benenson. 2020. Long-term observation on browser fingerprinting: Users' trackability and perspective. *Proceedings on Privacy Enhancing Technologies* (2020).
[44] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *OpenAI blog* (2018).
[45] Andreas Reiter and Alexander Marsalek. 2017. WebRTC: your privacy is at risk. In *Proceedings of the Symposium on Applied Computing*. 664–669.
[46] Valentino Rizzo. 2018. *Machine Learning Approaches for Automatic Detection of Web Fingerprinting*. Ph. D. Dissertation. Politecnico di Torino.
[47] Muhammad Fakhrur Rozi, Sangwook Kim, and Seiichi Ozawa. 2020. Deep neural networks for malicious javascript detection using bytecode sequences. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.
[48] selenium [n. d.]. Selenium WebDriver. https://www.selenium.dev/documentation/webdriver/. Accessed: 2025.
[49] Asuman Senol, Alisha Ukani, Dylan Cutler, and Igor Bilogrevic. 2024. The double edged sword: identifying authentication pages and their fingerprinting behavior. In *Proceedings of the ACM Web Conference 2024*. 1690–1701.
[50] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to hide? studying minified and obfuscated code in the web. In *The world wide web conference*. 1735–1746.
[51] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.
[52] Brave Privacy Team. 2020. Fingerprint randomization. https://brave.com/privacy-updates/3-fingerprint-randomization/. Accessed: 2025.
[53] V8 team. 2017. Launching Ignition and TurboFan. https://v8.dev/blog/launching-ignition-and-turbofan. Accessed: 2025.
[54] tormitigations [n. d.]. Fingerprinting Defenses in The Tor Browser. https://2019.www.torproject.org/projects/torbrowser/design/#fingerprinting-defenses. Accessed: 2025.
[55] tracing_CDP [n. d.]. Tracing domain in Chrome DevTools protocol. https://chromedevtools.github.io/devtools-protocol/tot/Tracing/. Accessed: 2025.
[56] Tim van Zalingen and Sjors Haanen. 2018. Detection of Browser Fingerprinting by Static JavaScript Code Classification.
[57] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. {Fp-Scanner}: The privacy implications of browser fingerprint inconsistencies. In *27th USENIX Security Symposium (USENIX Security 18)*. 135–150.
[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[59] w3mitigations [n. d.]. Mitigating Browser Fingerprinting in Web Specifications. https://www.w3.org/TR/fingerprinting-guidance/?utm_source=chatgpt.com#narrow-scope-availability. Accessed: 2025.
[60] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 205–216.
[61] webkitBlocking [n. d.]. Tracking Prevention in WebKit. https://webkit.org/tracking-prevention/. Accessed: 2025.
[62] Mike West. 2024. Combating Fingerprinting with a Privacy Budget Explainer. https://github.com/bslassey/privacy-budget. Accessed: 2025.
[63] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. 2019. Rendered private: Making {GLSL} execution uniform to prevent {WebGL-based} browser fingerprinting. In *28th USENIX Security Symposium (USENIX Security 19)*. 1645–1660.

A List of high entropy APIs

The following table lists 322 features, consisting exclusively of API call counts that have been flagged as 'high entropy' by Chromium.

Table 5: List of APIs in Chrome explicitly flagged by Chromium as 'High Entropy APIs'.

High Entropy APIs		
AnalyserNode.getByteFrequencyData	NavigatorUAData.platform.get	RTCIceCandidate.candidate.get
AnalyserNode.getByteTimeDomainData	NavigatorUAData.toJSON	RTCIceCandidate.port.get
AnalyserNode.getFloatFrequencyData	Navigator.userAgent.get	RTCIceCandidate.relatedAddress.get
AnalyserNode.getFloatTimeDomainData	Navigator.vendor.get	RTCIceCandidate.relatedPort.get
AudioBuffer.copyFromChannel	Navigator.vendorSub.get	RTCRtpReceiver.getCapabilities
AudioBuffer.getChannelData	Navigator.webkitGetUserMedia	RTCRtpSender.getCapabilities
AudioContext.baseLatency.get	NetworkInformation.downlink.get	Screen.availHeight.get
AudioContext.constructor	NetworkInformation.downlinkMax.get	Screen.availLeft.get
AudioContext.outputLatency.get	NetworkInformation.effectiveType.get	Screen.availTop.get
AudioNode.connect	NetworkInformation.rtt.get	Screen.availWidth.get
AuthenticatorAttestationResponse.getTransports	NetworkInformation.saveData.get	Screen.colorDepth.get
BackgroundFetchRegistration.failureReason.get	NetworkInformation.type.get	ScreenDetailed.devicePixelRatio.get
BaseAudioContext.createDynamicsCompressor	OfflineAudioContext.constructor	ScreenDetailed.isInternal.get
BaseAudioContext.createOscillator	OfflineAudioContext.startRendering	ScreenDetailed.isPrimary.get
BaseAudioContext.sampleRate.get	OffscreenCanvas.convertToBlob	ScreenDetailed.label.get
BatteryManager.charging.get	OffscreenCanvasRenderingContext2D.arc	ScreenDetailed.left.get
BatteryManager.chargingTime.get	OffscreenCanvasRenderingContext2D.beginPath	ScreenDetailed.top.get
BatteryManager.dischargingTime.get	OffscreenCanvasRenderingContext2D.clearRect	ScreenDetails.oncurrentscreenchange.get
BatteryManager.level.get	OffscreenCanvasRenderingContext2D.clip	ScreenDetails.oncurrentscreenchange.set
BeforeInstallPromptEvent.platforms.get	OffscreenCanvasRenderingContext2D.closePath	ScreenDetails.onscreenschange.get
BluetoothAdvertisingEvent.appearance.get	OffscreenCanvasRenderingContext2D.drawMesh	ScreenDetails.onscreenschange.set
BluetoothAdvertisingEvent.name.get	OffscreenCanvasRenderingContext2D.ellipse	Screen.height.get
BluetoothAdvertisingEvent.txPower.get	OffscreenCanvasRenderingContext2D.fill	Screen.isExtended.get
BluetoothDevice.name.get	OffscreenCanvasRenderingContext2D.fillRect	Screen.onchange.get
CanvasRenderingContext2D.arc	OffscreenCanvasRenderingContext2D.fillStyle.get	Screen.onchange.set
CanvasRenderingContext2D.beginPath	OffscreenCanvasRenderingContext2D.fillStyle.set	ScreenOrientation.angle.get
CanvasRenderingContext2D.clearRect	OffscreenCanvasRenderingContext2D.fillText	ScreenOrientation.type.get
CanvasRenderingContext2D.closePath	OffscreenCanvasRenderingContext2D.font.get	Screen.pixelDepth.get
CanvasRenderingContext2D.drawMesh	OffscreenCanvasRenderingContext2D.font.set	Screen.width.get
CanvasRenderingContext2D.ellipse	OffscreenCanvasRenderingContext2D.getImageData	SVGAnimationElement.requiredExtensions.get
CanvasRenderingContext2D.fill	OffscreenCanvasRenderingContext2D.globalCompositeOperation.get	SVGAnimationElement.systemLanguage.get
CanvasRenderingContext2D.fillRect	OffscreenCanvasRenderingContext2D.globalCompositeOperation.set	SVGGeometryElement.getPointAtLength
CanvasRenderingContext2D.fillStyle.get	OffscreenCanvasRenderingContext2D.isPointInPath	SVGGeometryElement.getTotalLength
CanvasRenderingContext2D.fillStyle.set	OffscreenCanvasRenderingContext2D.isPointInStroke	SVGGeometryElement.isPointInFill
CanvasRenderingContext2D.fillText	OffscreenCanvasRenderingContext2D.lineTo	SVGGeometryElement.isPointInStroke
CanvasRenderingContext2D.font.get	OffscreenCanvasRenderingContext2D.measureText	SVGGraphicsElement.requiredExtensions.get
CanvasRenderingContext2D.font.set	OffscreenCanvasRenderingContext2D.moveTo	SVGGraphicsElement.systemLanguage.get
CanvasRenderingContext2D.getImageData	OffscreenCanvasRenderingContext2D.rect	SVGMaskElement.requiredExtensions.get
CanvasRenderingContext2D.globalCompositeOperation.get	OffscreenCanvasRenderingContext2D.rotate	SVGMaskElement.systemLanguage.get
CanvasRenderingContext2D.globalCompositeOperation.set	OffscreenCanvasRenderingContext2D.roundRect	SVGPatternElement.requiredExtensions.get
CanvasRenderingContext2D.isPointInPath	OffscreenCanvasRenderingContext2D.scale	SVGPatternElement.systemLanguage.get
CanvasRenderingContext2D.isPointInStroke	OffscreenCanvasRenderingContext2D.setTransform	SVGTextContentElement.getComputedTextLength
CanvasRenderingContext2D.lineTo	OffscreenCanvasRenderingContext2D.shadowBlur.get	SVGTextContentElement.getEndPositionOfChar
CanvasRenderingContext2D.measureText	OffscreenCanvasRenderingContext2D.shadowBlur.set	SVGTextContentElement.getExtentOfChar
CanvasRenderingContext2D.moveTo	OffscreenCanvasRenderingContext2D.shadowColor.get	SVGTextContentElement.getStartPositionOfChar
CanvasRenderingContext2D.rect	OffscreenCanvasRenderingContext2D.shadowColor.set	SVGTextContentElement.getSubStringLength
CanvasRenderingContext2D.rotate	OffscreenCanvasRenderingContext2D.shadowOffsetX.get	Touch.force.get

High Entropy APIs (continued)		
CanvasRenderingContext2D.roundRect	OffscreenCanvasRenderingContext2D.shadowOffsetX.set	VisualViewport.height.get
CanvasRenderingContext2D.scale	OffscreenCanvasRenderingContext2D.shadowOffsetY.get	VisualViewport.offsetLeft.get
CanvasRenderingContext2D.setTransform	OffscreenCanvasRenderingContext2D.shadowOffsetY.set	VisualViewport.offsetTop.get
CanvasRenderingContext2D.shadowBlur.get	OffscreenCanvasRenderingContext2D.stroke	VisualViewport.pageLeft.get
CanvasRenderingContext2D.shadowBlur.set	OffscreenCanvasRenderingContext2D.strokeRect	VisualViewport.pageTop.get
CanvasRenderingContext2D.shadowColor.get	OffscreenCanvasRenderingContext2D.strokeStyle.get	VisualViewport.scale.get
CanvasRenderingContext2D.shadowColor.set	OffscreenCanvasRenderingContext2D.strokeStyle.set	VisualViewport.width.get
CanvasRenderingContext2D.shadowOffsetX.get	OffscreenCanvasRenderingContext2D.strokeText	WebGL2RenderingContext.getExtension
CanvasRenderingContext2D.shadowOffsetX.set	OffscreenCanvasRenderingContext2D.transform	WebGL2RenderingContext.getInternalformatParameter
CanvasRenderingContext2D.shadowOffsetY.get	OffscreenCanvasRenderingContext2D.translate	WebGL2RenderingContext.getParameter
CanvasRenderingContext2D.shadowOffsetY.set	OffscreenCanvas.transferToImageBitmap	WebGL2RenderingContext.getRenderbufferParameter
CanvasRenderingContext2D.stroke	PaintRenderingContext2D.arc	WebGL2RenderingContext.getShaderPrecisionFormat
CanvasRenderingContext2D.strokeRect	PaintRenderingContext2D.beginPath	WebGL2RenderingContext.getSupportedExtensions
CanvasRenderingContext2D.strokeStyle.get	PaintRenderingContext2D.clearRect	WebGL2RenderingContext.makeXRCompatible
CanvasRenderingContext2D.strokeStyle.set	PaintRenderingContext2D.closePath	WebGLCompressedTextureASTC.getSupportedProfiles
CanvasRenderingContext2D.strokeText	PaintRenderingContext2D.drawMesh	WebGLRenderingContext.getExtension
CanvasRenderingContext2D.transform	PaintRenderingContext2D.ellipse	WebGLRenderingContext.getParameter
CanvasRenderingContext2D.translate	PaintRenderingContext2D.fill	WebGLRenderingContext.getRenderbufferParameter
FeaturePolicy.features	PaintRenderingContext2D.fillRect	WebGLRenderingContext.getShaderPrecisionFormat
Gamepad.id.get	PaintRenderingContext2D.fillStyle.get	WebGLRenderingContext.getSupportedExtensions
GPUAdapterInfo.architecture.get	PaintRenderingContext2D.fillStyle.set	WebGLRenderingContext.makeXRCompatible
GPUAdapterInfo.description.get	PaintRenderingContext2D.globalCompositeOperation.get	WheelEvent.deltaMode.get
GPUAdapterInfo.device.get	PaintRenderingContext2D.globalCompositeOperation.set	WheelEvent.wheelDelta.get
GPUAdapterInfo.vendor.get	PaintRenderingContext2D.isPointInPath	WheelEvent.wheelDeltaX.get
History.length.get	PaintRenderingContext2D.isPointInStroke	WheelEvent.wheelDeltaY.get
HTMLCanvasElement.captureStream	PaintRenderingContext2D.lineTo	Window.devicePixelRatio.get
HTMLCanvasElement.getContext	PaintRenderingContext2D.moveTo	Window.devicePixelRatio.set
HTMLCanvasElement.toBlob	PaintRenderingContext2D.rect	Window.innerHeight.get
HTMLCanvasElement.toDataURL	PaintRenderingContext2D.rotate	Window.innerHeight.set
HTMLMediaElement.canPlayType	PaintRenderingContext2D.roundRect	Window.innerWidth.get
HTMLVideoElement.webkitDecodedFrameCount.get	PaintRenderingContext2D.scale	Window.innerWidth.set
HTMLVideoElement.webkitDroppedFrameCount.get	PaintRenderingContext2D.shadowBlur.get	Window.matchMedia
InputDeviceCapabilities.firesTouchEvents.get	PaintRenderingContext2D.shadowBlur.set	Window.orientation.get
Keyboard.getLayoutMap	PaintRenderingContext2D.shadowColor.get	Window.outerHeight.get
MediaCapabilities.decodingInfo	PaintRenderingContext2D.shadowColor.set	Window.outerHeight.set
MediaCapabilities.encodingInfo	PaintRenderingContext2D.shadowOffsetX.get	Window.outerWidth.get
MediaDevices.enumerateDevices	PaintRenderingContext2D.shadowOffsetX.set	Window.outerWidth.set
MediaDevices.getUserMedia	PaintRenderingContext2D.shadowOffsetY.get	Window.pageXOffset.get
MediaRecorder.audioBitsPerSecond.get	PaintRenderingContext2D.shadowOffsetY.set	Window.pageXOffset.set
MediaRecorder.mimeType.get	PaintRenderingContext2D.stroke	Window.pageYOffset.get
MediaRecorder.videoBitsPerSecond.get	PaintRenderingContext2D.strokeRect	Window.pageYOffset.set
MouseEvent.screenX.get	PaintRenderingContext2D.strokeStyle.get	Window.screenLeft.get
MouseEvent.screenY.get	PaintRenderingContext2D.strokeStyle.set	Window.screenLeft.set
Navigator.appVersion.get	PaintRenderingContext2D.transform	Window.screenTop.get
Navigator.cookieEnabled.get	PaintRenderingContext2D.translate	Window.screenTop.set
Navigator.deviceMemory.get	PaintWorkletGlobalScope.devicePixelRatio.get	Window.screenX.get
Navigator.doNotTrack.get	Path2D.arc	Window.screenX.set
Navigator.getUserMedia	Path2D.closePath	Window.screenY.get

High Entropy APIs (continued)		
Navigator.hardwareConcurrency.get	Path2D.ellipse	Window.screenY.set
Navigator.javaEnabled	Path2D.lineTo	Window.scrollX.get
Navigator.language.get	Path2D.moveTo	Window.scrollX.set
Navigator.languages.get	Path2D.rect	Window.scrollY.get
Navigator.maxTouchPoints.get	Path2D.roundRect	Window.scrollY.set
Navigator.mimeTypes.get	PaymentRequest.canMakePayment	WorkerNavigator.appVersion.get
Navigator.pdfViewerEnabled.get	PaymentRequest.hasEnrolledInstrument	WorkerNavigator.deviceMemory.get
Navigator.platform.get	Plugin.description.get	WorkerNavigator.hardwareConcurrency.get
Navigator.plugins.get	Plugin.filename.get	WorkerNavigator.language.get
Navigator.productSub.get	Plugin.name.get	WorkerNavigator.languages.get
NavigatorUAData.brands.get	PushManager.supportedContentEncodings.get	WorkerNavigator.platform.get
NavigatorUAData.getHighEntropyValues	RTCIceCandidate.address.get	WorkerNavigator.userAgent.get
NavigatorUAData.mobile.get		

B List of bytecode tokens

This subsection presents a list of JavaScript bytecode tokens extracted from all scripts collected across a dataset of 100,000 websites. These tokens represent the low-level instructions generated by the V8 JavaScript engine (used in Chromium-based browsers) after parsing and compiling JavaScript code.

Each token corresponds to a specific operation (e.g., loading a variable, calling a function, reading a property) in the V8 interpreter's bytecode. By analyzing the frequency and patterns of these tokens, we can gain insights into script behavior and potentially identify patterns related to tracking, fingerprinting, or other complex client-side logic.

For reference, a complete list of available bytecode instructions (tokens) can also be found in the V8 source code, specifically in the file bytecode.h.

Table 6: Bytecode tokens

Bytecode tokens			
CreateFunctionContext	SetKeyedProperty.Wide	LdaLookupGlobalSlot	PushContext
GetKeyedProperty.Wide	CreateFunctionContext.Wide	Ldar	GetNamedProperty.Wide
JumpIfTrue.Wide	StaCurrentContextSlot	CallUndefinedReceiver.Wide	JumpIfUndefined.Wide
CreateMappedArguments	ShiftRightSmi.Wide	ForInPrepare.Wide	Star1
TestGreaterThanOrEqual.Wide	ForInNext.Wide	LdaCurrentContextSlot	Add.Wide
JumpIfNotNullConstant	JumpIfToBooleanFalse	CallProperty1.Wide	Div.Wide
Star3	BitwiseAnd.Wide	LdaLookupContextSlot	GetNamedProperty
TestLessThan.Wide	ModSmi.ExtraWide	JumpIfToBooleanTrue	CallProperty0.Wide
Star.Wide	LdaImmutableContextSlot	TestEqualStrict.Wide	Ldar.Wide
Star4	ShiftLeft.Wide	Mov.Wide	CallProperty2
AddSmi.Wide	JumpIfNotUndefined.Wide	Return	BitwiseOr.Wide
BitwiseXorSmi	CreateEmptyObjectLiteral	Dec.Wide	BitwiseXorSmi.Wide
Star5	CallProperty2.Wide	StaLookupSlot	Star6
Inc.Wide	BitwiseXorSmi.ExtraWide	CallUndefinedReceiver1	Sub.Wide
ConstructWithSpread	CallProperty1	TestLessThanOrEqual.Wide	BitwiseXor.Wide
LdaImmutableCurrentContextSlot	MulSmi.Wide	GetNamedPropertyFromSuper	LdaConstant
TestGreaterThan.Wide	StaGlobal.Wide	Star0	ShiftRight.Wide
CloneObject.Wide	LdaZero	LdaGlobal.Wide	CreateCatchContext.Wide
GetKeyedProperty	Construct.Wide	PushContext.Wide	SetNamedProperty
Mul.Wide	CallRuntime.Wide	CreateArrayLiteral	CallProperty.Wide
CallWithSpread.Wide	CreateClosure	Mod	JumpIfJSReceiver.Wide
LdaSmi	ShiftRight	Mod.Wide	TestEqualStrict
CallUndefinedReceiver0	JumpIfUndefinedOrNull.Wide	JumpIfFalse	DefineKeyedOwnProperty
LdaLookupSlotInsideTypeof	SetKeyedProperty	ToBoolean	GetTemplateObject.Wide
Add	CreateCatchContext	JumpIfNotUndefinedConstant	Jump
LdaTheHole	ShiftLeftSmi.Wide	DivSmi	SetPendingMessage
ShiftRightLogical.Wide	CallUndefinedReceiver	PopContext	CallJSRuntime
TestTypeOf	DeletePropertyStrict	ThrowReferenceErrorIfHole.Wide	JumpIfTrue
JumpIfToBooleanFalseConstant	CallUndefinedReceiver1.ExtraWide	LdaUndefined	SwitchOnSmiNoFeedback
Add.ExtraWide	LdaNull	ReThrow	CallUndefinedReceiver2.ExtraWide
Star7	TestReferenceEqual	SetKeyedProperty.ExtraWide	Star8
StaContextSlot	ShiftRightLogicalSmi.Wide	StaInArrayLiteral	JumpIfNotNull
BitwiseNot.Wide	Star2	JumpIfNull	JumpLoop.ExtraWide
Star	TypeOf	JumpIfTrueConstant.Wide	CallProperty0
ThrowReferenceErrorIfHole	JumpConstant.Wide	Mov	FindNonDefaultConstructorOrConstruct

Bytecode tokens (continued)

PopContext.Wide	CreateObjectLiteral	ThrowIfNotSuperConstructor	ToObject.Wide
DefineNamedOwnProperty	ConstructForwardAllArgs	ForInEnumerate.Wide	Star10
CreateBlockContext	ForInContinue.Wide	Star9	LdaImmutableContextSlot.Wide
ForInStep.Wide	Star11	LdaImmutableCurrentContextSlot.Wide	LdaModuleVariable.Wide
CallProperty	Div	LdaModuleVariable	Construct
Negate	StaModuleVariable	DeletePropertySloppy	StaContextSlot.Wide
JumpIfNull.Wide	TestGreaterThan	LdaContextSlot.Wide	LdaLookupGlobalSlot.Wide
JumpIfUndefinedOrNull	ToString	LdaLookupGlobalSlotInsideTypeof	ToObject
CreateRestParameter	Exp	ForInEnumerate	GetIterator
LdaLookupGlobalSlotInsideTypeof.Wide	ForInPrepare	JumpIfJSReceiver	Debugger
ForInContinue	CallRuntime	ExpSmi	CreateArrayFromIterable
CreateWithContext	JumpIfUndefined	JumpIfNullConstant	TestReferenceEqual.Wide
Star12	JumpIfUndefinedConstant	DeletePropertyStrict.Wide	Star13
LdaCurrentContextSlot.Wide	CallAnyReceiver.Wide	TestUndefined	StaCurrentContextSlot.Wide
ShiftRightSmi.ExtraWide	LogicalNot	ThrowSuperAlreadyCalledIfNotHole	ShiftLeftSmi.ExtraWide
Star15	ThrowSuperNotCalledIfHole	StaModuleVariable.Wide	Star14
BitwiseOrSmi.Wide	InvokeIntrinsic.Wide	ForInStep	TestNull
SuspendGenerator.Wide	JumpLoop	SwitchOnGeneratorState	ResumeGenerator.Wide
Inc	InvokeIntrinsic	SwitchOnSmiNoFeedback.Wide	LdaContextSlot
SuspendGenerator	Exp.Wide	CreateEmptyArrayLiteral	ResumeGenerator
LdaLookupSlot.Wide	LdaFalse	SetNamedProperty.Wide	JumpIfNotNull.Wide
LdaTrue	CallUndefinedReceiver2.Wide	LdaLookupContextSlot.Wide	CallUndefinedReceiver2
CreateRegExpLiteral	LdaLookupContextSlotInsideTypeof	TestUndetectable	CloneObject
GetNamedPropertyFromSuper.Wide	LdaGlobal	CreateObjectLiteral.Wide	StaLookupSlot.Wide
Sub	CallUndefinedReceiver0.Wide	DefineNamedOwnProperty.ExtraWide	ShiftLeft
DivSmi.Wide	SetNamedProperty.ExtraWide	TestLessThan	ToName
GetNamedProperty.ExtraWide	TestLessThanOrEqual	DefineKeyedOwnPropertyInLiteral	CreateObjectLiteral.ExtraWide
ShiftLeftSmi	GetTemplateObject	CallUndefinedReceiver0.ExtraWide	SubSmi
CreateClosure.Wide	LdaGlobal.ExtraWide	BitwiseAnd	CreateBlockContext.Wide
CallProperty1.ExtraWide	ShiftRightSmi	LdaConstant.Wide	Construct.ExtraWide
LdaGlobalInsideTypeof	CallUndefinedReceiver1.Wide	CallProperty2.ExtraWide	TestInstanceOf
Negate.Wide	CreateEvalContext.Wide	MulSmi	StaGlobal
JumpIfFalseConstant.Wide	ShiftRightLogicalSmi	CreateArrayLiteral.Wide	DeletePropertySloppy.Wide
SwitchOnSmiNoFeedback.ExtraWide	ModSmi	StaInArrayLiteral.Wide	Throw
DefineKeyedOwnProperty.Wide	GetKeyedProperty.ExtraWide	AddSmi	JumpIfUndefinedOrNullConstant
DefineKeyedOwnProperty.ExtraWide	BitwiseOr	BitwiseXor	StaInArrayLiteral.ExtraWide
BitwiseAndSmi.ExtraWide	GetIterator.Wide	CreateArrayLiteral.ExtraWide	BitwiseAndSmi.Wide
DefineNamedOwnProperty.Wide	CreateEmptyArrayLiteral.ExtraWide	ToNumeric	DefineKeyedOwnPropertyInLiteral.Wide
ConstructWithSpread.Wide	Dec	AddSmi.ExtraWide	BitwiseAndSmi
CreateEmptyArrayLiteral.Wide	CallUndefinedReceiver.ExtraWide	CallProperty.ExtraWide	LdaSmi.ExtraWide
TestEqual.Wide	CallJSRuntime.Wide	ShiftRightLogical	SubSmi.ExtraWide
CallProperty0.ExtraWide	LdaSmi.Wide	LdaLookupSlot	GetSuperConstructor
TestGreaterThanOrEqual	CallRuntimeForPair	CreateClosure.ExtraWide	SubSmi.Wide
CallAnyReceiver	LdaConstant.ExtraWide	CreateUnmappedArguments	ToNumeric.Wide
FindNonDefaultConstructorOrConstruct.Wide	TestIn	CreateRegExpLiteral.Wide	ThrowIfNotSuperConstructor.Wide
ToBooleanLogicalNot	TestInstanceOf.Wide	JumpIfToBooleanFalseConstant.Wide	Mul
ModSmi.Wide	JumpIfToBooleanTrueConstant.Wide	ToNumber	CallWithSpread
CreateWithContext.Wide	JumpIfNotUndefined	JumpIfToBooleanTrue.Wide	CallRuntimeForPair.Wide
BitwiseOrSmi	JumpIfFalse.Wide	TestEqual.ExtraWide	JumpLoop.Wide
Jump.Wide	BitwiseAnd.ExtraWide	MulSmi.ExtraWide	DivSmi.ExtraWide
TestEqualStrict.ExtraWide	TestEqual	ToNumber.Wide	CreateRegExpLiteral.ExtraWide
BitwiseNot	BitwiseOrSmi.ExtraWide	GetTemplateObject.ExtraWide	JumpIfTrueConstant
CreateEvalContext	CloneObject.ExtraWide	JumpIfToBooleanTrueConstant	JumpConstant
LdaGlobalInsideTypeof.Wide	JumpIfFalseConstant	JumpIfToBooleanFalse.Wide	