Artifact Evaluated
NDSS
Available
Functional
Reproduced

Chasing Cookies to the Source: In-browser Data Flow Backtracking for Cookie Compliance Triage

Yi Yang*, Mingxue Zhang*†¶, Yuxiang Ma*, Cong Zhang*†,
Penghui Li‡, Changhua Luo§, Weina Niu‖, Kui Ren*
*The State Key Laboratory of Blockchain and Data Security, Zhejiang University
†Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
‡Nanyang Technological University §Wuhan University
‖University of Electronic Science and Technology of China
{yiyang25, mxzhang97, yuxiangma, 0022090}@zju.edu.cn,
penghui.li@ntu.edu.sg, chluo2502@whu.edu.cn, vinusniu@uestc.edu.cn, kuiren@zju.edu.cn

***Abstract*—After the enforcement of privacy protection regulations like GDPR, cookie-related violations have proven to be prevalent on real-world websites, *e.g.*, advertising cookies being set before explicit user consent. Prior studies have either focused on detecting such non-compliant cookies, or on approximately attributing the violation to one or more scripts. Without localizing the exact responsible code, however, it remains difficult to determine the responsibility distribution, remediation choices, and the potential intentions and consequences.**

In this paper, we present COOKIETRACE, a browser-based framework for tracing the source (*e.g.*, variables carrying cookie attributes) of non-compliant cookies, thereby pinpointing and characterizing the culprit code for in-depth analysis. It conducts selective lazy data flow graph (DFG) construction at run time to minimize the computational overhead while maintaining high accuracy. Using COOKIETRACE, we analyze 6,995 websites with cookie violations collected from the Tranco top 500K list. Overall, the identified sources of non-compliant unnecessary and consent cookies span 522 and 45 distinct scripts, respectively, jointly determining cookie names, encoded values, and attributes such as expiration time.

Our analysis reveals that non-compliant cookie writes are typically rooted in a concentrated set of sources, while their effects propagate through complex data flows. This disparity makes violations difficult to audit and reason about using existing approaches, further hindering responsibility attribution in practice. We further showcase several possible incentives and downstream consequences of the violations, *e.g.*, monetization via advertisement distribution, and discuss the implications for potential mitigation.

I. INTRODUCTION

Privacy regulations such as the General Data Protection Regulation (GDPR) [1] and ePrivacy Directive (ePD) [2] impose requirements on how websites use cookies and process related personal data. The ePD specifically regulates the storage of and access to information on users' terminal equipment, including cookies, while the GDPR provides the broader framework for personal data processing and the validity of consent. Cookies therefore play a central role in online privacy compliance. Non-essential cookies are generally subject to consent requirements, while consent decisions are commonly collected through specialized user interfaces (*i.e.*, cookie banners) and stored in consent cookies.

Prior studies have investigated various compliance issues on the web [3], [4], [5]. In particular, extensive efforts have been invested to detect cookie-related violations (*e.g.*, consent cookies stored before user actions). For example, Bollinger *et al.* [6] analyzed 29,398 websites, revealing that 94.7% of them contained at least one GDPR compliance violation in their cookie settings. However, these studies focused on detecting *whether* the violations exist—by identifying any non-compliant cookies—without examining *who* cause the violations and *how* the underlying mechanisms work. In practice, non-compliant cookies are commonly written by JavaScript code. Understanding the culprit scripts and their behaviors is crucial for practical compliance enforcement. Existing violation attribution tools [7], [8], [9] adopt a simplified approach by inspecting the call stack when the violation occurs, and treating the bottom-/top-frame or all in-stack scripts as the culprits, acknowledged to be imprecise [7]. Nonetheless, without a precise attribution, it remains challenging to determine the legal responsibility and remediation strategy. Blindly blocking the entire scripts risks functional and compliance breakage, as compliant and non-compliant behaviors are often intertwined. Intervening only at cookie writes could be effective but leads to redundant remediation across websites and obscures the root causes of violations. As a result, the mitigation becomes fragmented, and the responsibilities are often misaligned across the ecosystem.

To enable precise triage of cookie violations, we aim to track cookies back to their sources. We define cookie sources

¶Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.230609
www.ndss-symposium.org

as the origin of constituent data (*e.g.*, strings and numbers) in cookies, *i.e.*, the literals ultimately used to construct cookies. Such sources expose the concrete logic used to construct a cookie, providing more precise guidance for subsequent triage and remediation. Nonetheless, tracking cookie sources is a challenging problem. First, modern websites commonly include a huge amount of scripts, which intensively cooperate or interfere with each other to fulfill diverse functionalities. The intricate interactions and dependencies make it difficult to precisely model the JavaScript data flows, *e.g.*, purely based on call stacks. Our evaluation results also proved that manual auditing is challenging, due to the heavy obfuscation and dynamic interactions among scripts. Second, non-compliant cookies may originate from any object in any script. A comprehensive data flow tracking would be extremely time-consuming, if not impossible.

In this paper, we present the first study that systematically traces non-compliant cookies back to their sources. We aim to understand which specific scripts are responsible for the violations, and how they construct the cookies. To this end, we design and implement CookieTrace, a browser-based data flow analysis framework that tracks the provenance of cookies. It addresses the first challenge by inspecting the data flows, including those through functions absent from the call stack, at run time, intrinsically capturing interactions among scripts in the actual execution environment. CookieTrace further incorporates selective and lazy data flow graph (DFG) construction, which significantly limits the analysis scope and reduces the computational overhead. Unlike prior slicing [10], [11] or demand-driven and selective analysis [12], [13] that rely on preconstructed program representations, execution-wide dependence information, or predefined taint sources, CookieTrace records compact runtime metadata and materializes invocation-specific DFGs only for function invocations relevant to the observed cookie write, thus effectively reduces the analysis scope.

Using CookieTrace, we analyzed the cookie violations on 6,995 websites collected from the Tranco top 500K websites [14]. Note that our goal is not to detect non-compliant cookies. Instead, we focus on the precise attribution of known cookie violations by locating the cookie sources. The results demonstrate that data flows related to non-compliant cookie writes are quite complex, where the constituent strings of cookies span across on average 14.47 functions. In particular, 50% of websites have more than 35% of their cookie sources invisible in the call stack, further complicating manual or even automatic auditing. Our manual verification proved the efficiency and effectiveness of CookieTrace: it achieves a precision of 98.7% and recall of 100% in locating the cookie sources. We also discuss the main suspected incentive to sustain advertisement distribution and reduce compliance burdens. The results demonstrate that CookieTrace enables accurate and efficient measurement of cookie violations at scale. Beyond cookies, we believe CookieTrace also provides a general methodology for tracing the provenance of other sensitive data in web applications.

In summary, we outline the contributions of this work in the following list.

- We design and develop CookieTrace, an in-browser data flow analysis framework. To the best of our knowledge, it is the first tool for effective provenance tracking of non-compliant cookies, enabling precise triage of cookie violations while limiting data-flow analysis to relevant runtime function invocations.
- We measure and characterize cookie violations on the Tranco top 500K websites. We demonstrate the effectiveness and efficiency of CookieTrace, and show that the propagation of cookie values is highly complex and often obscured from call-stack–based analysis.
- For the two types of cookie violations analyzed, we observe a distinct responsibility concentration and scripts collaboration patterns. We further discuss the potential incentives, consequences, and the implications for mitigation accordingly.
- We plan to open source CookieTrace and the associated dataset (with appropriate sanitization) to enable reproducibility and future research.

II. Background

A. Privacy Regulations

Ever since 2018, various privacy regulations have been enforced to regulate how websites collect and process user data. One typical example is the General Data Protection Regulation (GDPR) [1], which regulates personal data processing and establishes requirements for valid user consent. The ePrivacy Directive (ePD) [2] complements the GDPR with more specific rules for electronic communication privacy. Its Article 5(3) requires user consent before storing information on user equipment, including setting non-essential cookies (*e.g.*, tracking or advertising cookies). Violations can result in substantial fines, *e.g.*of up to €20 million or 4% of annual global revenue under GDPR [15]. Similar legislation, such as the California Consumer Privacy Act (CCPA) [16], reflects a global trend toward stricter consent and data usage requirements.

B. Cookies

Cookies are small pieces of data stored on a user's device and managed by the browser. They are widely used to maintain session state, remember user preferences, and enable tracking or targeted advertising across websites. Technically, cookies can be set either through the `Set-Cookie` HTTP response header or JavaScript APIs such as `document.cookie` and the Cookie Store API [17]. Since Cookie Store usage remains rare [18], prior compliance studies have primarily focused on `document.cookie` [19], [20], [21], [22]. Although modern browsers have provided a range of cookie protection mechanisms such as the HttpOnly [23], SameSite [24], and Secure [25] attributes, a significant portion of cookies are still set via JavaScript [26] without such protections [27], leaving privacy- and compliance-relevant cookies prevalent in the JavaScript execution environment.

C. Cookie Violations

Technical standards have also been proposed to facilitate compliant implementation with privacy regulations. For example, IAB Europe [28] introduced the Transparency and Consent Framework (TCF) [29], which requires that user consents be encoded in consent cookies, as a base64url string. Specifically, the consent string specifies which "vendors" are allowed to operate, the declared "purposes" of data processing, whether the data collection is based on "legitimate interests", and any "special features" that may be enabled. The collection of user choices is accomplished by Consent Management Platform (CMP) scripts. The latest technical standard for CCPA is the Global Privacy Protocol (GPP) [30]. Unlike GDPR, CCPA employs an opt-out mechanism: cookies may be created before user actions, while the consents must be correctly encoded in the consent cookies. Additionally, Article 5(3) of the ePD requires unnecessary cookies, *i.e.*, those not strictly necessary to provide the user-requested service, such as advertising cookies, to be created only with user consents.

Prior studies have identified various violations against the above requirements. Specifically, two types of violations have been detected: storage of non-compliant consent cookies, and non-compliant unnecessary cookies. For instance, positive consents to data collection are frequently encoded in consent cookies against or before user choices [31], [7]. Unnecessary cookies can also be stored without user consents [6]. Accordingly, in this work, we aim to track the sources of the two categories of non-compliant cookies.

III. Problem Statement

A. Motivation

Prior studies have demonstrated that cookie violations are widespread, particularly for consent cookies and unnecessary cookies (*e.g.*, marketing cookies). To the best of our knowledge, however, none of the existing approaches have precisely identified the culprits, *e.g.*, which code snippets are responsible for constructing the non-compliant cookies. In practice, one cookie may be assembled across multiple assignments, reused by shared logic, or injected through third-party code. This diversity and complexity make it difficult to localize the specific code responsible for the violation. Consequently, any remediation tends to be ad-hoc and fragmented, *e.g.*, blocking the entire scripts, patching sinks blindly, or repeatedly fixing symptoms across websites, none of which reliably addresses the root causes. The regulators would also struggle to determine whether a violation results from an erroneous implementation of the consent management logic, or the indirect effect of external third-party components, undermining responsibility attribution and the feasibility of targeted enforcement actions.

Note that cookies in the `Set-Cookie` header can be unambiguously attributed to the responding server and have clear origins. In this work, we focus on cookie writes in JavaScript, which often involve multiple scripts, complex data flows, and dynamic behaviors. Listing 1 shows a simplified code snippet we observed on a real-world website, where three distinct scripts cooperate to define a non-compliant consent cookie, *i.e.*, by setting vendors A and C as consented without user awareness. Specifically, `configdata.js` obtains the configuration data such as domain name, and records positive consents to data usage from vendor A (Line 2). The second script `session.js` retrieves the configuration from `configdata.js` and adds positive consent to vendor C (Line 8-9). Finally, script `main.js` normalizes and encodes the above non-compliant configuration, and performs the cookie write (Line 22). In particular, the consent cookie name was also constructed by `generateCookieName()` from `configdata.js`.

Existing approaches (*e.g.*, CSChecker [7], styxJS [9]) perform inaccurate violation attribution by capturing the call stack, and identify the culprit scripts based on the stack frames. These methods face three major limitations. First, the call stack offers insufficient visibility into the data flow involved in cookie writes. In Listing 1, the call chain `finalizeCookieValue -> fetchRawPreferences -> getUserConfig` is invisible in the call stack at the point where `document.cookie` is defined (Line 22), missing the origin of cookies in Line 2 and 9. Second, attributing a violation only to the cookie-writing (*i.e.*, top-frame) scripts may obscure the actual root cause of violations. In Listing 1, the cookie write (Line 22) in `main.js` is semantically neutral. The root cause of the violation lies in the value passed to it, which contains non-compliant inputs from Lines 2 and 9. Without analyzing data propagation among values inside scripts, existing approaches may misalign the responsibility. Third, modern scripts are commonly large and minified [32]. Even with the responsible scripts identified, the actual origin of non-compliant cookies remains hidden in the huge number of program statements, making it challenging to choose the remediation strategy.

In this work, we aim to attribute the violation by backtracking data flows from cookie write operations to their corresponding cookie sources, thereby accurately locating their root causes. For example, by tracing back from Line 22 in Listing 1, we can identify the cookie sources from `configdata.js` and `session.js`, offering concrete guidance for subsequent compliance assessment and remediation decisions.

B. Research Challenges

Tracking cookie sources in the JavaScript code is far from trivial. An intuitive approach is to construct a complete inter-procedural data flow graph for all JavaScript functions that write cookies, and then trace each cookie back to the sources of its constituent data. The complicated inter-script dependencies can be captured via a dynamic in-browser data flow analysis. However, real-world websites often include hundreds or even thousands of complicated scripts. Constructing and maintaining DFGs for all of them would impose prohibitive runtime overhead. Our experiments on 100 randomly sampled websites proved that exhaustive DFG construction prolonged the page loading beyond 10 minutes, demonstrating the impracticality (§V-C). While taint analysis tools provide an alternative by marking variables as sources and propagating labels to cookie

```
1  // configdata.js
2  const user_config={domain:'.example.com',
   ↪  cookiePrefix:'mk_',vendors:'A=1;B=0'};
3  const getCookiePrefix=()=>user_config.cookiePrefix;
4  const generateCookieName=()=>getCookiePrefix()+'_id';
5
6  // session.js
7  function fetchRawPreferences(userId) {
8    config = getUserConfig(userId);
9    return config.vendors + ';C=1';
10 }
11
12 // main.js
13 function finalizeCookieValue(userId) {
14   const raw = fetchRawPreferences(userId);
15   const norm = normalizePrefs(raw);
16   const enc = encodePrefs(norm);
17   writeConsentCookie(enc);
18 }
19
20 function writeConsentCookie(value) {
21   ...
22   document.cookie = generateCookieName() + value + ";
   ↪  domain=example.com; path=/";
23 }
24
25 finalizeCookieValue("user-1");
```

Listing 1: A simplified real-world cookie violation on adage.com.

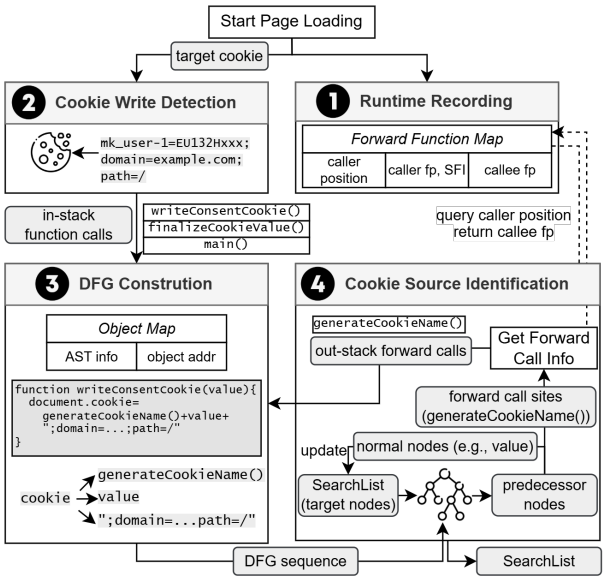

Fig. 1: The workflow of CookieTrace.

writes (sinks), these approaches also suffer from substantial runtime overhead, as *all* variables could be the sources.

To make the problem tractable, one possible way to limit the analysis scope is to focus on data flows related to the target cookie writes. Upon observing a cookie write, one can focus on functions in the current call stack and selectively construct DFGs at run time only for those functions. For example, in Listing 1, the call stack when executing to Line 22 includes `main()` (*i.e.*, Line 25 in the global scope), `finalizeCookieValue()`, and `writeConsentCookie()`. The rest irrelevant functions on the same page can be excluded from the analysis, significantly reducing the overhead.

However, such a pruning strategy may overlook some cookie sources, since it is common for functions to define constituent variables and return them for later use, thereby disappearing from the active call stack at the moment of cookie writes. One typical example is the `generateCookieName()` call at Line 22, which is absent from the stack at the cookie write. In the experiments, we observed on each website an average of 18.02 such function calls along the non-compliant cookie-related data flows (§V-B3). To strike a balance between the analysis accuracy and computational efficiency, it is necessary to capture data flows across *all relevant* code, which would require on-demand DFG construction for involved JavaScript code only. To conclude, the technical challenges are as follows.

- **Invisible data dependencies from call stacks (C1).** Existing tools (*e.g.*, StyxJS, CSChecker) rely on call stacks to characterize data flow during cookie writes. Although call stacks intuitively reflect runtime code execution paths, they are inherently unable to capture complete data flows and break the continuity of data flow tracking, making it impossible to rely solely on stack traces to understand how cookie values are constructed.
- **Unlimited source space (C2).** Non-compliant cookies can be constructed from any variables, including objects and primitive type values. This greatly enlarges the analysis scope, hindering the application of traditional dynamic data flow analysis, *e.g.*, taint tracking, which usually requires a comprehensive definition of the potential *sources*.
- **Unpredictable execution scope (C3).** In order to make the tracking practical and efficient, we need to limit the analysis scope. However, during the execution, it will be very difficult to predict which scripts or functions may participate in the definition of non-compliant cookies. Designing a pruning strategy that retains the accuracy is also challenging.

IV. Design

In this section, we present CookieTrace, a novel dynamic browser-based framework equipped with selective and lazy DFG construction, for tracking sources of cookies on the web. The workflow is depicted in Fig. 1.

CookieTrace first records necessary information for DFG construction and cookie source identification during the *runtime recording* stage (S1). Upon detecting a target cookie write (S2), it inspects the current call stack and starts constructing DFGs for functions in the stack (S3). It then starts the backtracking in the reverse order of function calls to identify cookie sources, in the *cookie source identification* stage (S4). Once encountering function calls that return cookie-related values (referred to as *forward function calls*) during S4, CookieTrace retrieves the information recorded in S1 to build the corresponding DFGs, and iteratively continues the backtracking. The analysis of forward function calls addresses C1 by capturing relevant function invocations that do not appear in the call stack after returning. The on-demand DFG construction further addresses C2 and C3, reducing the computational overhead.

In the following, we first introduce how CookieTrace detects cookie writes (§IV-A) and constructs the DFGs (§IV-B). We then describe the information recorded at run time that

enables lazy DFG construction (§IV-C). Finally, we present the cookie source identification method (§IV-D) and the implementation details (§IV-E).

A. Cookie Write Detection

As mentioned in §II-B, `document.cookie` remains the dominant way for setting non-compliant cookies. Therefore, similar to the existing studies [19], [20], [21], [22], we focus on the detection of cookie writes via `document.cookie`. However, `document.cookie` can still be accessed through aliases (*e.g.*, `var d = document; d.cookie = data;`) or bracket notation (`document["cookie"]`).

In V8, the JavaScript variables are represented as `Object` instances. Therefore, to comprehensively detect non-compliant cookie writes, COOKIETRACE hooks the property setter method of `Object`. It then checks whether the constructor of the receiver object is `HTMLDocument` and the property name equals `cookie`. This also allows COOKIETRACE to extract the cookie name that is being written. Note that it is not our main focus to detect cookie violations on websites. Instead, we aim to analyze the root cause and culprits of the violations. Therefore, in this work, we rely on existing compliance analysis tools (§V-A) to detect non-compliant cookies and compile a violation cookie list. During the analysis, COOKIETRACE compares the extracted cookie name against the list to locate the target cookie write for further analysis.

After detecting the target cookie write, COOKIETRACE instantly captures the current JavaScript call stack, and activates the DFG construction stage, as described below.

B. DFG Construction

COOKIETRACE builds DFGs based on abstract syntax trees (ASTs), and propagates the information to runtime objects. In this section, we first present an overview of the data flow identification strategy. We then describe how we process several special syntax features to support precise backtracking.

1) Data Flow Identification: In a data flow graph, an edge represents the data dependency between objects. For instance, an assignment statement `var a = b` produces an edge between `a` and `b`. Therefore, one straightforward way to identify the data flow is to inspect the AST structure, *i.e.*, adding edges at each assignment operation. Nonetheless, as an object can be accessed using multiple aliases (thus different AST nodes), keeping track of the data dependencies purely based on the AST is difficult. In contrast, the memory address offers a unique identifier for each object, regardless of the variable name. Therefore, COOKIETRACE identifies data dependencies based on ASTs, and creates DFG edges between the corresponding objects using their memory addresses. To this end, COOKIETRACE maintains an *object map* from AST node to the corresponding object address.

Object map. COOKIETRACE reuses the object map in Arcanum [33], the state-of-the-art in-browser data flow analysis tool for JavaScript. Specifically, it inserts runtime function calls during JavaScript compilation, *e.g.*, at each variable definition, to record the addresses assigned to objects during script execution, as well as the corresponding AST node information (*i.e.*, node type and position). When updating the object map, COOKIETRACE uses AST node information and object addresses as the *key* and *value*, respectively.

Edge direction. Unlike conventional DFGs where the edges are directed from sources to their derived values (*e.g.*, `b` $\rightarrow$ `a` for `a = b + c`), COOKIETRACE reverses the edges and connects the derived objects to their sources (*i.e.*, `a` $\rightarrow$ `b`, `a` $\rightarrow$ `c`). This enables natural backward traversal starting from the cookie object.

2) Augmented Syntax Handling: COOKIETRACE specifically handles several JavaScript syntax features to enable precise address-based backtracking, as summarized in Table I.

Template literal. When a template literal appears on the right-hand side (RHS) of an assignment, COOKIETRACE identifies the embedded expressions and creates DFG edges from the LHS variable to them.

Destructuring assignment. Destructuring assignments are handled by matching each destructured target with the corresponding RHS expression and creating the DFG edges.

Compound assignment. A compound assignment such as `a += b` makes the new value of `a` depend on both `b` and its previous value. However, the AST contains only one node for `a`. Updating its entry in the object map therefore overwrites the previous object address, preventing further backtracking to the original `a`. To preserve this dependency, COOKIETRACE flattens `a += b` as `a = a + b` in the V8 parser, so `a` at the RHS and LHS are represented by distinct AST nodes.

Call argument. Expression arguments also introduce a address-mapping issue. For example, in `myfun(a + b)`, the callee parameter receives the evaluated result of `a + b`, whose address differs from those of `a` and `b`. Address matching alone therefore cannot recover the dependency from the parameter to the argument expression.

A naive solution is to explicitly create inter-procedural DFG edges from callee parameters to objects in argument expressions, but this requires correlating objects across the caller and callee contexts. Instead, we reinterpret argument passing as a form of specialized assignment, by modifying the AST. COOKIETRACE rewrites the call as `myfun(`p_0 `= a + b)`, where the shadow variable p_0 records the evaluated argument address and introduces DFG edges $p_0 \rightarrow$ `a` and $p_0 \rightarrow$ `b`. The address of p_0 can then be matched with the callee parameter, bridging the inter-procedural data flow. To avoid variable name conflicts, we assigned a special name to the shadow variable. Details can be found in §IV-E.

Loop assignment. For assignments inside loops, the same LHS AST node may be evaluated repeatedly, causing its object-map entry to be overwritten across iterations and losing dependencies from earlier iterations. To address this, COOKIETRACE uses V8's built-in `in_loop` flag to identify such in-loop assignments and retains all observed object addresses instead of replacing previous entries. For example, for `a = obj[k]`, COOKIETRACE records the addresses of `a` and

TABLE I: Augmented JavaScript Syntax Handling.

Category	Example Expression	AST Modification	DFG Edges
Template Literal Assignment	`var a = `${b} + ${c}``	—	a → b, a → c
Destructuring Assignment	`var [a1,b1,c1] = [x1,y1,z1]` `var {a2,b2,c2} = {a2:x2,b2:y2,c2:z2}`	—	a1 → x1, b1 → y1, c1 → z1 a2 → x2, b2 → y2, c2 → z2
Compound Assignment	`a += b`	`a = a + b`	$a_{LHS} \rightarrow a_{RHS}$, $a_{LHS} \rightarrow$ b
Call Argument Assignment	`myfun(a + b)`	`myfun(`p_0 `= a + b)`	$p_0 \rightarrow$ a, $p_0 \rightarrow$ b
Loop Assignment	`for(k=0;k<obj.length();k++): a = obj[k]`	—	$a_k \rightarrow$ obj[k], $(0 \leq k <$ obj.size())

`obj[k]` at each iteration without deduplication, allowing the corresponding DFG edges to be reconstructed across iterations.

C. Runtime Recording

To minimize the runtime overhead, COOKIETRACE aims to build DFGs only for functions involved in cookie writes. Such functions can be typically classified into two types: 1) those in the call stack at the cookie write (type-1), and those returning values later used to construct the cookie (type-2). The type-1 functions can be easily captured in the call stack at cookie writes. The type-2 functions (referred to as *forward functions*), however, are more difficult to cover, as they can only be detected after finishing the execution, at which time the information for DFG construction is no longer available.

To address the challenge, COOKIETRACE implements another map (*forward function map*) to record the information needed for DFG construction during script execution. It can then retrieve the information and selectively construct DFGs for forward functions when necessary. We describe detailed design of the map next.

1) Map Structure: To differentiate multiple invocations of the same function, except for the compilation information (`compilation_info`), COOKIETRACE additionally uses the callee frame pointer (`callee_fp`) to construct DFGs. Therefore, both types of information need to be stored in the forward function map as the *value*, for future reference. Note that the stored information would only be retrieved for forward functions, after they finish execution. We thus use the position of function call statement (`call_site_position`) as the *key*, because it effectively identifies function calls and remains available after the function returns.

Maintaining the forward function map, however, is still non-trivial. Although `call_site_position` can be obtained at the call site, the forward functions do not leave any stack frame before or after their execution, making `callee_fp` available only on the callee side. To correlate the information, for each function call, we need to construct a unique identifier that is accessible on both the caller and callee sides.

One straightforward approach is to use the function name. However, it introduces many conflicts as scripts commonly declare different functions with the same name. In V8, a special object `SharedFunctionInfo (SFI)` is created after the compilation of a function to store compile-time metadata, which provides higher uniqueness. To further distinguish different dynamic calling contexts, we incorporate frame pointer of the caller function (`caller_fp`) to construct a composite unique identifier for each function call. In particular, the caller and callee can both access `caller_fp` via the call stack. This motivates our 2-level mapping structure for the forward function map. Specifically, COOKIETRACE first maps `call_site_position` to the composite identifier `(SFI, caller_fp)`, which can then be used to retrieve `compilation_info` and `callee_fp`, thereby supplying all necessary information for lazy DFG construction at forward function calls.

2) Updating the Map: COOKIETRACE inserts two runtime function calls during JavaScript compilation to update the forward function map: one at the call statement (caller side) and the other at the return statement (callee side).

Specifically, when a function call is executed, the first runtime function will be invoked to update the map from `call_site_position` to `(SFI, caller_fp)`. The map from `(SFI, caller_fp)` to `(compilation_info, callee_fp)`, in contrast, can be updated at arbitrary statement inside the callee function. COOKIETRACE updates the map at return statements, since every forward function contains at least one such statement. This ensures completeness while avoiding the overhead of recording irrelevant functions.

D. Cookie Source Identification

In this section, we introduce how COOKIETRACE tracks the cookie source, and identifies participating scripts along the data flows. Specifically, it first performs *backward tracking* on in-stack functions (§IV-D1). During this procedure, when COOKIETRACE encounters a forward function call, it performs *forward tracking* (§IV-D2) accordingly.

1) Backward Tracking: As stated in §IV-B, COOKIETRACE records data flows using the object addresses. It starts the backward tracking with memory address of the cookie object, and iteratively searches for the related object addresses. For example, given an assignment statement `document.cookie = b + c`, COOKIETRACE first identifies the addresses of `b` and `c` along the data flow, and then recursively traces their predecessors in the DFG. This process continues until no further predecessors can be found for a given address. Once the tracking within the current function terminates, COOKIETRACE proceeds along the call stack in the reverse order of function invocation, and repeats the tracking for each function in the stack frames. Data flows across functions, *e.g.*, those between the arguments and function parameters, can be tracked using the object addresses.

In addition, for each object address, COOKIETRACE maintains an auxiliary map that records the corresponding AST

node information. During the backward tracking process, if an object is identified as a function return value, *i.e.*, the associated AST node represents a function call statement, COOKIETRACE leverages the AST node position (`call_site_position` in §IV-C1) to retrieve the information from forward function map, and activates the DFG construction stage again for the forward functions. It then continues the analysis by performing forward tracking, as described below.

2) Forward Tracking: The forward tracking proceeds in a similar manner to backward tracking, except that it starts from the object address of the return value. When a forward function invokes another forward function, the tracking is recursively triggered until all relevant functions are analyzed and the sources of the involved objects are identified. The resulting source list is then propagated back to the backward tracking module, which integrates it into the analysis to produce the complete set of cookie sources.

E. Implementation

We implemented COOKIETRACE atop Arcanum [33], only to avoid re-implementing low-level instrumentation (*e.g.*, the object map). All core components, including lazy-selective DFG construction, the forward function map, and forward tracking, were independently implemented by us in ∼4K lines of code. COOKIETRACE also remains applicable to modern cookie features such as CHIPS and partitioned cookies since they refine cookie management without altering the underlying cookie write operations that COOKIETRACE analyzes.

We implement the object and forward function maps using `V8GCHashTable` in V8 and represent DFGs with adjacency lists. Target cookie names identified by existing compliance analysis tools (see §V-A) are passed to COOKIETRACE via command-line parameter `--js-flags`. For call-argument rewriting, we assign the shadow variable a special name `call_arguments_assign_LHS`, which is unlikely to collide with user-defined variables. Since the original parameter names remain unchanged and DFG construction relies on runtime object addresses rather than variable names, the rewriting affects neither program functionality nor DFG construction. Finally, we use the nlohmann library [34] to output the identified cookie sources in JSON format.

Additionally, COOKIETRACE requires complete call stacks for backtracking, whereas V8 exposes only synchronous stacks that terminate at asynchronous boundaries. We therefore implement lightweight stack stitching by instrumenting asynchronous primitives such as `setTimeout` to associate each callback with its parent synchronous stack in a hash table. During backtracking, COOKIETRACE traverses these associations to recover the preceding execution contexts and reconstruct the complete call chain.

V. EVALUATION

In this section, we present the comprehensive evaluation results of COOKIETRACE. In particular, we aim to answer the following research questions.

- **Effectiveness (RQ1)**: How precise and effective is COOKIETRACE in tracking non-compliant cookie sources, and what contributes to its advantage over existing works?
- **Efficiency (RQ2)**: What is the runtime overhead incurred by COOKIETRACE?
- **Structural Opacity (RQ3)**: What are the key challenges in tracing cookie sources? Specifically, how far are cookie sources from cookie-writing operations, and how dispersed are they across functions?
- **Responsibility Concentration (RQ4)**: Are violations concentratedly dominated by a small set of "heavy-hitter" scripts or spread across a long tail of distinct actors? What are the predominant functional categories (*e.g.*, advertising, analytics, CMPs) of the culprit scripts?
- **Collaboration Measurement (RQ5)**: To what extent do scripts collaborate in producing non-compliant cookies, and what are the common collaboration patterns?

A. Experimental Setup

1) Dataset: We conducted a large-scale scanning of the Tranco top 500K websites using CSChecker [7] to detect non-compliant cookies. We chose CSChecker, as it is compatible with the latest TCF implementation standards. We used the tool to record all cookies stored before users grant explicit consent, decoded the consent strings to identify positive consents to specific vendors, purposes, special features, or legitimate interests. We matched the recorded cookies against Open-Cookie-Database [35], which describes the categories of major cookies, to identify any non-compliant unnecessary cookies. We considered "personalization" and "marketing" cookies as unnecessary ones and excluded "functional", "analytics", and "security" cookies, as the former categories mandate consents according to the IAB TCF guidelines [36]. We considered only the cookies stored before user interactions, to avoid inaccuracy incurred by subjective interaction strategies. The conservative detection ensures that only genuine non-compliant cookies are included in the analysis, eliminating potential false positives. Nevertheless, COOKIETRACE can be applied to analyze any non-compliant cookie writes, as discussed in §VI.

Overall, we used CSChecker to detect 6,995 websites with 9,783 non-compliant cookies. More specifically, we derived a list of 1,287 websites with 1,287 consent cookies, and 5,708 websites with 8,496 unnecessary cookies. In this work, we used COOKIETRACE to track the sources of these cookies.

2) Running COOKIETRACE*:* We conducted all experiments on Debian GNU/Linux 12 using multi-threaded parallel processing. To emulate a realistic GDPR-compliant European network environment, we visited all 6,995 websites from a German IP address. We automated the visits via the Selenium WebDriver. For each website, we waited at most 10 minutes for a full page load, and recorded the loading time. We additionally recorded three timestamps at the end of runtime recording, DFG construction, and cookie source identification, to measure the execution time of the primary stages.

3) Baseline Selection: We do not include *static* JavaScript analyzers as baselines because they mainly target self-contained

libraries or Node.js projects [37], [38], [39] and do not fully model dynamic web behaviors, *e.g.*, dynamically loaded code, asynchronous events, and cross-script interactions [40], [41]. Minification and obfuscation further limit their applicability [42]. A direct comparison is not like-for-like.

4) Disclosure: Following the coordinated vulnerability disclosure principles recommended by CERT/CC [43], we notified affected websites and relevant script providers before wider dissemination. Additional details are provided in the Ethics Considerations section.

B. Effectiveness of CookieTrace (RQ1)

To evaluate the effectiveness of CookieTrace, we measured the precision and recall of CookieTrace in cookie source identification. In particular, we randomly sampled 160 non-compliant cookies on 152 websites, and manually audited the code to establish a ground truth. We further compared the performance with state-of-the-art baselines, and explained the performance gain through a large-scale analysis.

1) Ground truth collection: The ground truth was collected through manual analysis of 40 and 120 non-compliant consent cookies and unnecessary cookies, respectively. We performed a stratified sampling in proportions close to their prevalence in the full dataset, to construct a representative sample. We exhaustively collected all cookie sources as the ground truth and compared them with the output of CookieTrace. Two authors, both with over a year of experience in JavaScript analysis, independently established the ground truth using only Chrome DevTools [44] to identify sources of each component of a target cookie. §C details ground truth collection and manual verification. We restricted the evaluation to 160 non-compliant cookies, because manually extracting all sources for target cookies is highly time-consuming for real-world JavaScript, which is often extensively minified [42] and exhibits complex call structures, requiring over 30 minutes per cookie on average. In total, the verification took around 20 person-days.

2) Results: We measured the precision and recall of at both the script and source levels. We refer to scripts that defined at least one cookie source as the *source scripts*. The true positives (TP) denote correctly identified cookie sources (at the source level) or source scripts (at the script level). Correspondingly, false positives (FP) denote incorrectly identified ones, and false negatives (FN) refer to the missed ones. We computed precision and recall relative to its ground truth, and reported the overall values across all sampled cookies.

Across the 160 non-compliant cookies, CookieTrace achieved an average precision of 98.7% and a perfect recall of 100% at the source level (Table II). The FPs primarily stem from a limitation of our address-based tracking. V8 employs string interning, storing identical string literals at a single shared memory location. Consequently, distinct variables that reference the same interned literal become indistinguishable by the address, leading to spurious matches. Nonetheless, the effect is narrowly scoped. It occurs only when multiple variables are *initialized* with the same literal constant, which is later used to construct cookies. Dynamically constructed strings (*e.g.*, string produced through concatenation, slicing, or encoding operations), which dominate real-world cookie manipulations, remain unaffected. Foxhound [45] avoids this issue by selectively disabling interning for tainted strings. Without a priori source knowledge, CookieTrace would need to disable interning broadly. Given the limited observed impact, we retain V8's default string interning. At the script level, CookieTrace achieved 100% precision and recall.

Deduplicated Result. Due to the centralized nature and common reuse of JavaScript code on the web, multiple non-compliant cookies can share the same writing logic. To further ensure that the reported precision and recall are not inflated by repeated samples with the same execution logic, we additionally evaluate CookieTrace on a deduplicated subset. Specifically, we treat two cookie writes as duplicates when they share the same cookie name and identical control flow, including the same call sequence and intra-procedural conditional branches, and retain only one representative sample (§C). Under this criterion, 85 samples remain after deduplication. On this deduplicated subset, CookieTrace still achieves high source-level precision and recall of 98.3% and 100%, respectively. Detailed results are presented in Table III.

TABLE II: Comparison of CookieTrace and prior stack-based tools on the original (160) sampled cookies.

Approach	Source-level		Script-level	
	Recall (%)	Precision (%)	Recall (%)	Precision (%)
CSChecker	38.7	N/A	31.0	89.4
styxJS	73.0	N/A	77.9	88.5
Sanchez-Rola *et al.*	32.8	N/A	32.7	94.4
CookieTrace	100	98.7	100	100

TABLE III: Comparison of CookieTrace and prior stack-based tools on the deduplicated (85) sampled cookies.

Approach	Source-level		Script-level	
	Recall (%)	Precision (%)	Recall (%)	Precision (%)
CSChecker	42.2	N/A	39.1	88.2
styxJS	77.8	N/A	81.8	90.2
Sanchez-Rola *et al.*	33.0	N/A	40.1	90.6
CookieTrace	100	98.3	100	100

3) Comparison with existing works: We further compared the effectiveness of CookieTrace against three existing tools for cookie attribution: CSChecker, styxJS, and Sanchez-Rola *et al.* [8], which attribute cookie violations to scripts based on the bottom, top, and all in-stack frames, respectively.

As CookieTrace in contrast detects the specific cookie sources instead of the culprit scripts only, we align the granularity discrepancy to enable the comparison. Specifically, for the source-level comparison, when a baseline identifies a culprit script, we conservatively treat all cookie sources in that script as detected. This alignment favors the baselines by extending script-level detections to multiple fine-grained sources. Nevertheless, our empirical evaluation on both the 160 original sampled non-compliant cookies and the 85 deduplicated ones shows that CookieTrace still achieves the

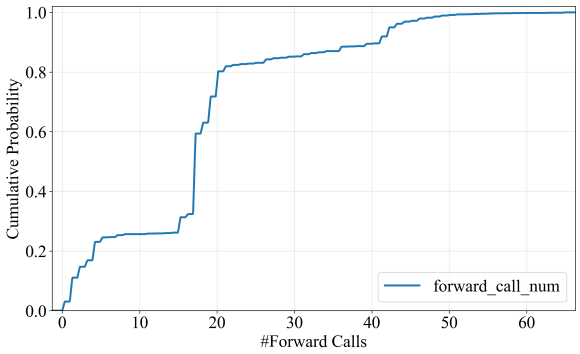

Fig. 2: Cumulative distribution of forward calls.

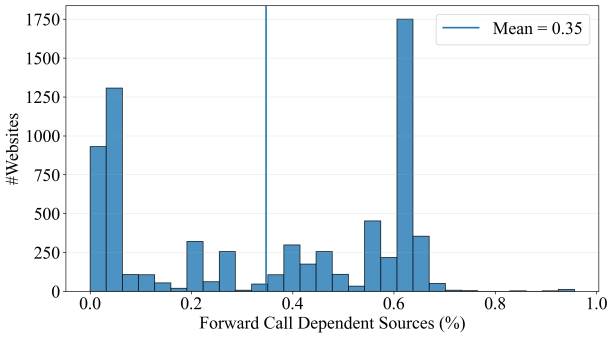

Fig. 3: Distribution of forward-call-dependent cookie sources across websites.

highest recall (Table II and Table III). Note that the baselines do not locate specific cookie sources. Therefore, the false positives at the source level are inherently undefined, making source-level precision incomparable. We also performed a script-level comparison to quantify the precision and recall for source script identification. Notably, CookieTrace still exhibited the highest precision and recall, representing a substantial improvement over existing tools.

Explaining the performance gain. The superior performance stems directly from a key advantage of CookieTrace. While the baselines rely solely on the call stack for attribution, CookieTrace further traces the forward function calls beyond the immediate call stack boundary. To quantify the real-world prevalence of forward function call, we measured its occurrence across all 6,995 websites, and assessed how many cookie sources were defined in such calls.

We observed that forward tracking was triggered very frequently, with an average of 18.02 forward function calls per website (Fig. 2). These calls were absent from the call stacks yet actively involved in the non-compliant cookie writes.

More importantly, we assessed the necessity of tracking the forward function calls by measuring the proportion of cookie sources whose discovery *depends* on following such calls. We define a source as "forward-call-dependent" if it originates in a forward function. For example, a cookie name may be a return value like `cookieName = getPrefixedName(id)`, where `getPrefixedName` is a forward function outside the immediate call stack. Without tracking this function call, the sources of the cookie name remain opaque. As illustrated in Fig. 3, our measurement revealed that a striking 50% of websites have more than 35% of their cookie sources being forward-call-dependent. This demonstrates that forward tracking is indispensable rather than an optional enhancement for complete cookie source tracking.

Takeaway. CookieTrace demonstrated strong effectiveness, achieving an average precision of 98.7% and recall of 100% in locating the cookie sources. For source script identification, it achieved a perfect precision and recall of 100%. It uncovered significantly more cookie sources than existing stack-based tools, primarily because it resolved forward function calls inherently missed by prior approaches.

C. *Efficiency of* CookieTrace *(RQ2)*

To evaluate the efficiency of CookieTrace, we recorded the runtime overhead it introduces. Specifically, we measured the elapsed time from initial navigation to the full page loading where `document.readyState == "complete"`. The timeout is empirically set as 10 minutes (600 seconds). As a baseline, we measured the same metric on an uninstrumented build of Chromium. To ensure fairness, all runtime measurements were conducted on the same hardware and network environment and repeated three times, with the mean reported.

As shown in Fig. 4, the average page loading time we observed with CookieTrace is 79.34 seconds, corresponding to a 2.71× increase over the vanilla baseline. Over 95% of this overhead originated from the runtime recording stage (S1), where CookieTrace continuously updated both the object map and the forward function map before the target cookie is written. In contrast, once a cookie write is detected (S2), the subsequent DFG construction (S3) and cookie source identification (S4) introduced only negligible overhead. The overhead is lower than that reported for most state-of-the-art data flow analysis tools for JavaScript [46]. The only exception, Foxhound [45], taints only selected strings. Therefore, it cannot capture general data dependencies, causing taint to be lost across data type conversions. It also does not capture control dependencies, which can be analyzed by CookieTrace. Importantly, unlike detection tasks that require real-time responsiveness, cookie compliance triage is an *offline one-time* analysis task. Overall, our experiment results sufficiently demonstrate the practical feasibility of our tool for large-scale analysis.

1) Ablation study: We compare CookieTrace with two eager baselines below to evaluate the contribution of selectivity and laziness, respectively. We omit the lazy-exhaustive baseline because it still needs to analyze all function invocations, while deferred DFG construction only adds additional runtime recording overhead. Therefore, it must be less efficient than eager-exhaustive. The comparison was conducted on 100 randomly sampled websites with a 10 minute timeout.

Eager-exhaustive baseline. The eager-exhaustive baseline constructs a DFG for every JavaScript function invocation during page loading. We observed over 100,000 function invocations on every website, and *none* completed loading

within the 10-minute timeout. The browser also frequently became unresponsive and crashed due to excessive resource consumption. In contrast, COOKIETRACE completed page loading in 67.28 seconds on average on the same websites. These results demonstrate that exhaustively constructing DFGs is impractical in real-world page executions, highlighting the necessity of selective DFG construction.

Eager-selective baseline. To isolate the contribution of laziness, this baseline assumes access to an oracle that provides exact prior knowledge of the function invocations eventually required for S4, which is impractical to obtain in real-world executions. We derive the oracle using COOKIETRACE because the oracle serves only to fix the invocation set for isolating lazy construction, rather than to provide an independent ground truth. Obtaining the same exact invocation set independently would otherwise require extensive manual analysis or an equivalent provenance analysis. See §B for more details.

We use Web Page Replay (WPR) [47] to derive the oracle through a controlled record-replay experiment. The first stage uses COOKIETRACE to record the exact function invocations eventually required for source tracking, forming the oracle. During the second stage, the eager-selective baseline eagerly constructs DFGs for invocations matching the oracle. We finally verify that both executions analyze the same invocations and thus differ only in DFG-construction timing.

In this ablation, we compare only the costs of DFG construction (S3) and source identification (S4), because the oracle externally fixes the relevant invocation set. Including runtime recording (S1) cost would not provide a fair comparison, as S1 is the cost COOKIETRACE incurs to discover the set at run time, whereas the baseline receives the information for free. For S3–S4, COOKIETRACE and eager-selective baseline incurred an average overhead of 3.70s and 3.61s, respectively. This shows that, given the same selected function invocations, deferring DFG construction alone does not directly reduce runtime overhead. The major efficiency gain instead comes from the selectivity. However, such prior knowledge of the relevant invocation set is unavailable in practice. Laziness enables COOKIETRACE to identify relevant invocations during backtracking and constructs their DFGs on demand, making selective analysis practical without such prior knowledge.

2) Comparison with taint-based analysis: We use Arcanum as a representative modern in-browser taint-tracking baseline as it shares the Chromium/V8 architecture with COOKIETRACE. This avoids cross-browser implementation differences that could otherwise affect the comparison. In contrast, Foxhound is implemented on Firefox/SpiderMonkey and achieves lower overhead under a more restricted tainting scope, making its performance not directly comparable to our setting.

As discussed in III-B, potential cookie sources are not known a priori and may include arbitrary JavaScript values. To keep the experiment tractable and conservative, we adapt Arcanum to our task by: 1) intercepting assignments to `document.cookie` as sinks; and 2) treating all JavaScript string literals as taint sources upon its first runtime evaluation, with taint labels propagated through assignments. At the target cookie write sink, we collect the propagated labels and map them back to the source literals. Note that this setup under-approximates the full taint-tracking workload for arbitrary cookie sources, including non-string values (*e.g.*, numbers).

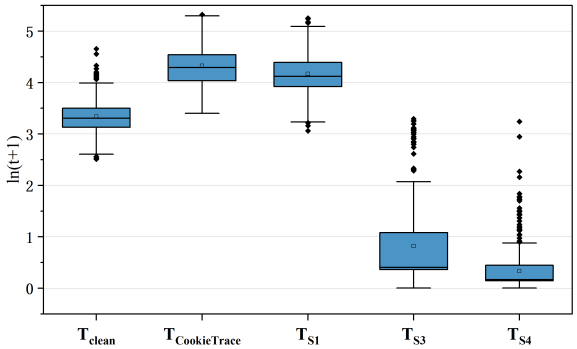

Fig. 4: Average page loading time for all websites in the dataset. The time t has been transformed to ln(t + 1) to improve visibility of the distribution.

On the same 100 websites and 10-minute timeout, no adapted Arcanum executions reached the target cookie sink. All timed out while propagating the taint labels. This shows that even with sources restricted to string literals, the large number of tainted runtime values imposed substantial propagation overhead. This observation is consistent with prior works [46], [48], which also report substantial runtime overhead in existing JavaScript taint-tracking tools. More generally, prior work [49] attributes the high overhead of dynamic taint analysis to expensive online taint propagation and intensive runtime information collection. In our experiments, the cost is amplified by treating all string literals as taint sources, spreading the taint to a large number of runtime values. In contrast, COOKIETRACE constructs DFGs only for relevant function invocations and backtracks only values on the dependency paths of the target cookie, substantially reducing the analysis scope.

Takeaway. COOKIETRACE introduces moderate overhead acceptable for offline, one-time cookie-compliance triage. Selectivity provides the main efficiency gain, while laziness enables selective analysis without prior oracle knowledge. Taint tracking incurs substantially higher overhead.

D. Structural Opacity (RQ3)

To understand the practical challenges in attributing cookie writes to specific sources, we characterized the structural opacity of cookie source identification. We quantify the opacity in two metrics: 1) the length of propagation paths, defined as the number of data propagation steps (*e.g.*, via assignments, passing function arguments, and returns) between a cookie source and the final cookie write, and 2) the distribution of cookie sources across functions.

1) Propagation path length: We measured the propagation path length per identified cookie source. Across the 6,995 websites, the average propagation length was 17.55. Fig. 5 illustrates the distribution, revealing that a significant portion of

TABLE IV: Top prevalent source scripts identified by CookieTrace.

Consent Cookie		Unnecessary Cookie	
Script	**#Website**	**Script**	**#Website**
https://eu-us-cdn.consentmanager.net/delivery/js/cmp_final.min.js	756	https://www.googletagmanager.com/gtm.js?id=...	2,290
https://eu-us-cdn.consentmanager.net/delivery/customdata/bv...A3.js	756	https://www.googletagmanager.com/gtag/js?id=...	1,910
https://cdn.consentmanager.net/delivery/js/cmp_final.min.js	432	https://ak.sail-horizon.com/spm/spm.v1.min.js	410
https://cdn.consentmanager.net/delivery/customdata/bV...Y5.js	432	https://a.omappapi.com/app/js/api.min.js	300
other 41 scripts	99	https://analytics.tiktok.com/i18n/.../main...MQ.js	290

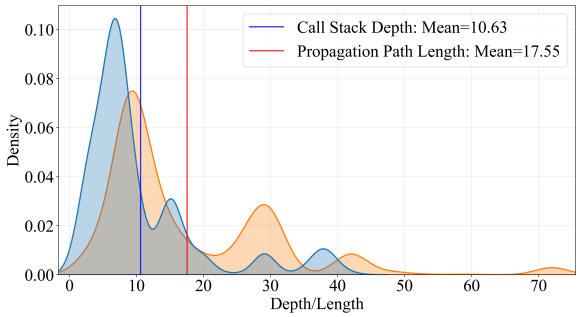

Fig. 5: Propagation path length vs. call stack depth distribution.

sources traversed lengthy and circuitous paths. Notably, 81.6% of sources had a propagation length greater than the average call stack depth of 10.63, indicating that their construction involved complicated intra-procedural flows.

2) Distribution of cookie sources across functions: Across the 6,995 websites, each non-compliant cookie write involved on average 117.02 sources distributed across 14.47 different functions. Only 49.16% cookie sources were defined within the same function that ultimately performed the cookie write , and only 93 (1.32%) non-compliant cookie originated from sources within a single function altogether.

Takeaway. Non-compliant cookies source identification is practically challenging. The sources propagated over complex data flows and were distributed widely across diverse functions.

E. Responsibility Concentration (RQ4)

To investigate where responsibility lies for deploying non-compliant cookies, we collected and analyzed all scripts that contained cookie sources (*i.e.*, the source scripts). Overall, we identified 522 and 45 distinct scripts responsible for non-compliant unnecessary cookies and consent cookies, respectively. The top prevalent scripts are listed in Table IV. Notably, we observed that scripts for non-compliant unnecessary cookies, while diverse, were largely operated by a few number of organizations (*e.g.*, Google Tag Manager, TikTok). In contrast, the culprits for non-compliant consent cookie writes exhibited a long-tail distribution, dominated by a single provider consentmanager.net, writing 92.31% of such cookies. The remaining violations were caused by 41 other scripts, which served only 99 websites. It is important to note that we do not aim to draw accusatory conclusions, as the violations could also stem from implementation practices rather than deliberate misuse. Nonetheless, reporting the prevalence is still valuable to characterize the ecosystem and prioritize follow-up analysis.

To further understand the functional roles of these scripts, we classified them according to two widely adopted reference lists: 1) the IAB's official CMP list [50] for identifying Consent Management Platform (CMP) scripts, and WhoTracks.Me [51] for script semantic categorization (*e.g.*, advertising, analytics, social media). Specifically, we matched the script domain names against the lists to determine their categories. The results are shown in Fig. 6. We were not able to reliably classify the source scripts on 274 websites, as the scripts did not match any of the predefined functional categories. Such scripts were primarily hosted under the first-party domains and were site-specific. Since they appeared on only 3.92% of the target websites, they would not alter the overall attribution patterns or the responsibility concentration observed in the ecosystem. For the remaining 268 scripts on 6,721 websites, we classified 76 as "site analytics", 60 as "advertising", and 57 as "CMPs". Note that a single script may belong to multiple categories. For example, https://snap.licdn.com/li.lms-analytics/insight.min.js was classified as both "misc" and "site analytics". Overall, 10 scripts were classified into multiple categories.

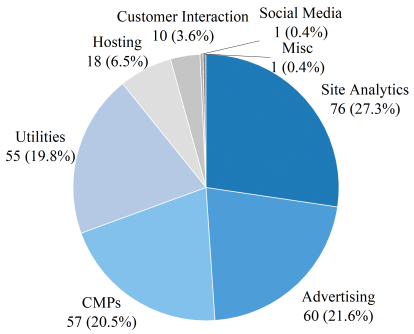

Fig. 6: Proportion of different categories of scripts.

Takeaway. We observed a striking asymmetry in responsibility concentration for the two types of cookie violations. Our categorization further revealed the prevalent roles of these scripts include site analytics, advertising, and consent management, indicating that violations were often associated with a small set of functional components.

TABLE V: Breakdown of the collaboration among source scripts.

	Category	#Consent	#Unnecessary
Single-script	First-party	0	499
	Third-party	7	6,202
Multi-scripts	First-party + First-party	1	23
	First-party + Third-party	12	499
	Third-party + Third-party	1,267	1,273

F. Collaboration Measurement (RQ5)

In our analysis of 9,783 non-compliant cookies, we observed that a substantial number of cookies originated from sources in multiple scripts. We refer to this as script collaboration. To investigate how scripts collaborated in constructing the non-compliant cookies, we quantified the proportion of multi-script involvement. We also classified the scripts as first-party or third-party based on their domains, and measured the collaboration among them. The results are presented in Table V, where the numbers represent the count of cookies in each category.

Specifically, among 1,287 non-compliant consent cookies, only 7 (0.54%) had sources provided by a single script. All 7 were defined by third-party scripts. Of the remaining 1,280 consent cookies, 1,267 (98.98%) and 1 were constructed by multiple third- and first-party scripts, respectively. The third-party scripts were predominantly loaded from consentmanager.net. This suggests that consentmanager might be favoring a distributed, multi-script implementation strategy. Only 12 (0.93%) cookies involved collaboration between first- and third-party scripts, suggesting that consent cookie violations were typically confined within a single party.

In contrast, unnecessary non-compliant cookies exhibited a stronger tendency toward single-script construction. We observed 6,202 (73.00%) defined by a single third-party script, and 499 (5.87%) by a single first-party script. This may stem from the relatively self-contained management logic of personalization or marketing cookies. Furthermore, 1,795 unnecessary cookies involved more first- and third-party collaboration than consent cookies, accounting for 21.13% of violations. This reflects a more varied—but often less regulated—ecosystem that could easily lead to non-compliance.

1) Case Studies: We now showcase how single-script and multi-scripts non-compliant cookie writes occur in practice.

Single-script non-compliant cookie writes. Although all sources in these cases reside in a single script, script-level attribution remains insufficient for remediation because it collapses distinct internal data flows. Real-world scripts are often large, obfuscated, and interact with other scripts, making manual auditing time-consuming and error-prone. For example, on `abcplay.online` (Listing 2), the responsible script contained over 700K heavily minified characters. At the cookie write, we observed a stack with 26 function calls, and 42 forward functions invoked during the execution, reflecting the substantial complexity of the single responsible script. Additionally, the existing stack-based tools also offer incomplete insights. For instance, through forward tracking, CookieTrace revealed two critical call chains—`Gs()->Fs()->fs()` (Line 9->3->2), `Gs()->Gb()->Fb()` (Line 9->8->7)—that remain invisible from the call stack, proving the necessity to design an accurate analysis framework.

```
1  // https://www.googletagmanager.com/gtag/js?id=...
2  function fs(a) { /* hash computation */ }
3  function Fs(a) {
4      var b = Math.round(Math.random() * 2147483647);
5      return a ? String(b ^ fs(a) & 2147483647) :
       ↪  String(b)
6  }
7  function Fb() { return new Date(Date.now()) }
8  function Gb() { return Fb().getTime() }
9  function Gs(a) {
10     return [Fs(a), Math.round(Gb() / 1E3)].join(".")
11 }
12 function nt(a, b) {
13     var f = ot(a.prefix), g = Gs();
14     us(f, g);
15 }
16 function us(a, b) {
17     document.cookie = a + b;
18 }
```

Listing 2: Simplified cookie write for `_gcl_au` on `abcplay.online`.

```
1  // https://cdn.consentmanager.net/.../bV...zgx.js
2  window.cmp_config_data.vendors = [{...},...,{...}];
3
4  // https://b.delivery.consentmanager.net/.../cmp.php
5  window.cmp_config_data = { "intID": 86685, ...};
6
7  // https://cdn.consentmanager.net/.../cmp_final.min.js
8  this.getCookieNames = function() {
9     var f = {};
10    if (i == 5) { f.name = "__cmpconsentx" + j; ...}
11    return f;
12 }
13 this.setConsent_ = function(q, u) {
14     var Y = G.toString();
15     w.storeConsent(Y)
16 }
17 this.storeConsent = function(v) {
18    var i = this.getCookieNames();
19    if (v != "") { this.writeCookie(i.name, v, ...) }
20 }
21 this.writeCookie = function(n, v, j, m, i) {
22     document.cookie = n + "=" + v + ";expires=" +
       ↪  j.toUTCString() + ";path=" + m + ";domain=" + i
23 }
```

Listing 3: Simplified cookie write for `__cmpconsentx86685` on `britannica.com`.

Multi-scripts non-compliant cookie writes. In these cases, cookie sources and writes span multiple scripts, further complicating the analysis. For example, on `britannica.com`, three third-party scripts from `consentmanager.net` collaboratively constructed the non-compliant consent cookie (Listing 3). From the write at Line 22, CookieTrace tracked backward to `storeConsent` (Lines 17–20) and further to `getCookieNames()` (Line 18), revealing that the cookie name originated from the prefix `"__cmpconsentx"` (Line 10) and the suffix `j` from `window.cmp_config_data` (Line 5). The cookie value was produced by `G.toString()` in `setConsent_` (Line 14), where `G` was derived from `window.cmp_config_data.vendors`, defined in another third-party script (Line 2). While the call stack contained 9 calls, CookieTrace identified 21 additional forward calls involved in constructing the cookie.

Takeaway. We observed distinct collaboration patterns between consent and unnecessary cookies violations. Consent cookie violations were predominantly produced by coordinated third-party scripts, whereas unnecessary ones often originated from a single third-party script. However, neither existing stack-based tools nor manual auditing can readily and precisely reason about the violations, motivating an analysis framework like CookieTrace.

VI. Discussion

Evaluation Scope. We have focused on tracing cookies stored before user interactions. Such cookies are not violations under CCPA, and extending the analysis to post-consent scenarios would require manual interactions or reliable automation tools. Existing automated cookie-banner interaction tools can introduce noise due to incorrect interactions [52]. While a comprehensive study of all categories of non-compliant cookies is beyond the scope of this work, our framework can in principle be applied to analyze any target cookies. We plan to broaden the evaluation scope in future work.

Broader Applications. CookieTrace is not limited to cookie compliance analysis. It could also be extended to study how scripts persist identifiers in local storage, exfiltrate data through network requests, or exchange information across contexts via messaging and fingerprinting APIs, *etc.* Such capabilities would enable analysis of a broader range of web security problems, including hidden profiling and misuse of sensitive identifiers. We will explore other applications next.

Potential Incentives and Consequences. Our findings suggest that non-compliant cookies may be associated with incentives to sustain advertising and monetization, while reducing the compliance overhead. Most non-compliant cookies encoded consents for purposes or legitimate interests closely tied to advertising and analytics, often involving multiple vendors. For unnecessary cookies, although they frequently originated from single scripts, both first- and third-party scripts can be responsible. The observed collaboration further suggests that the incentives of different parties were often aligned rather than conflicting. Together, our study highlights the potential combination of economic incentives, and proves the technical complexity and fragmented responsibility distribution, which complicate the compliance enforcement in practice.

Implication for Mitigation. Our findings show that both first-party and third-party scripts can be responsible for non-compliant cookie writes, and that even CMP scripts frequently introduced the violations. This suggests that effective mitigation requires a combination of technical and regulatory measures. On the regulatory side, our results indicate that auditing scripts—not just cookies—should become part of compliance inspection, and our backtracking approach could support regulators in identifying the responsible code. On the technical side, operators should defer cookie-writing scripts until explicit consent is obtained, and adopt clearer disclosures and user controls to reduce information asymmetry.

VII. Related Work

Privacy Compliance Analysis. Existing studies have extensively measured the GDPR compliance [53], [54], [55], and developed various tools to detect and mitigate cookie-related violations [56], [57], [58]. Many studies [59], [60], [61], [31] revealed that consent dialogs commonly nudge users toward consenting and often omit or hide a clear "reject all" option. In this work, we aim to perform a more in-depth analysis of cookie violations, by tracing the cookie sources with a novel dynamic data flow backtracking framework.

In-browser Data Flow Analysis. Dynamic in-browser JavaScript analysis [62], [63], [64], [65] monitors data flows at runtime. Mystique [66], Arcanum [33], PanoptiChrome [40], Foxhound [45], and FP-tracer [67] use dynamic in-browser taint tracking for security and privacy analysis. CoCo [68] implements concurrent taint propagation with concurrent abstract interpretation [69], [70], [71] to find exploitable vulnerabilities in extensions. ProbeTheProto [41] tracks whether prototype pollution vulnerabilities [72] can affect sensitive operations. Others use network traffic [73] or source code instrumentation [74], [75] to track JavaScript data flows. However, most dynamic JavaScript tools remain incompatible with the modern Web, suffer transparency and coverage limitations, and incur high overhead [46]. No existing method is applicable for identifying cookie sources when a violation occurs.

Static JavaScript Data Flow Analysis. Prior static analyses target browser fingerprinting [76], browser-extension data flows [77], [78], and general JavaScript or lightweight application platforms [79], [80], [81]. However, real-world web pages involve dynamically loaded or generated code, asynchronous events, minification and obfuscation, which substantially limit the applicability of static analysis in our setting [41], [40], [42], [82], [83], [84]. We therefore adopt dynamic in-browser analysis to cope with the issues.

Inter-procedural Slicing and Selective Data-Flow Analysis. Prior techniques share high-level ideas with CookieTrace in inter-procedural dependency tracking and reducing the scope of data-flow analysis. Horwitz *et al.* [10] perform context-sensitive inter-procedural slicing using system dependence graphs. Agrawal and Horgan [11] compute dynamic slices from execution-derived dependence graphs. Duesterwald *et al.* [12] perform demand-driven inter-procedural data-flow analysis, and SelectiveTaint [13] reduces taint-tracking overhead by selectively instrumenting instructions based on predefined taint sources. FlowDist [85] uses coarse-grained method-level information-flow paths to restrict subsequent fine-grained dependence analysis. Falcon [86] resolves transitive data dependencies on demand over pre-computed symbolic value-flow graphs. These techniques either operate on available program representations or determine the analysis scope from predefined taint sources, which are unavailable to CookieTrace. In contrast, CookieTrace defers DFG construction and progressively identifies the required function invocations during backtracking, expanding beyond the current call stack only when upon dependencies on already-returned function invocations.

VIII. Conclusion

In this work, we present a systematic study of the sources of non-compliant cookies on the web. We introduce CookieTrace, an in-browser data flow backtracking analysis framework, that performs selective and on-demand data flow graph (DFG) construction at run time. Using CookieTrace, we analyzed 9,783 non-compliant cookies across 6,995 websites. It demonstrated high effectiveness in tracking the cookie sources, achieving an average precision of 98.7% and recall of 100%. The selective DFG construction also dramatically reduces the computational overhead. Our analysis revealed that non-compliant cookies often originate from a small set of functional components, yet the underlying data flows are highly complex and frequently span multiple scripts. The findings underscore the need for precise cookie provenance tracking to support regulatory auditing, enable effective triage of cookie violations, and inform forensic analysis. We believe CookieTrace also provides a viable foundation to investigate the root causes of broader web security issues, thereby benefiting the research and practitioner community.

Acknowledgment

We thank the reviewers and shepherd for their helpful comments and feedback. This work was supported by the National Natural Science Foundation of China (No. 62402423), the Fundamental Research Funds for the Central Universities (No. 226202400143), and NTU startup grant.

Ethics Considerations

We identified website operators as the primary stakeholder that can be directly affected by our research, as we used CookieTrace to visit their websites and analyze cookie write behaviors. Our methodology minimized potential harms by recording only a single live interaction per website, thereby avoiding repeated requests that could impose unnecessary server load or bandwidth costs. We further reduced the operational impact by avoiding authentication attempts, form submissions, or interactive clicks, and by relying solely on passive page loads. Each visit was performed with a fresh, isolated browser profile containing no browsing history, extensions, or stored credentials, ensuring that our interactions appeared as standard, anonymous page visits. Our study did not involve human subjects or access to private user data. All analysis was restricted to publicly observable scripts and website behaviors accessible to any visitor.

A further ethical concern relates to the potential reputational impact of our findings. Identifying scripts and websites responsible for violations may be mistakenly perceived as accusatory, even though our intent is purely scientific. To mitigate this, we follow a responsible disclosure process: we have notified relevant script providers and affected websites of the issues we found, without releasing any details publicly. Following RFC 2142 [87] and Stock et al. [88], we contacted them via the role-based email addresses, including `info@`, `security@`, `abuse@`, `support@`, and `postmaster@`. At the time of writing the paper, we have completed all notification and several operators have acknowledged our reports. This ensures that operators have an opportunity to remediate the issues prior to wider dissemination. Among the 6,995 websites, we received 2,641 (37.8%) responses, most of which were automatic replies or ticket acknowledgements. Among them, 141 explicitly engaged with the reported issues, including 4 confirmations and 1 planned fix. Most responses were automated acknowledgements, suggesting challenges in notification of compliance problems in practice.

We also acknowledge that CookieTrace, as an in-browser data flow backtracking framework, could in principle be extended and misused to analyze or reverse-engineer third-party scripts. In line with requirements, we provide an artifact to enable reproducibility, but it is released in a research-oriented form: we excluded the raw traces, full cookie values, or other potentially sensitive data, and we document its intended use for compliance and security studies rather than adversarial purposes. By doing so, we balance transparency and reproducibility with the need to limit relevant risks.

Finally, the target website selection from the Tranco top 500K list ensures the representativeness, avoiding excessive focus on any particular group of users or service providers. We believe the societal benefits of this work—improving transparency, strengthening accountability, and supporting privacy protection practices—outweigh the limited risks, and that responsible publication can provide both regulators and operators with actionable insights to remediate the cookie violations.

References

[1] P. Regulation, "General data protection regulation," *Intouch*, vol. 25, pp. 1–5, 2018.

[2] European Parliament and Council of the European Union, "Directive 2002/58/EC of the European Parliament and of the Council of 12 July 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector (Directive on privacy and electronic communications)," https://eur-lex.europa.eu/eli/dir/2002/58/2009-12-19/eng, 2009, consolidated version of 19 December 2009.

[3] T. Urban, D. Tatang, M. Degeling, T. Holz, and N. Pohlmann, "Measuring the impact of the gdpr on data sharing in ad networks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 222–235.

[4] J. Sørensen and S. Kosta, "Before and after gdpr: The changes in third party presence at public and private european websites," in *The World Wide Web Conference*, 2019, pp. 1590–1600.

[5] T. Sakamoto and M. Matsunaga, "After gdpr, still tracking or not? understanding opt-out states for online behavioral advertising," in *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2019, pp. 92–99.

[6] D. Bollinger, K. Kubicek, C. Cotrini, and D. A. Basin, "Automating cookie consent and GDPR violation detection," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 2893–2910. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bollinger

[7] M. Zhang, W. Meng, Y. Zhou, and K. Ren, "Cschecker: Revisiting GDPR and CCPA compliance of cookie banners on the web," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 174:1–174:12. [Online]. Available: https://doi.org/10.1145/3597503.3639159

[8] I. Sanchez-Rola, M. Dell'Amico, D. Balzarotti, P.-A. Vervier, and L. Bilge, "Journey to the center of the cookie ecosystem: Unraveling actors' roles and relationships," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1990–2004.

[9] K. Drakonakis, S. Ioannidis, and J. Polakis, "Dredging the river styx: Fortifying the web through robust and real-time script attribution," in *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2025, pp. 188–208.

[10] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, 1988, pp. 35–46.

[11] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPlan Notices*, vol. 25, no. 6, pp. 246–256, 1990.

[12] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical framework for demand-driven interprocedural data flow analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 6, pp. 992–1030, 1997.

[13] S. Chen, Z. Lin, and Y. Zhang, "{SelectiveTaint}: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1665–1682.

[14] "Tranco Top Sites Ranking," 2025, https://tranco-list.eu.

[15] J. Wolff and N. Atallah, "Early gdpr penalties: Analysis of implementation and fines through may 2020," *Journal of Information Policy*, vol. 11, pp. 63–103, 2021.

[16] "California Consumer Privacy Act (CCPA)," 2020, https://en.wikipedia.org/wiki/California_Consumer_Privacy_Act.

[17] WHATWG, "Cookie store api specification," https://cookiestore.spec.whatwg.org/, 2026, living Standard; accessed: 2026-01-13.

[18] P. Nikkhah Bahrami, A. Fass, and Z. Shafiq, "Cookieguard: Characterizing and isolating the first-party cookie jar," in *Proceedings of the 2025 ACM Internet Measurement Conference*, 2025, pp. 645–661.

[19] P. Bernardo, L. Veronese, V. Dalla Valle, S. Calzavara, M. Squarcina, P. Adão, and M. Maffei, "Web platform threats: Automated detection of web security issues with {WPT}," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 757–774.

[20] Q. Chen, P. Ilia, M. Polychronakis, and A. Kapravelos, "Cookie swap party: Abusing first-party cookies for web tracking," in *Proceedings of the Web Conference 2021*, 2021, pp. 2117–2129.

[21] I. Fouad, C. Santos, A. Legout, and N. Bielova, "My cookie is a phoenix: detection, measurement, and lawfulness of cookie respawning with browser fingerprinting," in *PETS 2022-22nd Privacy Enhancing Technologies Symposium*, 2022.

[22] S. Munir, S. Siby, U. Iqbal, S. Englehardt, Z. Shafiq, and C. Troncoso, "Cookiegraph: Understanding and detecting first-party tracking cookies," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3490–3504.

[23] "OWASP community wiki: Httponly," https://owasp.org/www-community/HttpOnly, 2025, accessed: 2025-08.

[24] "OWASP community wiki: Samesite," https://owasp.org/www-community/SameSite, 2025, accessed: 2025-08.

[25] "OWASP community wiki: Secure cookie attribute," https://owasp.org/www-community/controls/SecureCookieAttribute, 2025, accessed: 2025-08.

[26] Q. Chen, P. Ilia, M. Polychronakis, and A. Kapravelos, "Cookie swap party: Abusing first-party cookies for web tracking," in *Proceedings of the Web Conference 2021*, 2021, pp. 2117–2129.

[27] S. Chen, J. McCracken, K. Lu, T. Wang, and T. Hou, "Taking a look into the cookie jar: A comprehensive study towards the security of web cookies," in *Proceedings of the Twenty-fourth International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, 2023, pp. 474–479.

[28] "IAB Europe," https://iabeurope.eu/, 2025, accessed: 2025-08.

[29] "IAB Europe Transparency & Consent Framework," https://iabeurope.eu/transparency-consent-framework/, 2025, accessed: 2025-08.

[30] "Global Privacy Protocol (GPP)," https://iabtechlab.com/gpp/, 2025, accessed: 2025-08.

[31] C. Matte, N. Bielova, and C. Santos, "Do cookie banners respect my choice?: Measuring legal compliance of banners from iab europe's transparency and consent framework," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 791–809.

[32] HTTP Archive. (2024) HTTP Archive - JavaScript. HTTP Archive. [Online]. Available: https://almanac.httparchive.org/en/2024/javascript

[33] Q. Xie, M. V. K. Murali, P. Pearce, and F. Li, "Arcanum: Detecting and evaluating the privacy risks of browser extensions on web pages and web content," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4607–4624.

[34] "Json for modern c++," 2025, https://github.com/nlohmann/json.

[35] "Open-Cookie-Database," 2025, https://github.com/jkwakman/Open-Cookie-Database.

[36] IAB Europe, "Transparency & consent framework (tcf)," On-line: https://iabeurope.eu/iab-europe-transparency-consent-framework-policies, 2025, accessed: 2026-01-12.

[37] M. Ferreira, M. Monteiro, T. Brito, M. E. Coimbra, N. Santos, L. Jia, and J. F. Santos, "Efficient static vulnerability analysis for javascript with multiversion dependency graphs," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 417–441, 2024.

[38] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1059–1076.

[39] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node. js vulnerabilities via object dependence graph and query," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 143–160.

[40] R. Kanyal and S. R. Sarangi, "Panoptichrome: A modern in-browser taint analysis framework," in *Proceedings of the ACM Web Conference 2024*, 2024, pp. 1914–1922.

[41] Z. Kang, S. Li, and Y. Cao, "Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites." in *NDSS*, 2022.

[42] P. Skolka, C.-A. Staicu, and M. Pradel, "Anything to hide? studying minified and obfuscated code in the web," in *The world wide web conference*, 2019, pp. 1735–1746.

[43] A. D. Householder, G. Wassermann, A. Manion, and C. King, "The CERT Guide to Coordinated Vulnerability Disclosure," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2017-SR-022, Aug. 2017.

[44] Google LLC, "Chrome devtools," https://developer.chrome.com/docs/devtools/, 2025.

[45] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns, "Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2022, pp. 236–250.

[46] S. Calzavara, S. Casarin, and R. Focardi, "Dynamic security analysis of javascript: Are we there yet?" in *Proceedings of the ACM on Web Conference 2025*, 2025, pp. 1105–1115.

[47] Chromium Project, "Web Page Replay," https://chromium.googlesource.com/webpagereplay/, accessed: Aug. 2026.

[48] M. W. Aldrich, A. Turcotte, M. Blanco, and F. Tip, "Augur: Dynamic taint analysis for asynchronous javascript," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–4.

[49] Y. Zhang, T. Liu, Y. Wang, Y. Qi, K. Ji, J. Tang, X. Wang, X. Li, and Z. Zuo, "Hardtaint: Production-run dynamic taint analysis via selective hardware tracing," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1615–1640, 2024.

[50] "Cmp list," 2025, https://iabeurope.eu/cmp-list.

[51] A. Karaj, S. Macbeth, R. Berson, and J. M. Pujol, "Whotracks. me: Shedding light on the opaque world of online tracking," *arXiv preprint arXiv:1804.08959*, 2018.

[52] N. Demir, T. Urban, N. Pohlmann, and C. Wressnegger, "A large-scale study of cookie banner interaction tools and their impact on users' privacy," *Proceedings on Privacy Enhancing Technologies*, 2024.

[53] T. Libert, L. Graves, and R. K. Nielsen, "Changes in third-party content on european news websites after gdpr," 2018.

[54] E. Papadogiannakis, P. Papadopoulos, N. Kourtellis, and E. P. Markatos, "User tracking in the post-cookie era: How websites bypass gdpr consent to track users," in *Proceedings of the web conference 2021*, 2021, pp. 2130–2141.

[55] M. Nouwens, J. B. Kristensen, K. Maalt, and R. Bagge, "A cross-country analysis of GDPR cookie banners and flexible methods for scraping them," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, CHI 2025, YokohamaJapan, 26 April 2025- 1 May 2025*, N. Yamashita, V. Evers, K. Yatani, S. X. Ding, B. Lee, M. Chetty, and P. O. T. Dugas, Eds. ACM, 2025, pp. 858:1–858:28. [Online]. Available: https://doi.org/10.1145/3706598.3713648

[56] A. Bouhoula, K. Kubicek, A. Zac, C. Cotrini, and D. A. Basin, "Automated large-scale analysis of cookie notice compliance," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/bouhoula

[57] T. Libal, "Towards automated gdpr compliance checking," in *International Workshop on the Foundations of Trustworthy AI Integrating Learning, Optimization and Reasoning*. Springer, 2020, pp. 3–19.

[58] Y.-S. Martin and A. Kung, "Methods and tools for gdpr compliance through privacy and data protection engineering," in *2018 IEEE European symposium on security and privacy workshops (EuroS&PW)*. IEEE, 2018, pp. 108–111.

[59] G. Kampanos and S. F. Shahandashti, "Accept all: The landscape of cookie banners in greece and the uk," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2021, pp. 213–227.

[60] M. Nouwens, I. Liccardi, M. Veale, D. Karger, and L. Kagal, "Dark patterns after the gdpr: Scraping consent pop-ups and demonstrating their influence," in *Proceedings of the 2020 CHI conference on human factors in computing systems*, 2020, pp. 1–13.

[61] P. Vallina, Á. Feal, J. Gamba, N. Vallina-Rodriguez, and A. F. Anta, "Tales from the porn: A comprehensive privacy analysis of the web porn ecosystem," in *Proceedings of the Internet Measurement Conference*, 2019, pp. 245–258.

[62] D. Kong, J. Liu, C. Ni, D. Lo, and L. Bao, "More effective javascript breaking change detection via dynamic object relation graph," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 2340–2361, 2025.

[63] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87–97, 2009.

[64] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 196–206.

[65] E. Ogawa, T. Yamazaki, and R. Shioya, "Dynamic controllability analysis for preventing injection attacks," in *2024 IEEE 29th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2024, pp. 131–142.

[66] Q. Chen and A. Kapravelos, "Mystique: Uncovering information leakage from browser extensions," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1687–1700.

[67] S. Boussaha, L. Hock, M. Bermejo, R. C. Rumin, A. C. Rumin, D. Klein, M. Johns, L. Compagna, D. Antonioli, and T. Barber, "Fp-tracer: Fine-grained browser fingerprinting detection via taint-tracking and entropy-based thresholds," *Proceedings on Privacy Enhancing Technologies*, 2024.

[68] J. Yu, S. Li, J. Zhu, and Y. Cao, "Coco: Efficient browser extension vulnerability detection via coverage-guided, concurrent abstract interpretation," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2441–2455.

[69] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *International Static Analysis Symposium*. Springer, 2009, pp. 238–255.

[70] K. Dewey, V. Kashyap, and B. Hardekopf, "A parallel abstract interpreter for javascript," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 34–45.

[71] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1059–1076.

[72] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting node. js prototype pollution vulnerabilities via object lookup analysis," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 268–279.

[73] M. Weissbacher, E. Mariconti, G. Suarez-Tangil, G. Stringhini, W. Robertson, and E. Kirda, "Ex-ray: Detection of history-leaking browser extensions," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017, pp. 590–602.

[74] E. Ogawa, T. Yamazaki, and R. Shioya, "Dynamic possible source count analysis for data leakage prevention," in *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2024, pp. 98–111.

[75] B. Sayed, I. Traoré, and A. Abdelhalim, "If-transpiler: Inlining of hybrid flow-sensitive security monitor for javascript," *Computers & Security*, vol. 75, pp. 92–117, 2018.

[76] R. Zhao, "Toward the flow-centric detection of browser fingerprinting," *Computers & Security*, vol. 137, p. 103642, 2024.

[77] A. Fass, D. F. Somé, M. Backes, and B. Stock, "Doublex: Statically detecting vulnerable data flows in browser extensions at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1789–1804.

[78] M. M. Ahmadpanah, M. F. Gobbi, D. Hedin, J. Kinder, and A. Sabelfeld, "Codex: Contextual flow tracking for browser extensions," in *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy*, 2024, pp. 6–17.

[79] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "Jsai: A static analysis platform for javascript," in *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*, 2014, pp. 121–132.

[80] S. Meng, L. Wang, S. Wang, K. Wang, X. Xiao, G. Bai, and H. Wang, "Wemint: Tainting sensitive data leaks in wechat mini-programs," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1403–1415.

[81] S. Wang, Y. Li, K. Wang, Y. Liu, H. Li, Y. Liu, and H. Wang, "Miniscope: Automated ui exploration and privacy inconsistency detection of miniapps via two-phase iterative hybrid analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 6, pp. 1–29, 2025.

[82] M. R. Laursen, W. Xu, and A. Møller, "Reducing static analysis unsoundness with approximate interpretation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 1165–1188, 2024.

[83] H. Jiang, H. Lai, S. Wu, and B. Hua, "Jasfree: Grammar-free program analysis for javascript bytecode," in *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2024, pp. 326–337.

[84] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C.-A. Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–36, 2017.

[85] X. Fu and H. Cai, "{FlowDist}:{Multi-Staged}{Refinement-Based} dynamic information flow analysis for distributed software systems," in *30th USENIX security symposium (USENIX Security 21)*, 2021, pp. 2093–2110.

[86] P. Yao, J. Zhou, X. Xiao, Q. Shi, R. Wu, and C. Zhang, "Falcon: A fused approach to path-sensitive sparse data dependence analysis," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 567–592, 2024.

[87] D. Crocker, "Mailbox Names for Common Services, Roles and Functions," RFC 2142, May 1997. [Online]. Available: https://www.rfc-editor.org/info/rfc2142

[88] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of {Large-Scale} web vulnerability notification," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1015–1032.

APPENDIX

A. Open Science

All artifacts required to reproduce and evaluate the experiments and claims presented in this paper are publicly available. In particular, we open-source the implementation of COOKIETRACE at https://github.com/ZJU-Seclab/COOKIETRACE, together with the original and deduplicated ground truth dataset, effectiveness evaluation results on the original 160 sampled cookies and the 85 deduplicated cookies, and all scripts used for data processing.

B. Eager-selective Ablation Details

To ensure consistent executions between COOKIETRACE and the eager-selective baseline, we use Web Page Replay (WPR) [47] to record each page load and replay the same recorded responses for both executions.

During the first stage, COOKIETRACE records every dynamic function invocation whose DFG is eventually required for cookie source tracking. To distinguish different dynamic invocations, we characterize each invocation using its syntactic

information, including the script name and function source position, together with its dynamic calling context represented by the three immediate caller frames. The resulting invocation set forms the oracle used by the eager-selective baseline. During the replay stage, the eager-selective baseline matches runtime function invocations against the recorded oracle and eagerly constructs DFGs for the matched invocations. After execution, we compare the constructed DFG set with the oracle and confirm that all oracle-selected invocations are analyzed.

We construct the oracle using COOKIETRACE because the exact set of relevant function invocations cannot be known before the target cookie write occurs. Determining whether an invocation contributes to the cookie requires observing the sink and tracing the corresponding data dependencies backward. Therefore, an independent automatic oracle generator would still need to record runtime information and perform backward provenance analysis to identify the same invocation set, which essentially requires a source-tracking capability comparable to COOKIETRACE. Alternatively, constructing the oracle manually would require inspecting the cookie write, following each contributing value across assignments, arguments, and return values, and repeatedly identifying the corresponding dynamic function invocations until all relevant dependencies are covered, which would incur substantial manual effort.

C. Effectiveness Evaluation Details

Ground Truth Collection. To avoid circular validation, ground truth collection was conducted independently of COOKIETRACE. For each sampled non-compliant cookie, we used Chrome DevTools to set breakpoints at the corresponding cookie write operation, and inspected the runtime call stack when the target cookie was written. Starting from the top frame, we checked each function invocation on the stack and manually performed backward data flow tracking for every value that contributed to the cookie, until identifying its source. When a value was produced by a forward function call, we further navigated to the callee and recursively inspected the related data dependencies to collect additional cookie sources. For each contributing value, the inspection stopped until the backward tracking reached the initializing expression that introduced a literal value, such as a string literal, numeric literal, or object literal. The final ground truth maps each `target_cookie` to a `source_metadata_list`, including the culprit script URLs and concrete code locations of all identified cookie sources.

Correctness Checks. For each target cookie, we evaluated the collected ground truth source list by verifying whether every source accounts for a specific component of that cookie, such as the cookie name, value fragment, or attribute. A case was considered complete only when the recorded sources collectively accounted for all observed components of the target cookie, including the cookie name, value, and relevant attributes, *etc.* Two authors, both with over a year of experience in JavaScript analysis, independently established the ground truth and cross-checked the results. While manual validation cannot formally guarantee the absence of human error, this process helped improve the reliability of the ground truth in practice.

Manual Efforts. Although real-world scripts can be large and heavily minified, the manual task was substantial but remained manageable in practice. This is because the analysis was anchored at a concrete cookie write and its runtime context, rather than at the full program state of the website. For each target cookie, we used Chrome DevTools to hook the concrete write operation, which provided a clear starting point. DevTools also identified the runtime call stack at the moment of the write, which localized the inspection scope to the relevant execution context. The stopping criterion was also explicit (*i.e.*, reaching the initializing expression). The clear entry point and explicit stopping criterion made each manual inspection a bounded debugging task, rather than an open-ended exploration of all JavaScript execution paths or all scripts loaded by the page.

Importantly, the large number of sources did not translate into an equally large number of independent debugging tasks. In practice, many sources of the same cookie were introduced through shared configuration objects, arrays, literal tables, or common construction routines. Once the relevant runtime object, construction path, or script region was identified, multiple nearby sources could be verified within the same analysis context, rather than restarting the analysis from scratch. Nevertheless, the process still required careful inspection of runtime values, code locations, and construction logic, and was far from a simple counting task. Overall, the verification effort was substantial but still manageable, taking in total about 20 person-days.

Deduplication Criteria. We manually perform deduplication based on cookie writing behavior rather than cookie names or script URLs. Specifically, two samples are considered duplicates only when they have the same cookie name and identical control flow leading to the cookie write, including the same call sequence and intra-procedural conditional branches. Therefore, multiple cookies written by scripts from the same provider (domain) may remain in the deduplicated dataset if they write different cookies or follow different control flows. For example, on `bostonglobe.com`, both `BCSessionID` and `bc_tstgrp` are written by the same `cdn.blueconic.net/bostonglobemedia.js` script, but they are retained as separate samples because they have different cookie names.

D. Artifact Appendix

Chasing Cookies to the Source: In-browser Data Flow Backtracking for Cookie Compliance Triage

1) Description & Requirements: CookieTrace traces cookie writes back to contributing literals and scripts through in-browser data-flow backtracking. It extends Arcanum and Chromium 108.0.5359.71 with runtime maps, selective data-flow construction, forward-call tracking, and asynchronous stack stitching. The artifact includes browser modifications, the prebuilt browser, local examples, analysis scripts, and experimental records.

a) How to access: The complete artifact (version 2.0) is publicly archived on Zenodo:
Zenodo URL: https://zenodo.org/records/23643524
DOI: https://doi.org/10.5281/zenodo.23643524

Start with `README.md` at the extracted archive root. The custom browser and companion files are included in `ae/browser/sys/`; no separate browser download is required. `analysis/` contains offline calculations, `data/` their inputs, and `reference_results/` saved outputs for comparison.

b) Hardware dependencies: Use an x86-64 Linux desktop or laptop. Recommended provisioning is 8 CPU cores, 16 GB RAM, and 10 GB free disk space, with one browser worker at a time. No GPU or physical display is required.

c) Software dependencies: E1 uses Bash, Python 3.9+, and a local Linux Docker Engine. Its Dockerfile installs Debian 12, Xvfb, and browser libraries. E2, E3, and saved-classification aggregation use Python's standard library. Figure 6 reclassification also requires Node.js 18+, without npm packages. Internet access is needed for initial container construction and optional live-site tests; local examples and saved-data calculations work offline afterward.

d) Benchmarks: The source lists in `data/e2/selected_json/` and `data/e2/GT/` support 160 samples. The overhead CSV in `data/e3/` contains 9,783 timing rows covering 6,995 ranks. `data/e4/` contains the Figure 6 classification inputs and the two JSON data files for Tables IV–V. The ranking snapshot is `data/mapping/top-1m.csv`.

2) Artifact Installation & Configuration: Keep the bundled runtime intact, with its executable at `ae/browser/sys/chrome`. Run from the archive root:

```
(cd ae/browser &&
sha256sum -c BROWSER_FILES_SHA256SUMS)
chmod u+x ae/browser/sys/chrome \
ae/browser/sys/chrome_crashpad_handler
bash ae/run.sh --build
bash ae/run.sh --check
```

The build command constructs the dependency container, not Chromium. Each browser run uses Xvfb, a fresh profile, and a unique directory under `ae/results/`. A bind mount redirects the engine's fixed output location to the run's `raw/` directory. No research-server filesystem layout is required. The launcher accepts `--browser-dir`, `--output-dir`, and an optional `--proxy`.

To build from source, follow README: check out Chromium 108.0.5359.71, apply Arcanum's patches and build configuration, then apply `cookietrace.patch` in `src/v8/` and run `autoninja -C out/Arcanum chrome`.

3) Experiment Workflow: Run the local examples (E1), calculate effectiveness from the source lists (E2), aggregate recorded load times (E3), and evaluate the script records (E4). E4 covers Figure 6 and Tables IV–V. All commands below start at the archive root. Offline outputs go to `results/analysis/`; compare them with `reference_results/`.

To meet the NDSS 2027 AE one-day budget, E2–E4 extract and analyze records from the original collection instead of repeating full website collection.

Times are medians of three sequential runs after installation on an i7-14700, Ubuntu 24.04/WSL2 (28 logical CPUs, 16 GB RAM), excluding initial image construction and manual inspection.

4) Major Claims: **(C1) Source accuracy.** The CookieTrace source-level entries in Table II report pooled precision of 98.7% with 100% recall. E2 calculates these metrics directly from CookieTrace-recorded and ground-truth source lists.
(C2) Runtime overhead. Figure 4 reports a mean instrumented load time of 79.34 seconds and a 2.71× slowdown. E3 calculates the mean load times and their ratio from recorded timings.
(C3) Source characteristics. Figure 6 and Tables IV–V characterize script categories, normalized source scripts, and first-/third-party collaboration. E4 recomputes script-category aggregation and supports inspection of the table data, including 45 consent-script and 522 unnecessary-script entries.

5) Evaluation:

a) Experiment (E1): Browser execution: [Time] 14.7 seconds for both demos, sequentially. *[Preparation]* Complete installation and the environment check above. *[Execution]* Run the two bundled local examples:

```
bash ae/run.sh
bash ae/run.sh --demo --demo-case function
```

The no-argument command runs the direct-assignment demo. The HTML/JavaScript fixtures are served inside the container without external website access. *[Results]* Inspect the printed run directory under `ae/results/`. Require successful completion in `status.json`; inspect `sources.json`, `paths.json`, raw traces, and browser logs. `report.html` and `summary.txt` present the result. Confirm the requested cookie and expected source literals against `sources.json` and `paths.json` in `reference_results/e1/direct/` and `reference_results/e1/function/`. Both demos produce `ct_demo_cookie=ct_demo_value; path=/`, with four literals: `ct_demo_cookie`, `=`, `ct_demo_value`, and `; path=/`. Ignore timestamps, normalize loopback ports consistently, and compare sources without depending on array order. Optional live-site execution uses `--url URL` and `--cookie-name NAME`; `--timeout` sets its time limit.

b) Experiment (E2): Effectiveness (C1): [Time] 1.5 seconds. *[Preparation]* Use CookieTrace-recorded source lists

in `data/e2/selected_json/` and ground-truth lists in `data/e2/GT/`. Files are paired by filename; these two directories are the only E2 inputs. *[Execution]*

```
python3 analysis/evaluate.py e2
```

Source records are matched by script URL, code position, function bounds, type, and literal value. TP is the intersection of CookieTrace-recorded and GT sets; FP is recorded minus GT; FN is GT minus recorded. Pooled precision is $\sum TP/(\sum TP + \sum FP)$; recall is $\sum TP/(\sum TP + \sum FN)$. *[Results]* The command prints total TP, FP, FN, precision, and recall. Inspect `e2_source_comparison.json` for per-sample counts and TP/FP/FN records, and `e2_effectiveness.json` for totals and metrics under `results/analysis/`. Expect TP=10,005, FP=128, FN=0; precision rounds to 98.7% and recall is 100%.

c) Experiment (E3): Performance (C2): [Time] 0.2 seconds. *[Preparation]* Use `data/e3/combined_overhead_expanded.csv`, the supplied timing input. *[Execution]*

```
python3 analysis/evaluate.py e3
```

[Results] Inspect `results/analysis/e3_overhead.json`. Give all 9,783 rows equal weight. Mean `loadtime` is 79.34164123 seconds; mean `clean_loadtime` is 29.27735839 seconds. Divide these means to obtain 2.71×, rather than averaging per-row ratios. The script also checks positive finite values, duplicates, and stage totals.

d) Experiment (E4): Characterization (C3): [Time] 2.6 seconds for all E4 commands. *[Preparation]* Figure 6 inputs are in `data/e4/figure6/`: the frozen source-script list, CMP and WhoTracksMe lists, saved classifications, and saved category summary. *[Execution]* Re-run classification and aggregation:

```
( cd data/e4/figure6
  S=../../../analysis
  O=../../../results/analysis
  I=source_scripts_for_classification.json
  T=script_type_list
  node "$S/classify_scripts.mjs" \
    --in "$I" --cmp "$T/cmp-list.csv" \
    --track "$T/whotracksme.json" \
    --out "$O/reclassified_scripts.json"
)
R=results/analysis
F="$R/reclassified_scripts.json"
python3 analysis/evaluate.py figure6 \
  --classified-input "$F"
```

Alternatively, aggregate the saved classification directly:

```
python3 analysis/evaluate.py figure6
```

[Results] The aggregation must match the saved category summary. Outputs include category JSON/CSV and `e4_categories.svg`. There are 567 entries: 268 classified, 299 unknown, and 10 with multiple categories. Category memberships may overlap.

For Tables IV and V, print the counts with:

```
python3 analysis/table_iv.py
python3 analysis/table_v.py
```

The equivalent unified commands are `python3 analysis/evaluate.py table4` and `python3 analysis/evaluate.py table5`. Each command prints counts and comparison status, and writes JSON/CSV/TXT outputs under `results/analysis/`.

For Table IV, inspect `data/e4/reported_tables/script_summary.json`: `normalized`, `count`, and `examples` describe script identifiers, website counts, and example URLs. The example count is not the website count; `non-func` denotes unnecessary cookies. The paper's "other 41 scripts = 99" counts distinct websites after deduplicating membership across the remaining 41 scripts; their counts must not be added directly. For Table V, inspect `collaboration_summary.json` in the same directory. `details` groups cookie records by category and collaboration type; `stats` contains counts and percentages. Single- and multi-script percentages use their respective subtotals.

Verify website-level correspondence between Table V and E3 against the supplied ranking snapshot:

```
python3 analysis/evaluate.py mapping
```

Expect 6,995 mapped websites, 9,783 records, equal rank sets, and equal per-website counts. Outputs are `domain_rank_mapping.csv` and `domain_rank_checks.json`. Timing rows lack cookie identifiers, so this establishes website-level correspondence only.

As an alternative to separate offline calculations, run:

```
python3 analysis/evaluate.py all
```

This includes E2, E3, saved-classification aggregation, Tables IV–V, and mapping.