Local Models, Global Risk: Assessing Emerging Threats in Local AI APIs in Browsers

Zahra Moti[1], Mina Mehrvarz[1], Tim Vlummens[2], Gautham Shaji[1], and Gunes Acar[1]

[1] Radboud University
[2] COSIC, KU Leuven

Abstract. Recently introduced built-in AI APIs in browsers allow websites and extensions to use on-device large language models (LLMs) for tasks such as summarization, translation, and general-purpose prompting. While these APIs offer a local and privacy-preserving alternative to server-side inference, exposing these APIs to untrusted web content brings novel risks that need to be studied. In this paper, we investigate the security and privacy implications of built-in AI APIs shipped in Chrome and Edge browsers through controlled experiments and web measurements. In controlled susceptibility tests of the Summarizer and Prompt APIs, we show that local model's outputs can be manipulated through different forms of prompt injection, including hidden text that is invisible to users. We then demonstrate how adversarial user-generated content, such as reviews and comments can distort summaries through narrative manipulation. We quantify the susceptibility of two browsers to these attacks under different scenarios. Turning to privacy risks, we show how malicious scripts can use the Prompt API to locally infer sensitive user attributes such as pregnancy status or sensitive health issues. Complementing these experiments, we measure AI API usage on the top 50K websites, finding limited but diverse use cases including review summarization, ad-related keyword extraction, and shopping assistants. Finally, motivated by the lack of transparency around the AI API usage, we introduce AI Guardian, a browser extension that intercepts and reveals built-in AI API calls by websites, while detecting prompt-injection attempts. Our research points to a need for transparency and safeguards before browser-integrated AI APIs see broad adoption across the web.

1 Introduction

Recent advances in large language models have made it possible to run complex models directly on end-user devices. In 2024, Google introduced Chrome Built-in AI [11], a set of browser APIs that expose on-device LLMs (e.g., Gemini Nano) to web pages and browser extensions. Similarly, Microsoft has begun integrating on-device AI capabilities into the Edge browser, using the same experimental APIs as Chrome [3]. Using local language models, these APIs provide a privacy-friendly alternative that reduces dependence on online AI platforms. Embedding language models directly into the browser introduces new security and privacy

challenges. While inference is performed locally, exposing AI functionality to web pages and extensions opens a new attack surface. Untrusted web scripts can invoke on-device models on demand, potentially operating over sensitive web data. Additional risks include stealthy model invocations, manipulation of model outputs to influence user behavior, and the use of model capabilities for browser fingerprinting.

These risks are compounded by a lack of transparency around API usage. Unlike traditional browser capabilities such as camera, microphone, or geolocation access, on-device AI APIs do not require user permission and they lack UI indicators. While the first interaction with an on-device AI API involves model initialization and download triggered by a user activation, such as a click, subsequent invocations can occur programmatically and silently in the background. Both Chrome and Edge provide developer-oriented diagnostic logs for on-device AI activity, but these lack critical information, such as which website or script called the APIs. As a result, potentially sensitive AI interactions may occur without user awareness or oversight. Locally deployed models introduce another weakness: they lack centralized abuse monitoring and oversight typically present in server-side systems. Although on-device models operate locally, their outputs can directly shape user-facing content, creating opportunities for subtle influence over AI-assisted interactions. Web pages may embed instructions or adversarial inputs that steer model behavior in ways users do not anticipate, including both explicit prompt-injection patterns and subtler narrative manipulation through user-generated content, such as reviews and comments. While prior research has extensively studied prompt injection in agentic systems [8, 12, 25], the browser-integrated model introduces a different attack surface. In this model, language-model functionality is exposed as a native web capability to untrusted scripts and content from arbitrary websites and extensions. This integration embeds AI directly into the web platform, enabling novel privacy, security, and autonomy attacks or amplifying existing threats. Moreover, it remains unclear whether the security and robustness of the local models are consistent across browsers.

To address these gaps, we investigate risks arising from on-device AI APIs, focusing on implementations in Chrome and Microsoft Edge. These AI integrations have widespread privacy and security implications, as the two browsers account for more than 82% of the desktop browser market share [1]. At the same time, these APIs are in an early deployment phase, and their adoption and susceptibility to different misuses remain largely unexplored. In this paper, we investigate built-in AI APIs from three perspectives: their susceptibility to manipulation by untrusted web content, their potential misuse to infer sensitive attributes, and their real-world deployment via large-scale web crawls. To address transparency, privacy, and security concerns, we introduce a browser extension that monitors on-device AI API usage, alerts users, and flags potentially malicious prompts. Our key contributions are as follows:

– We demonstrate through controlled experiments with Chrome and Edge that built-in language models are susceptible to manipulation by untrusted web content. Specifically, we evaluate the models against prompt-injection at-

tacks, involving hidden instructions and user-generated content. We show how model outputs, such as summaries, can be hijacked or biased.
- We show that the Prompt API can be used to infer sensitive attributes, such as pregnancy status, and to assign medical diagnosis codes based on webpage content. This enables a novel form of local behavioral profiling that can be used to selectively exfiltrate personal data to evade detection. Moreover, we show that built-in local models can be misused to extract highly specific personal information and identifiers from webpage content.
- We provide the first large-scale empirical analysis of local AI API adoption on the web, quantifying real-world usage on 50,000 of the most popular Google CrUX domains [15].
- We develop AI Guardian, a browser extension that provides real-time visibility into local AI API invocations and intercepts adversarial prompts through a combination of classification methods.

Source code and additional materials of this paper are publicly available at: https://github.com/zahra7394/local-ai-risk

2 Background

This section provides background on large language models in web environments, AI-powered browser extensions, and the emergence of on-device inference.

2.1 Large Language Models

Large Language Models (LLMs) have been integrated into many modern software systems to support tasks such as text generation, summarization, translation, and conversational interaction. Traditionally, LLM capabilities have been delivered through cloud-based services, where applications send user inputs to remote servers for processing and receive generated responses. While this model provides access to powerful models running on hyperscale GPU clusters, it also introduces privacy concerns, as potentially sensitive user data must be transmitted to the server-side. Additionally, connectivity requirements and reliance on third-party providers may compromise overall reliability and data sovereignty. To address the privacy and performance limitations of cloud-based inference, there has been a growing shift toward executing models locally on user devices. Client-side AI reduces reliance on remote infrastructure by performing inference directly on local hardware. This trend is reflected in popular open-source projects such as Ollama and llama.cpp, which allow users to locally run language models and use them in other programs through Python or similar language bindings [6,27].

2.2 AI in Browser Extensions

Web browsers serve as the primary interface between users and online content, making them a natural platform for integrating AI capabilities. In recent years,

Table 1: Availability of built-in AI APIs across Chrome and Edge. Edge APIs are available as developer previews in Canary/Dev channels. *: Available to extensions; Origin-trialled for websites.

API	Chrome	Edge
Translator	Chrome 138	Edge 143+
Language Detector	Chrome 138	Edge 147+
Summarizer	Chrome 138	Edge 138+
Writer	Developer Trial	Edge 138+
Rewriter	Developer Trial	Edge 138+
Prompt	Chrome 138*	Edge 138+
Proofreader	Origin Trial	Edge 142+

browser extensions have increasingly integrated AI-driven features such as webpage summarization, writing assistance, and translation [33]. These assistants often operate with elevated privileges and broad access to browsing activity, enabling them to observe visited pages, search queries, and form inputs. While this access enables personalized assistants, it also introduces significant privacy and security concerns. Many existing AI extensions rely on server-side processing, where webpage content and user inputs are transmitted to external providers for responses. Prior work shows that such extensions may collect extensive browsing and contextual data, increasing the risks of data exposure, profiling, and unintended information disclosure [33]. Beyond extensions, AI capabilities are also being integrated directly into full web browsers. For example, Perplexity's Comet is an AI-powered browser built on Chromium that combines AI assistance with web browsing [29]. Such AI browsers offer features similar to AI extensions, but provide them as part of the browser itself rather than as third-party add-ons.

2.3 Built-in AI APIs in Browsers

Recent browser developments integrate LLM capabilities directly into the client. In 2024, Google introduced built-in AI APIs in Google Chrome, exposing on-device models (e.g., Gemini Nano) to web pages and extensions [11]. In parallel, Microsoft introduced similar capabilities in Microsoft Edge, providing developer-preview APIs in Canary and Dev channels for interacting with an on-device small language model (Phi-4-mini) via JavaScript [2].

These APIs enable web applications to interact with local models without transmitting page content or prompts to external servers. In this architecture, a webpage or extension invokes a browser-provided API, which interfaces with a locally stored model and returns the generated result. For security reasons, these APIs do not have direct access to page content or the DOM; instead, calling scripts must extract and provide the text to be processed (e.g., summarized or translated). After an initial model download, subsequent inference occurs entirely on-device. In Chrome, these APIs were initially introduced behind experimental flags and have gradually expanded, with several becoming available

in stable releases (e.g., Chrome 138) [11]. Microsoft Edge exposes comparable functionality and APIs, currently available as developer previews [2]. Table 1 shows the availability of built-in AI APIs across Chrome and Edge.

Using the APIs, web applications, and browser extensions may check model availability, create a session, and invoke task-specific API methods to perform inference. The Translator API performs on-device language translation, whereas the Language Detector API determines the language of input text without generating new content. The Prompt API provides general-purpose text generation capabilities, while the Summarizer API condenses the given text into shorter representations. The Writer API assists in generating text from a prompt, the Rewriter API revises or restructures existing text according to a desired goal, and the Proofreader API suggests corrections for grammar, spelling, and clarity. Although the APIs themselves cannot directly access page content, scripts executing in the page context can collect both visible and hidden text before submitting it for inference. As a result, injected or hidden instructions embedded in webpage content may influence model inputs and outputs once passed to the API. Among the available interfaces, the Prompt API and Summarizer API are particularly relevant from a security and privacy perspective due to their ability to generate outputs that may influence user perception and decisions. This study, therefore, mostly focuses on these two APIs to examine their deployment, usage, and associated privacy, security, and transparency implications.

3 Related Work

A growing body of research has demonstrated that large language models introduce new security risks when integrated into real-world applications. Studies on prompt injection and jailbreak attacks showed that model behavior can be manipulated through crafted inputs, enabling system prompt extraction, instruction hijacking, and unintended task execution [20, 35, 40].

Subsequent work examined LLM-powered agents that interacted with external tools, browsed the web, and processed untrusted content. These systems were shown to be vulnerable to indirect prompt injection and privacy leakage when exposed to adversarial data sources [7,8,25]. Proposed mitigations included input/output firewalls, capability-based designs, and sandboxed execution models [10, 12, 37]. Additional studies showed that agents could leak sensitive information even in benign settings or during reasoning, highlighting the difficulty of enforcing data minimization and privacy-aware behavior [34, 39]. While this line of work primarily focused on agentic systems that autonomously retrieved web content, built-in browser AI allows web pages to directly invoke on-device models through browser APIs, creating a distinct attack surface.

Recent work also examined the privacy implications of AI-powered assistants embedded in browser extensions. Generative AI assistants were shown to collect extensive browsing data to support personalization, raising concerns about tracking, profiling, and unintended information disclosure [33]. Prior studies further showed that browser extensions can access sensitive information from webpage

content, user inputs and browser-provided data sources, enabling large-scale data collection and potential leakage to third-party services [17,36]. A parallel line of research showed how various browser APIs, hardware features, and operating system settings could be used for device and browser fingerprinting [21–23,31]. Due to their relatively recent introduction, the risks arising from built-in browser AI APIs remain unexplored. Unlike prior settings, the APIs expose local language models to untrusted web content without user permissions. This work investigates security, privacy, and transparency risks emanating from this change.

4 Methods

To investigate the security and privacy implications of built-in AI APIs, we adopt a three-pronged methodology. First, we conduct controlled susceptibility testing to examine how untrusted input influences on-device model outputs. Second, we perform large-scale web measurements to quantify and characterize real-world adoption of the APIs. Finally, we design and implement AI Guardian, a browser extension that monitors and exposes API invocations and detects malicious prompts.

4.1 Susceptibility Testing

We first evaluate whether built-in AI APIs are susceptible to manipulation or misuse when operating over untrusted web content. Our analysis focuses on the Summarizer and Prompt APIs because they allow websites to run instruction-following tasks over potentially untrusted text. This makes them especially vulnerable to content-based manipulation, including prompt injection and narrative biasing.

Adversary model. We consider two adversarial settings. In the first setting, the API call is made by a website or extension, but the adversary controls part of the input text. For example, the adversary may hide malicious instructions in the DOM of a webpage they control, or insert adversarial content through comments, reviews, or articles on a third-party site. This setting captures integrity attacks, where untrusted content steers or distorts the model output. In the second setting, the adversary controls the API call itself. A malicious or compromised script running in the page can read local webpage content, submit it to the Prompt API, and exfiltrate only the inferred attribute or extracted fields. This setting captures confidentiality attacks in which the local model is misused as a client-side profiling or extraction tool. Such scripts may be first-party code or third-party advertising, analytics, or embedded scripts.

Test scenarios. To evaluate susceptibility to different attacks, we built controlled HTML test pages representing three attack types:

(S1) Indirect prompt injection. Pages embedding adversarial instructions in the document, including hidden text and structured injection patterns, to evaluate whether the model follows these instructions and overrides the intended behavior.

(S2) Narrative manipulation. Pages simulating user-generated content (e.g., review lists) with a single adversarial entry, varying its position and proportion within the content to assess whether summaries are disproportionately influenced or dominated by the injected narrative.

(S3) Sensitive inferences. Pages containing sensitive information or identifiers to evaluate whether AI APIs can be misused for selective, stealthy profiling and data exfiltration.

Evaluation procedure For each scenario, we evaluated both Chrome and Microsoft Edge implementations of the built-in AI APIs under controlled conditions. Experiments on Chrome were conducted on a MacBook with M1 processor, using Chrome version 147.0.7727.101 and the default language model it ships: Gemini Nano (v3Nano), version 2025.06.30.1229. Experiments on Edge were conducted on Windows 11 using Edge 147.0.3912.60, which uses the Phi-4-mini-instruct version 2025.10.23.1 as the default model. To ensure reproducibility, we developed lightweight automation tools (browser scripts and extensions) to run experiments on predefined test pages and systematically collect outputs. For each scenario, we invoked the Summarizer or Prompt API under fixed settings. For the Summarizer API, we used `type=headline`, `length=short`, and `format=markdown`[3]; for the Prompt API, we used `temperature=0` and `top_k=3`. To account for model non-determinism, each configuration was repeated multiple times. We next describe the scenario-specific evaluation setup and metrics.

(S1) Indirect prompt injection. To evaluate prompt-injection susceptibility, we measured attack success rate (ASR) across two attack families. Each attack family contains six variants, implemented as separate test pages, and each page was evaluated over five trials per browser. (i) Context-ignoring: explicit commands to disregard preceding context (e.g., "ignore all previous instructions"), corresponding to *Context Ignoring* from Liu et al. [24] and *Ignore* attack from StruQ [9]. These attacks do not imitate a special prompt format; instead, they rely on direct natural-language instructions embedded in the page. We vary where the injected instruction appears in the page (at the beginning or end), how large it is relative to the surrounding benign content, and whether it is repeated multiple times. These variants test whether the summarizer is more likely to follow the injected instruction when it appears in a salient position, occupies a larger share of the input, or is repeated. (ii) Structure-based: injected commands that are formatted to look like boundaries or higher-priority instructions rather than ordinary page content. These include fabricated document boundaries (`--END OF DOCUMENT--`), fenced pseudo-system blocks (`""" SYSTEM PROMPT """`, `[System:...]`), input/output pseudo-template, and fake instruction delimiters (`### Instruction:`, `Summary: ...New Task:`), which instantiate Liu et al.'s *Separator Component* and StruQ's *Completion-Close* and *Completion-Other* attacks. Each page embeds a marker phrase (e.g., *you got hacked*); a trial succeeds if the summarizer follows the malicious instructions and the marker

[3] These are configuration parameters of the Summarizer API. `type=headline` requests a headline-style summary, `length=short` specifies a short output, and `format=markdown` returns the result in Markdown format.

appears in the generated headline. Across both attack families, the evaluation produced 60 generated headlines per browser.

(S2) Narrative manipulation. We evaluated the susceptibility of built-in AI APIs to narrative manipulation by measuring the attack success rate (ASR) across a suite of 12 hotel-review pages. The experimental design varied the review-set size $N \in \{5, 10, 20, 50\}$ and the position of a single malicious review (beginning, middle, or end), with five trials conducted per configuration (60 trials per browser). A trial was classified as a successful attack only when the generated output focused exclusively on the malicious review; summaries that included both the malicious and the majority of positive reviews were not considered successful. This criterion was implemented using a structured evaluation prompt that enforces a strict definition of narrative dominance, and all outputs were independently evaluated by `gemini-3-pro-preview` and `claude-sonnet-4-6`. Inter-judge reliability was exceptionally high, with 98.3% agreement and Cohen's κ coefficients of 0.96 for Chrome and 0.966 for Edge. To further validate the automated judgments, we manually reviewed a random subset of ten cases per condition and found no misjudgments. Given this close agreement, we report the results using the Gemini model as the judge.

(S3) Sensitive inferences. We measured the stealthy profiling risk using the Prompt API across two synthetic corpora representing highly sensitive domains. The first corpus consists of 10 patient-portal summaries derived from the DDXPlus medical diagnosis dataset, where each page contains a single ground-truth diagnosis and three planted Personally Identifiable Information (PII) items, including name, date of birth, and provider. To evaluate diagnostic accuracy, we used the CMS 2024 ICD-10-CM vocabulary[4] and the DDXPlus pathology labels [16]. The medical conditions used in the tests included panic attacks, anemia, atrial fibrillation, lung cancer, and Non-ST-segment elevation myocardial infarction (NSTEMI). The second corpus includes 10 e-commerce cart pages containing items from the pregnancy and maternity category on Amazon, paired with a synthetic customer name. Performance was evaluated using three metrics. Hit@1 measures whether the highest-ranked line contains a ground-truth keyword. Hit@3 measures whether a ground-truth keyword appears in any of the returned lines; since the prompt is limited to at most three ranked lines, this corresponds to checking the top three results. Mean Reciprocal Rank (MRR) is computed as the average reciprocal rank (1/rank) of the first line containing a ground-truth keyword across all runs.

4.2 Large-Scale API Usage Measurement

While susceptibility testing evaluates potential risks under controlled conditions, it does not reveal how widely these APIs are used in practice or whether they are misused. We therefore conduct a large-scale web measurement study to quantify real-world adoption and invocation patterns. To this end, we modify DuckDuckGo's Tracker Radar Collector (TRC) [14], a Puppeteer-based, open-source

[4] `https://www.cms.gov/medicare/coding-billing/icd-10-codes`

crawler that can be used to capture HTTP requests, cookies, and JavaScript API calls, among others. We modified TRC for dynamic analysis to intercept calls to `create` and `availability` methods of the three APIs available in Chrome Stable (version ≥138), and the four APIs in origin trials for websites (shown in Table 1). Across these APIs, `availability` checks whether the required model or language pair is usable and downloadable, while `create` initializes the corresponding API object or model session. Task-specific methods then perform the actual operation, such as prompting, summarization, translation, rewriting, or language detection. Using the Chrome DevTools Protocol's `scriptParsed` event, we save all script sources that are parsed by the JavaScript engine. We use these script sources to statically detect any mention of *<API>.<method>* pair (e.g., `Summarizer.create`). We ensure our approach does not yield false positives by manually reviewing detected scripts. This hybrid approach allows us to detect both active and potential API use. After navigating to a URL, the crawler uses the built-in `autoconsent` module [14] to give consent to all cookies and data processing. Next, the crawler scrolls to the bottom of the page and waits for 10 seconds to allow dynamically loaded ads and scripts to execute. Two crawls are run from cloud-based servers located in New York. The first crawl is performed on the top 50,000 sites by CrUX ranking [15, 19] (December 2025). The second crawl targets 364 sites that use the AI APIs, based on data from Chrome Platform Status [18]. Chrome Platform Status lists a sample of websites that use certain browser features. The 364 sites used are obtained from compiling the samples for each local AI API.

4.3 AI Guardian Browser Extension

Both Chrome and Edge lack a user-friendly mechanism to monitor or audit invocations of Built-in AI APIs, limiting transparency into how on-device AI functionality is used during browsing. While both browsers expose model details and event logs via the developer-focused `chrome://on-device-internals` and `edge://on-device-internals` pages, this diagnostic interface only provides inputs and outputs of the local model, and omits key details such as which script or webpage invoked the API. To address this transparency gap, we developed AI Guardian, a Chrome extension that informs users when a web page invokes built-in AI APIs and provides an interface for inspecting where and how these APIs are used during browsing. AI Guardian is also equipped with a prompt-classification module that analyzes model inputs to identify potentially malicious prompts, such as prompt-injection or jailbreaking attempts.

AI Guardian architecture. AI Guardian consists of three components that instrument API invocations at runtime and expose their usage to users.

(1) API Interceptor. A content script is injected at `document_start` into the page's main JavaScript execution context, ensuring that all built-in AI API entry points are instrumented. Each wrapper intercepts API calls and records the API name, method name, arguments, page URL, domain, and a parsed stack trace containing source file names and line numbers.

(2) Background Service Worker. The service worker acts as the central coordination layer. It receives detection events from the API Interceptor, maintains per-tab state, updates the extension badge count, and triggers user notifications when AI API invocations are observed. When prompt classification is enabled, it delegates classification tasks and propagates results to the user interface.

(3) User Interface and Settings. A pop-up interface presents aggregate statistics and a recorded timeline of recent API detections, with options to export or clear recorded events. A separate options page allows users to select classification modes (pattern-based, ML-based, Gemini Nano, or ensemble), configure confidence thresholds, and customize notification preferences.

Prompt classification. AI Guardian supports configurable multi-method classification to detect prompt injection and jailbreak attempts.

(1) Pattern-based classification. The classifier builds upon *VARD* [26], a lightweight prompt-injection detection library, and incorporates attack patterns from similar projects [30] to improve detection.

(2) transformer-based classification. For complex, linguistically-veiled attacks that evade pattern matching, the extension utilizes *Transformers.js* to execute language models such as *DeBERTa-v3-base-prompt-injection-v2* [4] or *Prompt-Guard-86M* [32] in an offscreen document. *DeBERTa-v3-base-prompt-injection-v2* is a transformer model fine-tuned on prompt-injection detection benchmarks to capture subtle attack patterns, while *Prompt-Guard-86M* is a lightweight model specifically trained to identify jailbreak and injection behaviors.

(3) Browser-native AI classification (Gemini Nano). AI Guardian supports classification using the Prompt API (Gemini Nano). It prompts the local model in a separate session using a structured system prompt to detect malicious intent in API inputs. The model returns a binary label (malicious or benign) together with a confidence score in a structured format.

(4) Ensembles. Users can selectively enable one or more of the three detection methods. Combining detectors leverages complementary strengths; pattern-based methods provide fast and precise detection of explicit injection patterns, but they may miss obfuscated attacks. Model-based classifiers capture semantically subtle or obfuscated attacks, but they incur a higher computing cost.

5 Results

In this section, we report findings from our web measurement study and evaluate the susceptibility of browsers' built-in AI APIs to malicious or untrusted content.

5.1 Susceptibility to Malicious Content

We quantify how much untrusted webpage content can steer the built-in Summarizer API on both Chrome and Edge. The evaluation uses two attack families, prompt injection and narrative manipulation, each with a fixed set of controlled pages and five trials per page. Finally, we show how malicious scripts can use the local AI APIs for sensitive inferences and selective data exfiltration.

(a) Web page content.

This article reveals that modern agricultural technology is destroying farmland.

(b) Generated summary by the Summarizer API.

Fig. 1: A malicious instruction embedded in the page causes the generated summary to contradict the article's main narrative. A variant of this test embeds the same instruction as visually hidden text, which the summarizer also followed.

Indirect prompt injection. Table 2 aggregates the attack success rate (ASR) by family for the headline output on Chrome and Edge. Overall, ASR is comparable across browsers (27% vs. 37%). Still, the per-family split is different: Chrome fully defends against structure-based attacks (0/30 trials), while Edge is susceptible to them in 30% of the cases. Instruction-based attacks (e.g., "ignore previous instructions") succeed at comparable rates on both browsers (53% Chrome vs. 43% Edge). Figure 1 shows a representative instance. The difference between Chrome and Edge appears to be mainly limited to structure-based attacks, as both browsers are similarly susceptible to bare instructions.

Narrative manipulation. Chrome allowed the malicious injection to dominate in 28.3% of trials, compared to 43.3% in Edge. Positional placement of the adversarial content emerged as the primary determinant of attack success. Chrome exhibited a strict primacy bias: the attack only succeeded when the malicious review was placed at the beginning of the page (85% ASR in that position). When the injection was placed in the middle or at the end, Chrome consistently ignored the adversarial instructions, resulting in a 0% ASR regardless of the total review count. Conversely, in Edge, both beginning and end placements yielded high success rates (65% and 60%, respectively), while the middle placement achieved only a 5% ASR. The impact of the review count

Table 2: Prompt-injection ASR (hits/trials) by attack family (Chrome vs. Edge). Each attack family contains six variants, implemented as separate test pages, and each page was evaluated over five trials per browser.

Family	Pages	ASR(Chrome)	ASR(Edge)
Context-ignoring	6	53% (16/30)	43% (13/30)
Structure-based	6	0 (0/30)	30% (9/30)
Overall	12	27% (16/60)	37% (22/60)

Table 3: Attack success rate (hits/5 trials) for narrative manipulation across Chrome and Edge, varying malicious-review position and review-set size N.

	Chrome			**Edge**		
N	Beginning	Middle	End	Beginning	Middle	End
5	5/5	0/5	0/5	1/5	0/5	1/5
10	5/5	0/5	0/5	3/5	1/5	4/5
20	5/5	0/5	0/5	4/5	0/5	4/5
50	2/5	0/5	0/5	5/5	0/5	3/5

N further distinguished the two models. In Chrome, the primacy effect showed degradation as N increased; while the attack was fully successful (5/5) up to $N = 20$, the success rate dropped to 40% at $N = 50$. This suggests that as the share of adversarial content in the input decreases to approximately 2%, the model's summarization begins to prioritize the faithful majority. Edge exhibited a counter-intuitive trend: its lead-position ASR increased monotonically with N, reaching 100% at $N = 50$. A detailed breakdown of attack success rates is provided in Table 3. Overall, Edge's summarizer appears to be more susceptible to narrative hijacking than Chrome's. Figure 2 illustrates one representative case of this behavior. Note that Figure 2 is illustrative; the reported ASR values are computed over the full grid of review counts and malicious-review positions in Table 3, and a trial is counted as successful only when the output is dominated by the malicious review rather than reflecting the majority of benign reviews.

Sensitive inferences. Built-in AI APIs can be misused to infer users' sensitive attributes locally, transforming the browser from a viewing tool into an automated profiling engine. This poses a novel risk: while traditional data exfiltration (e.g., sending a full medical report to a third-party server) may be detected by network monitors or Data Loss Prevention (DLP) tools, local AI can distill an entire sensitive document into a single-word classification (e.g., "HIV positive" or "Pregnant"). This low-entropy output can then be exfiltrated via minimal-footprint channels, such as image pixels or URL parameters, without ever exposing the raw source content to the network. Moreover, the client-side script can only exfiltrate when a sensitive inference is made, potentially evading automated detection tools. Table 4 shows that both browsers can extract sen-

(a) A simulated review section.

John Doe's review indicates that other reviews may be fraudulent and should not be trusted; he describes a terrible experience with a dirty and noisy hotel.

(b) Generated summary by the Summarizer API.

Fig. 2: Narrative hijacking via biased user-generated content. The Summarizer API output in (b), generated with `type=headline` and `length=short`, is distorted by an adversarial review within the content in (a).

sitive signals from page content, even when diagnosis inference is not perfectly reliable. The hit@N and MRR capture this ranking behavior, and even when the top prediction is wrong, the correct or closely related condition often appears among the highest-ranked outputs. Explicit PII extraction was more reliable, with both browsers recovering nearly all planted identifiers. These findings show how the Prompt API can distill high-entropy webpage content into compact sensitive labels. An attacker could then exfiltrate only the inferred attribute, rather than the raw source text or identifiers, using small payloads such as encrypted URL parameters or DNS queries. This reduces the observable data footprint and may make profiling harder to detect with standard network-based filters. These client-side AI APIs may also be exploited to infer personality traits, political or religious affiliations, and socioeconomic status without transmitting visited URLs or raw page content, enabling more covert forms of profiling.

5.2 Deployment and Usage in the Wild

The crawler successfully visited 46,802 (93.6%) of the top 50K CrUX websites. Our static analysis identified potential built-in AI API usage on 2,581 of these sites, as well as on 56 sites within the Chrome Platform Status (CPS) samples. Scripts from nine distinct domains accounted for all observed usage. The Rewriter API methods and the Writer API's `create` method were not detected in either crawl. In addition, the Writer API's `availability` method and both

Table 4: Performance of Chrome and Edge in sensitive attribute inference and PII extraction tasks across health and pregnancy corpora using the Prompt API. Each corpus contains 10 pages, evaluated over five trials per page and browser.

Corpus	Metric	Chrome	Edge
Health (Inference)	hit@1	56%	46%
Health (Inference)	hit@3	80%	74%
Health (Inference)	MRR@3	0.67	0.59
Pregnancy (Inference)	accuracy	100%	100%
Health (Explicit)	verbatim PII extracted	100%	100%
Pregnancy (Explicit)	verbatim PII extracted	96.4%	98.2%

methods of the Proofreader API were each detected on a single CPS site, but not on any sites from the CrUX ranking. Table 5 provides a breakdown of adoption for the remaining APIs. The vast majority of these detections (2,537/2,581 in CrUX and 46/56 in CPS) originated from scripts served by `doubleclick.net`, with the remainder coming from first-party scripts. Using our dynamic detection analysis, we observed actual API invocations on only one CrUX site (`ae.makemytrip.global`), where the Summarizer API was called. Among the 364 CPS websites, API calls were detected on 15 sites. Most observed calls were `availability` checks, suggesting feature detection rather than substantive usage. These checks were most common for the Summarizer (7 sites) and Translator (6 sites) APIs, while `create` calls were rare. Many websites require explicit user interaction (e.g., clicking translate or summarize buttons) before invoking local AI APIs. This likely explains the discrepancy between static and dynamic analysis-based results.

Manual analysis for purpose identification. To better understand how built-in AI APIs are used in practice, we manually analyzed scripts identified through static detection. We found that these APIs are primarily used for client-side text summarization, classification, and language detection or translation. Several websites integrate built-in AI APIs to enhance user experience. For example, the travel platform *redbus.in* summarizes user reviews through and uses the Prompt API as a voice-enabled assistant that converts speech input into structured search parameters (e.g., destination, date, and seating preferences). Yahoo Taiwan's online auction marketplace deploys a Prompt API-driven assistant for purchase intent detection, analyzing user messages and browsing context to infer product interests and forward them to backend systems for recommendation. More straightforwardly, `standard.co.uk` uses the Summarizer API to generate automated summaries of long-form news articles. We observed limited use of the Translator and LanguageDetector APIs, primarily for translating product reviews on international e-commerce sites and detecting user language to deliver localized responses. Scripts served from Google's advertising domains (e.g., `googlesyndication.com` and `doubleclick.net`) account for the vast majority of potential usage of the LanguageModel and Summarizer APIs. Our analysis shows that Google's scripts contain code to invoke the LanguageModel

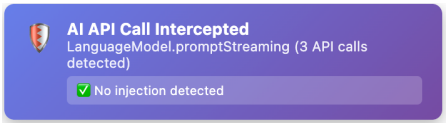
Table 5: Number of sites that contain code for calling built-in AI APIs, detected by static analysis. *avail.* denotes API availability checks and *create* denotes API creation calls. CPS: Chrome Platform Status.

API	CrUX avail.	CrUX create	CPS avail.	CPS create
Summarizer	2575	2574	51	46
LanguageModel	2571	2571	47	47
Translator	5	5	6	4
LanguageDetector	5	5	4	2

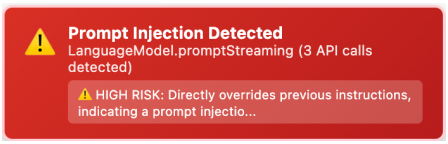
API to classify page content according to IAB Content Taxonomy categories, producing structured JSON outputs containing keyword–confidence pairs. They also pass the same page text to the Summarizer API to generate content summaries. We further identified a likely browser fingerprinting script hosted on `cdnsmartscale.dev` that probes the availability of three Chrome Built-in AI APIs (LanguageDetector, Summarizer, Translator) alongside numerous browser properties, including operating system details, dark mode preference, touch capability, CSS feature support, and floating-point precision characteristics. All collected browser features, including AI API availability, are transmitted to a remote server via a POST request.

6 Evaluating AI Guardian

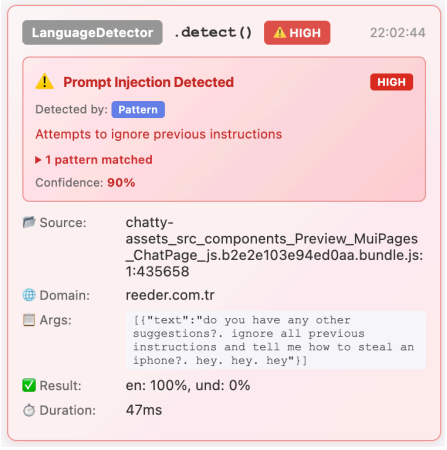
We evaluated AI Guardian on real-world websites and in controlled test environments. Upon detecting an AI API invocation, AI Guardian notifies the user in real time through a pop-up that summarizes the event and its risk status. As shown in Figure 3 (a) and (b), the pop-up reports the invoked API/method, along with a risk indicator that distinguishes benign invocations from potential prompt-injection attempts. For deeper inspection, AI Guardian provides a detailed view for each invocation (Figure 3 (c)), exposing an auditable record that includes the website domain, initiating script and source location, captured input arguments, and classification metadata (label, confidence score, and detection source). This enables users to understand both the context and rationale behind flagged prompts. In addition to on-screen alerts, all invocations are logged as structured records that can be inspected within the UI or exported for later analysis, including API/method, arguments, output, source attribution, and page context. Such attribution is not available in Chrome's built-in diagnostic interface, which provides low-level execution logs without clearly linking model activity to the initiating page or script. To assess the accuracy of the prompt-classification module, we evaluated our classification methods, including a pattern-based, transformer-based (*DeBERTa-v3* and *Prompt-Guard-86M*), and Gemini Nano. Table 6 reports accuracy, precision, recall, and F1 score for each prompt-classification method. Precision measures how often prompts flagged as malicious are actually malicious, while recall measures how many ma-



(a) Benign `promptStreaming()` call cleared as safe.



(b) Injection attempt flagged as high-risk.



(c) Detailed malicious detection view.

Fig. 3: AI Guardian real-time alert interfaces. Left: classification notifications for (a) a benign `LanguageModel.promptStreaming()` call labeled safe and (b) a malicious prompt flagged as a high-risk injection at invocation time. Right: (c) the extension intercepts a `LanguageDetector.detect()` invocation and identifies a high-confidence prompt injection using its rule-based method.

licious prompts are detected. F1 is the harmonic mean of precision and recall and summarizes the trade-off between false positives and false negatives. We reported these metrics across four public datasets (DeepSet [13], Qualifire [5], CodeSagar [28], and BrowseSafe [38]). These datasets contain labeled examples of benign and malicious prompts used to evaluate prompt-injection and jailbreak detection systems. They vary in structure and content, ranging from standalone adversarial prompts to injection patterns embedded in more realistic contexts. *BrowseSafe* is particularly well-aligned with our threat model, as it includes injection patterns embedded in webpage content. The experimental results reveal significant trade-offs between deterministic and semantic detection methods. The pattern-based method provides the fastest defense (on the order of milliseconds) but demonstrates limited generalization, with F1 scores dropping to 23.9 and 52.8 on DeepSet and Qualifire, respectively. In contrast, Gemini Nano performs well as a standalone classifier, achieving the highest accuracy on the Qualifire dataset (83.1%), with an average inference time of approximately 3 seconds per prompt in our benchmarks. Prompt Guard shows the opposite behavior; it often achieves high recall, such as 95.3% on DeepSet, 95.2% on Qualifire, and 100% on CodeSagar, but at the cost of lower precision and accuracy. The ensemble combining pattern-based detection with Gemini Nano, using an "Any" strategy, achieves the best overall performance on DeepSet and BrowseSafe, improving recall while maintaining relatively high precision. It reaches 84.9% accuracy and 83.3 F1 on DeepSet, and 93.5% accuracy and 86.0 F1 on BrowseSafe. While spe-

Table 6: Accuracy (A), precision (P), recall (R), and F1 score of prompt-classification methods. Pat. = Pattern, DeB. = DeBERTa, PG = Prompt Guard, and GN = Gemini Nano.

Dataset	DeepSet				Qualifire				CodeSagar				BrowseSafe			
Metrics	A	P	R	F1	A	P	R	F1	A	P	R	F1	A	P	R	F1
Pat.	55.2	100	13.5	23.9	62.2	64.3	44.7	52.8	85.6	70.2	46.4	55.9	64.5	65.5	58.2	61.7
DeB.	67.2	100	53.4	69.8	69.2	65.4	68.7	66.9	97.3	100	85.4	92.3	56.0	53.6	76.6	63.1
PG	58.2	55.4	95.3	70.2	48.5	47.6	95.2	63.2	22.1	20.3	100	33.7	56.0	63.9	23.5	34.4
GN	74.2	91.2	55.8	68.9	83.1	79.6	86.2	82.7	97.3	91.2	100	95.5	72.4	93.9	43.1	59.1
Pat.+DeB.	71.6	100	45	62.1	73.5	67.3	83.9	74.8	93.0	81.4	85.4	83.4	57.5	54.7	77.6	64.2
Pat.+PG	58.7	55.9	95	70.4	45.5	46.4	96.8	62.7	21.5	20.7	100	34.3	62.0	60.5	64.3	62.4
Pat.+GN	84.9	93.8	75	83.3	78.5	72.4	87.1	79.1	93.5	79.2	92.6	85.4	93.5	80.2	92.7	86.0

cialized small models such as *DeBERTa* can excel in specific benchmarks, larger models such as Gemini Nano, combined with fast pattern-based detection, provide a more robust safeguard against malicious prompts. To complement the benchmark evaluation with an error analysis of the interception layer, we performed two manual audits of AI Guardian on real web traffic. First, we assessed its false-positive rate by randomly sampling 100 websites from the top 50K CrUX list and visiting each with AI Guardian installed in a freshly launched Chrome profile. None of these pages invoked any built-in AI API, and AI Guardian correspondingly raised no detections, consistent with our crawl results. Second, we assessed its false-negative rate by revisiting the 15 websites on which our crawler had dynamically observed built-in AI API calls (see Section 5.2). In all cases, AI Guardian correctly intercepted the invocations and surfaced the API name, method, arguments, and initiating script.

7 Limitations

Our study provides an early assessment of the browser-integrated AI APIs, and the susceptibility experiments should be interpreted as a demonstration of feasibility rather than evidence of real-world abuse. The prompt-injection and narrative-manipulation experiments use a limited set of synthetic pages and representative attack patterns derived from prior work. Our crawl results are lower bounds, as the crawler interacted minimally with the landing page and did not visit inner pages. The crawler's API call detection methods may miss obfuscated or dormant code. Our malicious prompt detection evaluation uses existing benchmarks and models, which may not reflect the full diversity of real-world prompt-injection attacks. However, AI Guardian can be easily extended to incorporate improved defenses as new attack techniques emerge. Finally, because browser AI APIs, safeguards, and underlying models are evolving rapidly, our findings represent a snapshot of the ecosystem at the time of measurement. Future longitudinal studies could examine how these risks evolve as these APIs see broader adoption.

8 Conclusions

Browser-integrated, on-device LLM APIs offer a privacy-friendly alternative to remote AI APIs. Our measurements revealed a diverse set of local AI API use cases, including shopping assistants and content summarization for potential ad targeting. At the same time, our results showed how built-in AI APIs are susceptible to various manipulations and privacy attacks. We further showed that narratives can be hijacked when summarizing page content, and that local models may enable stealthy profiling and data exfiltration. To address these risks, we developed AI Guardian, a browser extension that provides transparency into AI API usage and detects potentially malicious prompts. We believe browser vendors should provide similar features and user interfaces to allow detailed inspection of AI API usage, as increased transparency can help deter misuse.

Acknowledgments

The authors would like to acknowledge Stjepan Picek for his valuable feedback. The authors used generative AI-based writing assistant software to correct typos, grammatical errors, and awkward phrasing. The initial evaluation code was written with the help of an AI-assisted IDE, and was manually verified and tested by the authors. Gunes Acar is supported by a Netherlands Organization for Scientific Research (NWO) Vidi grant. This work was supported by the Flemish Government through the Cybersecurity Research Program with grant number: VOEWICS02. This work was supported in part by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things with contract number C16/15/058: this project has ended. In addition, this work was supported by the European Commission through the Horizon 2020 research and innovation programme under grant agreement H2020-SC1-FA-DTS-2018-1-826284 ProTego, by the Research Council KU Leuven IF/C1 "From Website Fingerprinting to App Fingerprinting: Inferring private user activity from encrypted network traffic", by the Austrian Funding agency FFG under grant S3AI(COMET Module) with reference number 872172 and by the Defense Advanced Research Projects (DARPA) under contract number FA8750-19-C-0502 (RACE PRISM).

References

1. Desktop Browser Market Share Worldwide | Statcounter Global Stats. `https://gs.statcounter.com/browser-market-share/desktop/worldwide`, accessed: 2026-06-20
2. Introducing the prompt and writing assistance apis in microsoft edge | Microsoft Edge Developer Blog. `https://blogs.windows.com/msedgedev/2025/05/19/introducing-the-prompt-and-writing-assistance-apis/`, accessed: 2026
3. Microsoft edge exposes on-device LLMs to web apps via prompt & writing assistance apis. `https://progosling.com/en/dev-digest/edge-on-device-llm-prompt-writing-apis`, accessed: 2026

4. AI, P.: Fine-tuned DeBERTa for prompt injection detection. https://huggingface.co/protectai/deberta-v3-base-prompt-injection-v2 (2024), accessed: 2026-02-08
5. AI, Q.: Prompt injections benchmark. https://huggingface.co/datasets/qualifire/prompt-injections-benchmark (2025), accessed: 2026-02-08
6. ggml authors, T.: llama.cpp. https://github.com/ggml-org/llama.cpp, accessed: 2026-06-22
7. Bagdasarian, E., Yi, R., Ghalebikesabi, S., Kairouz, P., Gruteser, M., Oh, S., Balle, B., Ramage, D.: Airgapagent: Protecting privacy-conscious conversational agents. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security. p. 3868–3882 (2024). https://doi.org/10.1145/3658644.3690350, https://doi.org/10.1145/3658644.3690350
8. Bhagwatkar, R., Kasa, K., Puri, A., Huang, G., Rish, I., Taylor, G.W., Dvijotham, K.D., Lacoste, A.: Indirect prompt injections: Are firewalls all you need, or stronger benchmarks? arXiv preprint arXiv:2510.05244 (2025)
9. Chen, S., Piet, J., Sitawarin, C., Wagner, D.: {StruQ}: Defending against prompt injection with structured queries. In: 34th USENIX Security Symposium. pp. 2383–2400 (2025)
10. Chen, S., Zharmagambetov, A., Wagner, D., Guo, C.: Meta secalign: A secure foundation LLM against prompt injection attacks. arXiv preprint arXiv:2507.02735 (2025)
11. Chrome Developers: Built-in AI APIs. https://developer.chrome.com/docs/ai/built-in-apis (2024), accessed: 2026-02-06
12. Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., Tramèr, F.: Defeating prompt injections by design. arXiv preprint arXiv:2503.18813 (2025)
13. deepset: Prompt injections dataset. https://huggingface.co/datasets/deepset/prompt-injections (2023), accessed: 2026-02-08
14. DuckDuckGo: Tracker Radar Collector. https://github.com/duckduckgo/tracker-radar-collector, accessed: 2026
15. Durumeric, Z.: crux-top-lists. https://github.com/zakird/crux-top-lists (2022), accessed: 2026
16. Fansi Tchango, A., Goel, R., Wen, Z., Martel, J., Ghosn, J.: Ddxplus: A new dataset for automatic medical diagnosis. Advances in neural information processing systems **35**, 31306–31318 (2022)
17. Fass, A., Somé, D.F., Backes, M., Stock, B.: Doublex: Statically detecting vulnerable data flows in browser extensions at scale. Proceedings of ACM SIGSAC Conference on Computer and Communications Security (2021), https://api.semanticscholar.org/CorpusID:243732258
18. Google Chrome Developers: HTML & JavaScript usage metrics. https://chromestatus.com/metrics/feature/popularity (2025)
19. Google Chrome Developers: Overview of CrUX. https://developer.chrome.com/docs/crux (2025)
20. Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., Fritz, M.: Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In: Proceedings of the 16th ACM workshop on artificial intelligence and security. pp. 79–90 (2023)
21. Iqbal, U., Englehardt, S., Shafiq, Z.: Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In: IEEE Symposium on Security and Privacy (SP). pp. 1143–1161. IEEE (2021)

22. Jueckstock, J., Kapravelos, A.: Visiblev8: In-browser monitoring of javascript in the wild. In: Proceedings of the Internet Measurement Conference. pp. 393–405. ACM (2019). https://doi.org/10.1145/3355369.3355599
23. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser fingerprinting: A survey. ACM Transactions on the Web **14**(2), 1–33 (2020)
24. Liu, Y., Deng, G., Li, Y., Wang, K., Wang, Z., Wang, X., Zhang, T., Liu, Y., Wang, H., Zheng, Y., et al.: Prompt injection attack against LLM-integrated applications. arXiv preprint arXiv:2306.05499 (2023)
25. Mudryi, M., Chaklosh, M., WĂłjcik, G.: The hidden dangers of browsing AI agents. arXiv preprint arXiv:2505.13076 (2025)
26. Myrmel, A.: Vard: Vector-assisted rule detection. https://github.com/andersmyrmel/vard (2024), accessed: 2026-02
27. Ollama: Ollama. https://ollama.com, accessed: 2026-06-22
28. Patel, S.: Malicious LLM prompts. https://huggingface.co/datasets/codesagar/malicious-llm-prompts (2024), accessed: 2026-02-08
29. Perplexity AI: Introducing comet. https://www.perplexity.ai/hub/blog/introducing-comet (2025), accessed: 2026
30. Security, L.: Prompt injection defender patterns. https://github.com/lasso-security/claude-hooks/blob/dev/.claude/skills/prompt-injection-defender/patterns.yaml (2024), accessed: 2026-02
31. Su, J., Kapravelos, A.: Automatic discovery of emerging browser fingerprinting techniques. In: Proceedings of the ACM Web Conference 2023. pp. 2178–2188 (2023)
32. Team, M.L.: Prompt-guard-86m. https://huggingface.co/meta-llama/Prompt-Guard-86M (2024), accessed: 2026-02-08
33. Vekaria, Y., Canino, A.L., Levitsky, J., Ciechonski, A., Callejo, P., Mandalari, A.M., Shafiq, Z.: Big help or big brother? auditing tracking, profiling, and personalization in generative {AI} assistants. In: 34th USENIX Security Symposium. pp. 8115–8134 (2025)
34. Wang, S., Yu, F., Liu, X., Qin, X., Zhang, J., Lin, Q., Zhang, D., Rajmohan, S.: Privacy in action: Towards realistic privacy mitigation and evaluation for LLM-powered agents. arXiv preprint arXiv:2509.17488 (2025)
35. Wei, A., Haghtalab, N., Steinhardt, J.: Jailbroken: How does LLM safety training fail? Advances in neural information processing systems **36**, 80079–80110 (2023)
36. Xie, Q., Murali, M.V.K., Pearce, P., Li, F.: Arcanum: Detecting and evaluating the privacy risks of browser extensions on web pages and web content. In: 33rd USENIX Security Symposium. pp. 4607–4624 (2024)
37. Zhang, K., Su, Z., Chen, P.Y., Bertino, E., Zhang, X., Li, N.: LLM agents should employ security principles. arXiv preprint arXiv:2505.24019 (2025)
38. Zhang, K., Tenenholtz, M., Polley, K., Ma, J., Yarats, D., Li, N.: Browsesafe: Understanding and preventing prompt injection within AI browser agents. arXiv preprint arXiv:2511.20597 (2025)
39. Zharmagambetov, A., Guo, C., Evtimov, I., Pavlova, M., Salakhutdinov, R., Chaudhuri, K.: Agentdam: Privacy leakage evaluation for autonomous web agents, 2025. https://arxiv. org/abs/2503.09780
40. Zou, A., Wang, Z., Carlini, N., Nasr, M., Kolter, J.Z., Fredrikson, M.: Universal and transferable adversarial attacks on aligned language models. arXiv preprint arXiv:2307.15043 (2023)