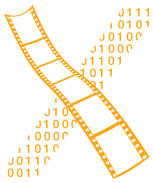


Chair for Computer Science 10 (Media Computing and Human-Computer Interaction)

RWTH AACHEN UNIVERSITY

DPGuard: An LLM-Based Browser Extension to Remove Deceptive Patterns In Situ

Bachelor's Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Lars vom Bruch

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr-Ing. Ulrik Schroeder

Registration Date: 07.07.2025
Submission Date: 03.11.2025

Eidesstattliche Versicherung
Declaration of Academic Integrity

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Student ID Number (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare under penalty of perjury that I have completed the present paper/bachelor's thesis/master's thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt; dies umfasst insbesondere auch Software und Dienste zur Sprach-, Text- und Medienproduktion. Ich erkläre, dass für den Fall, dass die Arbeit in unterschiedlichen Formen eingereicht wird (z.B. elektronisch, gedruckt, geplottet, auf einem Datenträger) alle eingereichten Versionen vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without unauthorized assistance from third parties (in particular academic ghostwriting. I have not used any other sources or aids than those indicated; this includes in particular software and services for language, text, and media production. In the event that the work is submitted in different formats (e.g. electronically, printed, plotted, on a data carrier), I declare that all the submitted versions are fully identical. I have not previously submitted this work, either in the same or a similar form to an examination body.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen/Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
§ 156 StGB (German Criminal Code): False Unsworn Declarations
Whosoever before a public authority competent to administer unsworn declarations (including Declarations of Academic Integrity) falsely submits such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment for a term not exceeding three years or to a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
§ 161 StGB (German Criminal Code): False Unsworn Declarations Due to Negligence
(1) If an individual commits one of the offenses listed in §§ 154 to 156 due to negligence, they are liable to imprisonment for a term not exceeding one year or to a fine.
(2) The offender shall be exempt from liability if they correct their false testimony in time. The provisions of § 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Contents

List of Figures and Tables

Abstract

Deceptive patterns are manipulative interface designs that influence users to act against their intentions. While prior research has focused on detecting and classifying these patterns, few technical solutions exist to mitigate them directly in the browser. We present *DPGuard*, a browser extension that leverages Large Language Models (*LLMs*) to remove deceptive patterns in situ, modifying the webpages' content in real time. Building on existing research approaches, *DPGuard* processes webpage elements selected by the user, transforms them into less manipulative versions, and reinserts them into the webpage while preserving functionality and privacy. The system supports both local and remote LLMs and is easily extendable with new models.

A technical evaluation on prebuilt and real-world websites compares *DPGuard*'s performance across multiple configurations. Results show that *DPGuard* effectively reduces manipulative elements on simpler pages but faces challenges with complex, dynamic websites. We conclude with limitations and future work, emphasizing the potential of LLM-based interventions but also the need for further refinement to handle diverse web content robustly.

Überblick

Deceptive Patterns sind manipulative Designs, die Nutzer dazu verleiten, gegen ihre eigentlichen Absichten zu handeln, zum Beispiel etwa unnötigem Tracking zuzustimmen oder ungewollte Abonnements abzuschließen. Während sich bisherige Forschung vor allem auf die Erkennung und Klassifizierung solcher Muster konzentriert hat, existieren nur wenige technische Lösungen, die sie direkt beheben. Diese Arbeit stellt *DPGuard* vor, eine Browser-Erweiterung, die Large Language Models (*LLMs*) nutzt, um die manipulativen Designs direkt auf der Website zu verändern. Aufbauend auf bestehenden Ansätzen verarbeitet *DPGuard* ausgewählte Elemente, macht sie weniger manipulativ und fügt sie wieder in die Seite ein, wobei die Funktionalität erhalten bleiben soll. Das System unterstützt sowohl lokale als auch serverseitige LLMs und ermöglicht eine einfache Integration neuer Modelle. Eine technische Evaluation auf vorgefertigten und realen Websites vergleicht die Leistung von *DPGuard* über mehrere Konfigurationen hinweg. Die Ergebnisse zeigen, dass manipulative Elemente auf einfacheren Seiten effektiv reduziert werden, komplexere Websites aber zu Problemen führen. Abschließend werden Einschränkungen und zukünftige Arbeiten diskutiert, wobei das Potenzial LLM-basierter Gegenmaßnahmen, zugleich aber die Notwendigkeit weiterer Verbesserungen hervorgehoben wird, um vielfältige Webinhalte robust zu verarbeiten.

Acknowledgements

Firstly, I would like to thank Prof. Dr. Jan Borchers and Prof. Dr-Ing. Ulrik Schroeder for examining my thesis.

Additional thanks go to my advisor, René Schäfer, for his continuous support and guidance throughout the development of this work.

Specifically, I want to thank my friends Jonas Broeckmann, Pascal Raabe, Julian Wallerius, Erik Østlyngen, and David Kopczynski for their advice and feedback on both the technical and written parts of this thesis.

Finally, I would like to express my gratitude to my family, especially my parents, for their encouragement and understanding during the course of this project.

Conventions

Throughout this thesis, we will use the following conventions.

Deceptive patterns (DPs) and dark patterns are synonymous. We use the term “deceptive patterns” as suggested by the ACM ethics board.

The text will reference deceptive patterns as described by Gray et al. [2024a].

Definitions of technical terms or short excursuses are set off in colored boxes.

Excursus:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition: Excursus.

Source code and implementation symbols are written in typewriter-style text: `myClass`.

Numerical values are given in SI units where applicable and rounded to two decimal places.

The whole thesis is written in American English.

Chapter 1

Introduction

The rapid growth of digital services and online commerce has led to an increasing reliance on web-based interfaces for everyday tasks, ranging from shopping and communication to accessing public services. While these interfaces are designed to guide users through the web experience, they frequently employ *deceptive patterns* [Mathur et al., 2019].

Dark Patterns / Deceptive Pattern:
Initially coined by Harry Brignull, *deceptive patterns* are now defined as: "tricks used in websites and apps that make you do things that you didn't mean to, like buying or signing up for something"[1].

Definition: Dark / Deceptive Pattern.

Examples include misleading content banners, such as cookie-banner popups that aim to make users accept all data collection, or bad pre-selected defaults that exploit cognitive biases [Bösch et al., 2016]. Often, users are unaware of the manipulation [M. Bhoot et al., 2021], or are unable to oppose such manipulative influences effectively [Bongard-Blanchy et al., 2021]. As such, understanding, identifying, and mitigating deceptive patterns is crucial for creating ethical and user-friendly web experiences.

Deceptive patterns manipulate users into doing something against their best interest.

[1] https://www.deceptive.design/, *Accessed August 2025*

Deceptive patterns are used all over the web.

Recent research has highlighted the widespread use of deceptive patterns across various online platforms. For instance, a 2024 report[2] by the Federal Trade Commission (FTC) studied the use of deceptive patterns in subscription services and found that over 75% of the 642 platforms that offered subscription services employed at least one deceptive pattern to influence user decisions.

Existing research lacks practical countermeasure implementations.

While the identification and detection of deceptive patterns have been the focus of research [Mathur et al., 2019; Chen et al., 2023; Nayak et al., 2024], the mitigation of deceptive patterns is still a relatively underexplored area. Recent research additionally focuses on assessing how mitigations should be presented to the user, but lacks practical, technical implementations of these mitigations in real-world scenarios [Schäfer et al., 2023; 2024].

Existing tools only detect specific DPs, without mitigating them.

Existing tools and browser extensions that aim to address deceptive patterns, such as the "Dapde Pattern Highlighter"[3] or "Insite"[4] on GitHub, aim at detecting and *highlighting* deceptive patterns. These tools often support only a limited set of known deceptive patterns and may not be regularly updated to keep pace with the evolving landscape of web design. Additionally, they typically do not offer cross-browser compatibility, limiting their broader accessibility.

LLMs are a promising tool for direct deceptive pattern mitigation.

To tackle the lack of technical countermeasures, Schäfer et al. [2025] propose the use of Large Language Models (LLMs) to mitigate deceptive patterns on websites directly, highlighting its feasibility by modifying exemplary HTML and CSS code. For that, they developed a prompt that can be used to instruct LLMs to directly mitigate deceptive patterns in web interfaces, skipping the need for a separate detection step. Their research shows promising results, with 91% of the self-

[2] https://www.ftc.gov/news-events/news/press-releases/2024/07/ftc-icpen-gpen-announce-results-review-use-dark-patterns-affecting-subscription-services-privacy, *Accessed August 2025*

[3] https://github.com/Dapde/Pattern-Highlighter, *Accessed August 2025*

[4] https://github.com/NicholasTung/dark-patterns-recognition, *Accessed August 2025*

created web elements being less manipulative after applying the LLM-based mitigation.

In this thesis, we will assess how the research by Schäfer et al. [2025] can be implemented in order to mitigate deceptive patterns on real-world web interfaces. We will investigate how LLMs can be used to counteract deceptive patterns in situ, providing users with another way of counteracting manipulative design choices while browsing the web.

This thesis explores real-world applicability of LLM-based mitigations.

We present *DPGuard*, a browser extension that leverages LLMs to directly mitigate deceptive patterns in situ as users interact with web pages, accessing and modifying the underlying Document Object Model (DOM). The *DOM* is the data structure used for websites and their HTML code. Browser extensions are mentioned as promising tools for this task [Mathur et al., 2019], as they can be easily installed by users and operate directly within the web browser [Schäfer et al., 2025]. We explore different ways of using the extension, including having the user choose which part of a website they want to apply mitigations to, and automatic mitigation, where the extension applies mitigations automatically. Besides focusing on the feasibility of real-world LLM-based deceptive pattern mitigation in general, we also address privacy concerns by running LLMs locally, thus avoiding the need to send user data to external servers. Additionally, we investigate intrusiveness, ensuring that the browser extension does not disrupt the user experience or interferes with the webpages' intended functionality.

DPGuard aims to implement LLMs as a direct technical countermeasure, without the need for an additional detection step.

In the following chapters of this thesis, we will present related work regarding deceptive patterns and their mitigations in Chapter 2. In Chapter 3, we will explain design requirements and the architecture of *DPGuard*. In Chapter 4, we will detail the implementation of the browser extension, followed by a technical evaluation of its effectiveness in Chapter 5. We will contextualize our findings and discuss future work in Chapter 6, closing with a summary of our contributions in Chapter 7.

Chapter 2

Related Work

2.1 Identification and Classification

Deceptive patterns are categorized and exposed to raise awareness of manipulative design.

In the last 15 years, researchers and taxonomists have come to understand deceptive patterns through studies and examples. In 2010, Harry Brignull coined the term "dark patterns" and created a website[1] showcasing different categories of deceptive design and a "Hall of Shame", exposing companies using deceptive patterns. An example is displayed in Figure 2.1. This was an important step for raising public awareness about deceptive design practices and identifying deceptive behavior in user interface designs. These original categories continue to provide a base for taxonomies in modern research [Gray et al., 2018; Mathur et al., 2019; Chen et al., 2023].

[1] https://deceptive.design, *Accessed September 2025*

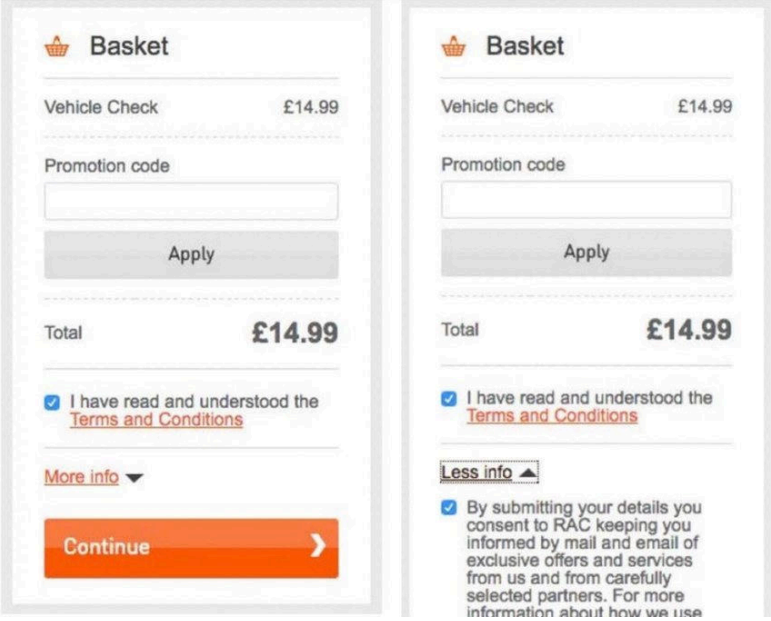


Figure 2.1: Example, taken from Gray et al. [2018], of the deceptive pattern *Preselection* as a type of *Interface Interference*, when trying to pay on a website. The preselected option of subscribing to a newsletter is hidden behind an expandable element. In order to continue without subscribing, the user has to expand the element and deselect the option actively.

Systematic inquiry of deceptive patterns is important.

Approximately at the same time as Brignull's first work, Conti and Sobiesk [2010] introduced one of the earliest academic taxonomies of what they identified as "Malicious Interface Design Patterns". Researchers systematically identified a series of strategies for deception that included *Misdirection*, *Forced Actions*, and *Hidden Information*. This added a layer of organization and expectation on the role of the components of an interface with the intent to manipulate the user. The fact that their taxonomy provided an alternative perspective to Brignull's functional classification and the academic efforts directed attention to the need for serious and systematic inquiry into the deceptive patterns within user interface design [Conti and Sobiesk, 2010].

Supplementing Brignull's early work, a popular work by Gray et al. [2018] studied and collected 118 artifacts from

numerous online platforms, such as Facebook, Reddit, and Wired.com. Based on these findings, Gray et al. [2018] proposed a revised taxonomy. Their taxonomy contains only five generic categories: *Nagging*, *Obstruction*, *Sneaking*, *Interface Interference*, and *Forced Action*. These categories aim to capture the essence of the classifications proposed by Brignull, while also being broad enough to encompass new and emerging patterns [Gray et al., 2018].

Brignull's early work was extended continuously.

Further attempting to grasp the variety of DPs, Mathur et al. [2019] carried out a large-scale, empirical study on deceptive patterns on the web that identified over 1,800 examples across thousands of websites [Mathur et al., 2019]. Their study further justified and built on previous taxonomies, providing evidence and context regarding the different types of deceptive practices and the varieties of instances and different sites on which they occurred. Mathur et al. [2019] also researched the effects of deceptive patterns on users, demonstrating that deceptive designs can shape choice and affect autonomy.

Large-scale empirical studies help to grasp the variety of DPs.

A key issue with the examination of deceptive patterns is the inconsistent terminology used across different research efforts. Many researchers have independently developed their own taxonomies and labels for similar or even identical categories of deceptive design [Gray et al., 2018; Mathur et al., 2019]. Conti and Sobiesk [2010] used names such as *Forced Work* and *Interruption*, while Brignull used *Nagging* and *Obstruction* to describe similar concepts. Furthermore, taxonomies are often used to improve future work, and so remote (as a latest addition) variants often appear, leading to a proliferation of labels that can be confusing and overlapping. This lack of standardization complicates efforts to compare findings, aggregate data, and start building an understanding of the area. It could also make it more difficult for practitioners and policymakers to put research into action, since there are many different definitions and classification terms that overlap in a range of meanings [Gray et al., 2024a].

Inconsistencies across different research efforts motivate the need for a consensus on the terminology.

Gray et al. [2024a] published an ontology to combine different taxonomies.

To address the problems of research using different and possibly incompatible definitions for categories of DPs, Gray et al. [2024a] have published an ontology that aims to provide a consensus about these categories. They conducted a systematic review of existing taxonomies and ontologies, identifying commonalities and differences among them. Their work synthesizes previous efforts and proposes a unified framework that can be used for future research and practice in the field of deceptive design [Gray et al., 2024a]. A key contribution of this ontology is the differentiation between various levels of deceptive patterns:

- *Low-level* patterns: Refer to specific interface elements or manipulations, such as *Immortal Accounts* or *Complex Language*.
- *Meso-level* patterns: Describe combinations of low-level patterns that work together to achieve a broader deceptive goal, like *Hiding Information* or *Urgency*.
- *High-level* patterns: Encompass systemic strategies, represented by the same five categories previously introduced by Gray et al. [2018]: *Nagging*, *Obstruction*, *Sneaking*, *Interface Interference*, and *Forced Action*.

This multi-level approach helps clarify the relationships between different types of deceptive practices and supports a more nuanced analysis of how they operate in digital environments [Gray et al., 2024a].

2.2 Countermeasures

Users struggle to recognize and resist deceptive patterns.

When encountering DPs, users often feel frustrated, as they struggle to navigate interfaces that are designed to manipulate their choices [Conti and Sobiesk, 2010]. The sheer variety and subtlety of these patterns make them difficult for users to recognize, let alone avoid. Most users lack the expertise or awareness to identify manipulative elements, and even when they are aware, research indicates that they

are frequently unable to resist or circumvent these influences [Bongard-Blanchy et al., 2021]. These issues motivate the need for effective countermeasures, as mentioned in a recent workshop that aimed at mobilizing research and regulatory action against deceptive patterns [Gray et al., 2024b].

Different countermeasures have been proposed to mitigate deceptive patterns.

To combat the variety of DPs, researchers have proposed various countermeasures. Bongard-Blanchy et al. [2021] has broadly categorized them into *educational*, *design*, *technical*, and *regulatory* interventions. Each category aims to provide four different scopes: *Awareness*, *Detection*, *Resisting*, and *Elimination*. *Eliminating* countermeasures are particularly interesting, as they do not require a deeper knowledge of the field and can be applied automatically, visually presenting their results [Schäfer et al., 2024]. Nevertheless, raising general awareness about DPs can empower users, developers, and designers to recognize and avoid manipulative designs [Gray et al., 2018].

LLMs are a promising tool for mitigating deceptive patterns.

Current advances in research have explored using large language models (*LLMs*) to apply mitigations to online deception. For example, LLMs can be used to help users automatically manage cookie consent banners [Porcelli et al., 2024]. Additionally, LLMs have been employed to detect DPs in user interfaces. Early successful attempts include using GPT-3.5 Turbo to identify deceptive patterns in text snippets [Sazid et al., 2023]. Using Generative AI, Mills and Whittle [2023] have been able to detect DPs by using HTML/JavaScript code and screenshots of websites.

Generative AI:
"Generative AI refers to deep-learning models that can generate high-quality text, images, and other content based on the data they were trained on."[2] ChatGPT[3] is an example of Generative AI.

[2]https://research.ibm.com/blog/what-is-generative-AI, *Accessed September 2025*

[3]https://chatgpt.com, *Accessed October 2025*

More recently, Schäfer et al. [2025] proposed a system that only implicitly contains a detection step, as it directly applies less manipulative alternatives to webpages, rather than merely identifying the presence of DPs.

2.3 Projects

Browser extensions are said to be promising tools to mitigate DPs.

According to Bongard-Blanchy et al. [2021], browser extensions are a technical intervention method when dealing with DPs. Browser extensions are perceived as a promising tool to apply countermeasures directly [Mathur et al., 2019]. There are projects published on *GitHub*[4] that aim to detect DPs while browsing the web.

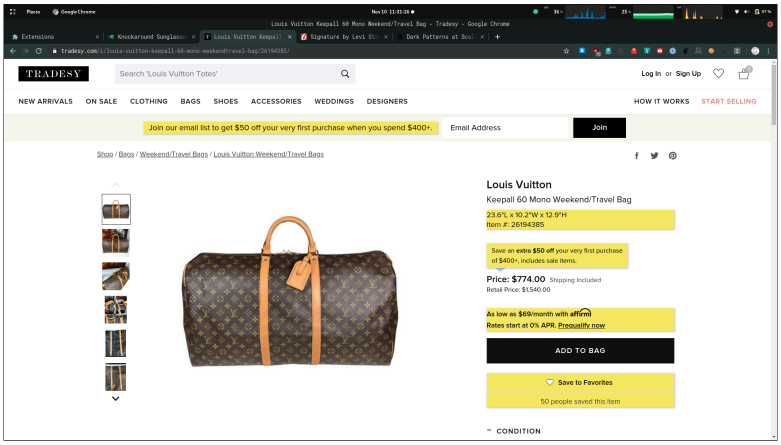


Figure 2.2: Screenshot of the *Insite*[5] browser extension as presented on their GitHub. The yellow elements highlight detected deceptive patterns on the website itself.

Insite detects and highlights DPs in text.

A browser extension called *Insite*[5] used the dataset of Mathur et al. [2019] to train a machine learning model that detects DPs in real-time while browsing the web. The detected elements are highlighted in yellow (see Figure 2.2), includ-

[4]https://github.com/topics/dark-pattern, *Accessed September 2025*

[5]https://github.com/NicholasTung/dark-patterns-recognition, *Accessed September 2025*

ing a popup that provides additional information about the category of the detected pattern. To realize this, the extension relies on a backend server hosting its machine learning model for text classification. *Insite* is theoretically not constrained in the number of DPs it can detect, although currently being limited to recognizing the categories from Mathur et al. [2019].

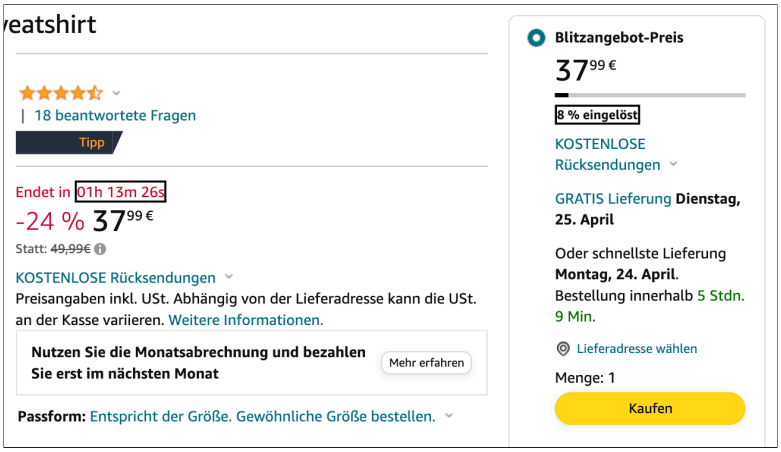


Figure 2.3: Screenshot (taken from their GitHub) of detected deceptive patterns on *amazon.de*[6] showcasing the *Dapde Pattern Highlighter*[7] browser extension. The highlighted elements are denoted with a black border around them. In this example, a *Countdown* (left) and *Scarcity* (right) pattern are detected.

DAPDE only detects and highlights specific DPs.

As a part of the *Dark Pattern Detection Project* (DAPDE)[7] , a team of researchers and developers at the University of Heidelberg created the *Dapde Pattern Highlighter* browser extension, receiving support from the German Research Institute for Public Administration. The extension detects and highlights (see Figure 2.3) a given set of patterns: *Countdown*, *Scarcity*, *Social Proof*, and *Forced Continuity* (see Figure 2.4). To account for not all DPs being present in a single frame, the extension creates two temporary copies of the webpage. This

[6]https://amazon.de, *Accessed September 2025*
[7]https://dapde.de/de/, *Accessed September 2025*

Figure 2.4: Popup of the *Dapde Pattern Highlighter*[8] browser extension, including the amount of detected deceptive patterns per language (DE). The popup also showcases a list of supported patterns and the ability to toggle the detection.

allows for the detection of patterns that may appear after a user interaction.

The *Dark Surfer Extension* uses LLMs to detect and highlight DPs.

Relating to this thesis, another browser extension called the *Dark Surfer Extension*[9] aims to detect DPs using LLMs. The extension uses custom models based on a simple self-trained LLM. Similar to *Insite*, the extension relies on a backend server to host the models, while being able to detect only a limited set of 5 patterns. After having detected DPs, the extension alerts the user and highlights the detected elements.

DPGuard aims to directly mitigate DPs using LLMs, skipping the detection step.

Overall, related work has shown that detecting DPs is not an easy task, as the variety of patterns is vast and the context in which they appear is often complex. This thesis aims to contribute to this field through the development of a browser

[8]https://github.com/Dapde/Pattern-Highlighter, *Accessed September 2025*

[9]https://github.com/Venkateeshh/DarkSurfer-Extension, *Accessed September 2025*

extension that directly implements Schäfer et al. [2025]'s approach, leveraging LLMs to skip the detection step and directly apply mitigations.

Chapter 3

The *DPGuard* Browser Extension

DPGuard is a browser extension that uses LLMs to mitigate deceptive patterns in situ.

Current solutions for defending against online deception often rely on rigid, rule-based systems that struggle to adapt to the evolving tactics of malicious actors. Furthermore, these systems mostly only provide warnings or highlights, leaving users to deal with the deceptive content themselves manually. *DPGuard* aims at providing a more flexible and user-friendly approach by leveraging the capabilities of LLMs to directly modify deceptive content in situ, meaning directly within the user's web browser and webpage contents, rather than having a separate detection and mitigation step. We aim to contribute a direct technical countermeasure without the additional need for user education or training. For that, we designed and implemented a browser extension that can connect to LLMs in order to identify and mitigate deceptive content in situ. In order to successfully prompt the LLM to perform the desired modifications, we used and adapted the advanced prompt (found in Listing A.1), designed by Schäfer et al. [2025].

3.1 Requirements

For the design of *DPGuard*, we followed several key principles to ensure it is effective and easy to use, for the user and for developers who want to extend or modify it. We wanted to keep the system modular, allowing developers and researchers to easily swap out components like the LLM. To allow for an organized and scalable codebase, we analyzed the requirements and derived the main components of the system, especially focusing on the interaction between the LLM, the users, and the Document Object Model (DOM).

Functional Requirements

New LLMs should be easily integrable into *DPGuard*.

The first of four functional requirements is the interaction with the LLM, as it is the base of the system. One of the main design goals was to allow both local and remote processing. For that, it should provide the ability to easily support new LLMs. Generally, the interaction with the LLM should allow for:

- A common interface for easy integration of new LLMs
- Local and remote processing
- Receiving HTML elements and metadata
- Returning HTML elements and metadata

The user should be able to select elements on the webpage.

The user interaction requirements focus on providing a simple and intuitive interface for users to interact with the extension. Its main features include the ability to show and hide an overlay in which the user can manually select elements they want to send for processing. The user should be able to:

- Show and hide the overlay
- Select HTML elements
- Undo, redo, and toggle changes

- Change settings
- View the state of ongoing processes
- Cancel ongoing processes

To be as transparent as possible, the overlay should display progress information during processing. In addition, by allowing undo, redo, and toggle functionality, we implement the *Switch (SW)* countermeasure introduced by Schäfer et al. [2023]. This countermeasure enables the user to switch between the unmodified and modified versions of elements. Furthermore, users should be able to modify settings, which will be discussed in more detail in Chapter 4.

The user should be able to toggle between original and modified content.

The most important requirement is the *Node Hierarchy*, as it is responsible for providing an intuitive interface when working with custom HTML elements. Since the extension heavily relies on the DOM content of webpages, it is crucial to have a robust and efficient structure, which should support:

We need a custom interface for working with complex HTML elements.

- Parsing and unparsing of HTML elements
- Calculating element size
- Evaluating and filtering elements
- Merging LLM responses into a compatible format

We should provide several methods for interacting with the DOM, as this is the only way *DPGuard* can interact with the webpage's content. More specifically, it should be able to:

- Retrieve and replace HTML elements
- Modify CSS styles and stylesheets
- Mark modified elements

Its ability to modify CSS styles and stylesheets is essential, as some mitigations may require changes to the visual appearance of elements outside their private attributes. Additionally, marking modified elements is crucial for later retrieval.

CSS modification is crucial for effective mitigations.

For the final functional requirement, the extension should support basic browser extension features, such as:

- Message passing between components
- Storage for settings and history

This is especially important for storing user settings and the history of modified elements. Additionally, a robust communication protocol between the different components of the extension is necessary to ensure smooth operation.

Non-functional Requirements

- Performance
- Privacy and Security
- Compatibility
- Documentation

Security and privacy are crucial for a multi-platform browser extension.

For non-functional requirements, performance is a key consideration, as the extension should be able to process elements quickly and efficiently, minimizing latency and ensuring a smooth user experience. Since we are dealing with potentially sensitive browsing data, privacy is another important requirement. The extension should be able to process data locally whenever possible, minimizing the amount of data sent to remote servers. Security and protection against malicious scripts are crucial for interacting with the DOM of webpages. The extension should be designed to minimize the risk of cross-site scripting (*XSS*) attacks and other security vulnerabilities, for example, by only requesting necessary permissions in the manifest file. As for compatibility, the extension should be able to work with a wide range of webpages and web applications, regardless of their structure or content. This includes being able to run in different browsers. Finally, documentation is an important non-functional requirement, as it helps developers and users understand how to use and extend the codebase.

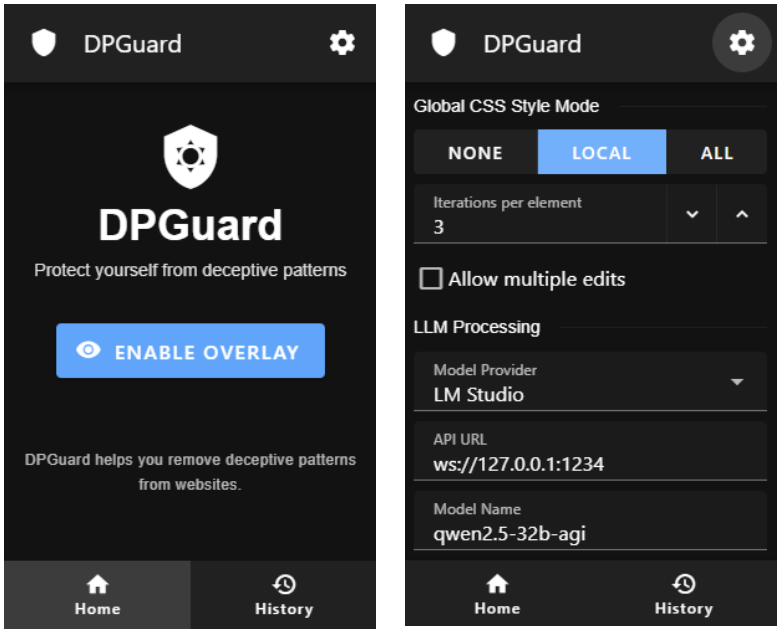
3.2 User Interface Design

Another crucial aspect of the *DPGuard* extension is its user interface, which is designed to be intuitive and straightforward, allowing users to easily interact with the extension and understand its functionality. The main components of the user interface include the *popup*, providing the main access point to the extension's features, and the *overlay*, which allows users to select and interact with elements on the webpage.

3.2.1 Popup

The popup provides access to the main features of the extension.

Providing the main UI for the extension, the popup is accessible by clicking on the extension icon in the browser toolbar. Figure 3.1 shows the home screen and the settings screen of the popup. The home screen (1) allows the user to enable or disable the selection overlay discussed in Section 3.2.2, as well as a short introduction to the extension's features. The settings screen (2) allows the user to configure (LLM-specific) settings, as already mentioned in the requirements. This includes the choice of LLM, the number of iterations to run for each request, and which CSS style mode to use. The number of iterations defines how often the LLM processes the same element. The style mode determines how much of the webpage's CSS styles is included when sending elements to the LLM, with options being inline styles, including style from the document's `<head>`, or including all computed styles. Additionally, the user should be able to configure whether the overlay should automatically close after a selection, indicated by the checkbox stating "Allow multiple edits". If enabled, the overlay should remain open after a selection, allowing the user to select and process multiple elements in one session.



(1) Home (2) Settings

Figure 3.1: The popup UI of the *DPGuard* browser extension, showing the home screen (1) and the settings screen (2). The home screen provides access to the main features of the extension, while the settings screen allows users to configure LLM-specific settings and other preferences.

The history screen provides access to previously modified elements.

For the last page of the popup, we implemented a history screen, shown in Figure 3.2. The history screen provides users with a list of previously mitigated elements, along with the ability to view more details or toggle between the original and modified versions, realizing the *Switch (SW)* visual countermeasure mentioned in the requirements. Each entry in the history list additionally includes metadata such as the date and time of the modification to make it easier for users to keep track of their actions. Users can click on an entry to view more details or revert changes. On the top right is a button to clear the entire history, in case the user wants to start fresh.

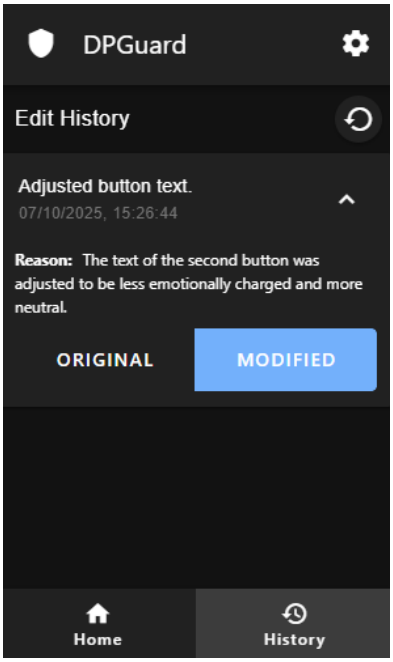


Figure 3.2: The history screen of the popup UI, showing (a list of) previously mitigated elements along with their metadata. Users can click on an entry to view more details or revert changes.

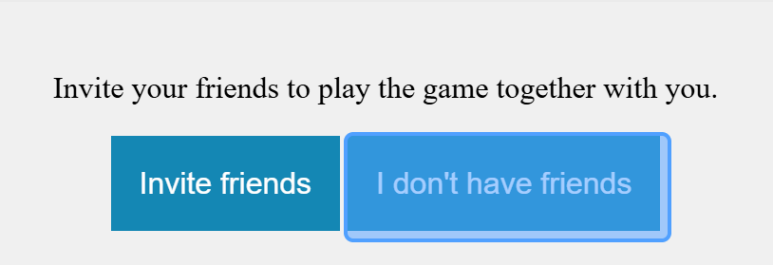
3.2.2 Overlay

The overlay allows users to select elements on the webpage for mitigation.

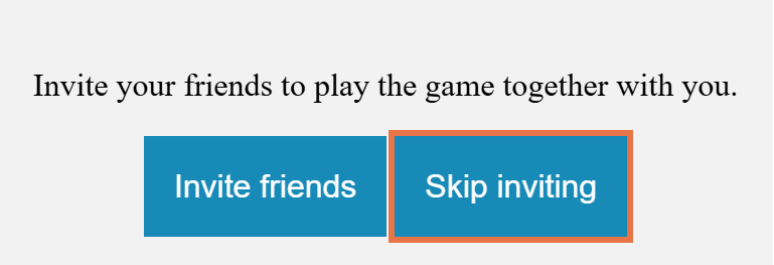
Since the overlay is the main way for users to interact with the webpage, it was designed to be unobtrusive and easy to use. The overlay provides a clear visual indication of when the extension is active, including the ability to close the overlay at any time. Figure 3.3 shows the two main features of the overlay: The element selection mode (1) and the history hover feature. The element selection mode allows users to click on elements they want to mitigate, highlighting currently hovered elements with a blue border. When clicked, the overlay will show a shimmer effect, indicating that the element has been selected and is being processed.

Modified elements are highlighted with a border in the history.

The history hover feature (2) provides users with the ability to quickly find modified elements on the page while browsing the history. When hovering over an entry in the history list, the corresponding element on the webpage is highlighted with a border in a different color, making it easier to find. This feature is particularly useful for users who want to review or revert changes made by the extension, as it allows them to quickly locate and identify modified elements without having to manually search through the webpage.



(1) Simple Overlay



(2) History Hover

Figure 3.3: The overlay UI of the *DPGuard* browser extension, showing the element selection mode (1) and the history hover feature (2). The selection mode allows users to click on elements they want to mitigate, while the history hover feature provides users with the ability to quickly find modified elements on the page while browsing the history.

Chapter 4

Implementation

Different browsers have different extension APIs.

Developing a browser extension comes with several challenges and limitations, especially when making it multi-platform. One of the main challenges is the limited set of APIs and capabilities provided by different browsers. While modern browsers like Chromium™, Firefox™, and Safari™ support a wide range of APIs, there are still differences in how these APIs are implemented and what features are available. Additionally, different browsers require different manifest file formats and have different policies regarding extensions, which adds complexity to the development process.

The extension is built using *WXT*, *Vue*, and *Vuetify* to support multiple browsers and streamline development.

To get around many of these challenges, we decided to use a library for creating a web extension called *WXT*[1]. *WXT* provides a unified API for creating extensions that work across multiple browsers, abstracting away many of the differences between them. It additionally includes a set of tools for building and deploying extensions, which integrate well into our *TypeScript*[2] project. By using *WXT*, we were able to focus on developing the core functionality of the extension without having to worry about the underlying differences

[1]https://wxt.dev/, *Accessed August 2025*

[2]https://www.typescriptlang.org/, *Accessed August 2025*

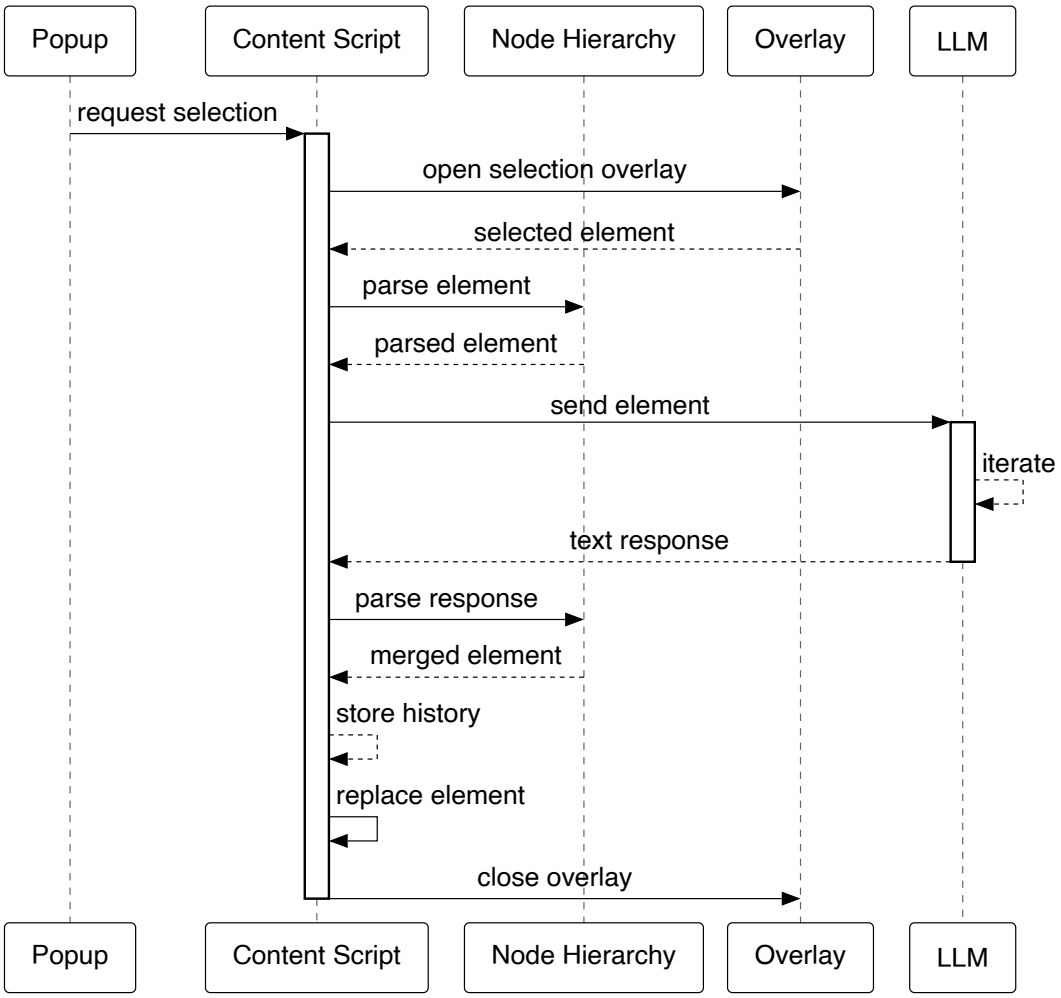


Figure 4.1: Sequence diagram showcasing the architecture and data flow of the *DPGuard* browser extension. The diagram illustrates how user interactions in the popup lead to element selection on the webpage, which is then parsed, analyzed by the LLM, and then sent back to the content script for updating the webpage accordingly.

between browsers. To streamline UI development, we also used the framework *Vue*[3] together with the UI library *Vuetify*[4], which provide a set of pre-built components and styles that can be easily customized to fit the needs of the extension. To further enhance the development process, we

[3]https://vuejs.org/, *Accessed August 2025*
[4]https://vuetifyjs.com/en/, *Accessed August 2025*

used a multi-platform messaging library called *WebExt-Messaging*[5], allowing us to easily send typed messages between the different components of the extension.

The manifest file defines a single content and background script with minimal permissions.

At the core of any extension is the *manifest file*. It provides important metadata about the extension, such as its name, version, and permissions. The manifest file also specifies the different components of the extension, such as the background script, content scripts, and popup. Since we are using *WXT*, we only need to provide a single manifest file, which is then automatically converted to the appropriate format for each browser during the build process. Specifically, our extension requires the *storage* permission and specifies a single content and background script.

The sequence diagram in Figure 4.1 illustrates the architecture and data flow of the *DPGuard* browser extension. In the following sections, we will discuss the individual components and their implementation in more detail, aligning with the data flow depicted in the diagram and the requirements outlined in Chapter 3.

4.1 Node Hierarchy

A custom *Node Hierarchy* abstracts interaction with DOM elements.

Even though the available DOM hierarchy implementation provides a robust interface for interacting with HTML elements, it lacks certain features that are crucial for our use case. To fulfill the requirements outlined in Chapter 3 and address these limitations, we implemented a custom class hierarchy depicted in Figure 4.2, for representing and interacting with HTML elements in the DOM. This *Node Hierarchy* is responsible for providing a structured representation of HTML elements, allowing for a much easier interaction and manipulation of the DOM. To achieve this, we created a base class called `DomElement`, which serves as the foundation

[5] https://www.npmjs.com/package/@webext-core/messaging, *Accessed August 2025*

for all other classes in the hierarchy. The `DomElement` class contains common properties and methods that are shared among all of our custom classes, such as the element's total and effective size, as well as a flag indicating whether the element should be ignored during LLM processing.

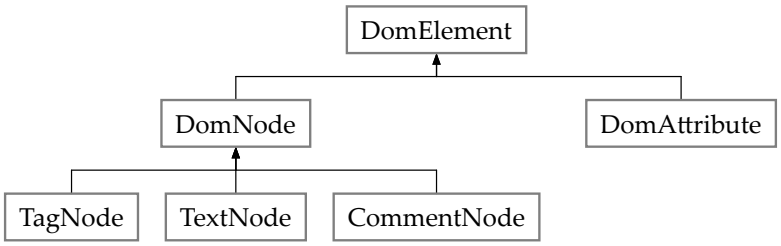


Figure 4.2: The class hierarchy of the DOM representation used in *DPGuard*. The `DomElement` class serves as the base class for all DOM nodes, with `DomNode` and `DomAttribute` as its direct subclasses. The `DomNode` class is further specialized into `TagNode`, `TextNode`, and `CommentNode`, representing different types of nodes in the DOM tree.

The `DomElement` class serves as the base class for all DOM nodes.

At the next level, we have two direct subclasses: `DomNode` and `DomAttribute`. The `DomNode` class represents nodes in the DOM tree, additionally implementing four methods: `parse`, `unparse`, `parseFromLLM`, and `unparseForLLM`. It is further specialized into three subclasses: `TagNode`, `TextNode`, and `CommentNode`. The `TagNode` class represents HTML tags, such as `<div>` or `<span>`, and contains properties for the tag name, attributes, and child nodes. The `TextNode` class represents text content within HTML elements, while the `CommentNode` class represents HTML comments. On the other hand, the `DomAttribute` class represents attributes of HTML elements, such as their `class` or `id`, and contains properties for the attribute's name and value.

The `Evaluator` reduces the amount of data sent to the LLM by filtering nodes.

One of the key features of our node hierarchy is its ability to evaluate and filter nodes based on certain criteria. Each class constructor takes an `Evaluator` object, which is responsible for determining the attributes of the `DomElement`. Its most important responsibility is to determine whether a node should be ignored for LLM processing, based on rules

depicted in Table 4.3, since reducing the amount of data sent to the LLM is crucial to avoid token limitations. Token limitations occur when the LLM has reached its maximum *context window*, meaning its maximum amount of data to process. Therefore, when evaluating whether a node should be ignored, the `Evaluator` has three possible actions: It can either decide to discard the node entirely, set its `ignoreForLLM` flag and thereby mark it for later replacement, and reduce the amount of data sent for processing, or it can keep the node as-is. The decision is based on the type of node, its attributes, and its content. For example, `CommentNodes` are always discarded, as they do not contribute to the visual representation of the webpage. To allow for more complex cases, we introduced two lists: the *ignore* and *irrelevant* lists. Each list contains strings whose occurrence is tested for in the currently evaluated node.

Type	Condition	Action
`TagNode`	found in tag irrelevant list	*discard*
	found in tag ignore list	*mark*
	otherwise	*keep*
`TextNode`	always	*keep*
`CommentNode`	always	*discard*
`DomAttribute`	found in attribute ignore list	*mark*
	otherwise	*keep*

Table 4.3: A decision table outlining the handling of different DomElement types during the parsing process. Possible actions include discarding the element, marking it for later replacement, or keeping it as-is.

The `Evaluator` also computes the total and effective size of each node.

Additionally, the `Evaluator` computes the total and effective size of each node by simply counting its characters, which is used to further filter out nodes when necessary, for example, when including the document's `<style>`. The total size represents the size of the node including all its children, while the effective size only counts nodes that are not ignored for LLM processing. This allows us to filter out nodes that are

too large based on their effective sizes, while still keeping track of their total sizes for other purposes.

Each class implements parsing and unparsing methods.

The `CommentNode` and `TextNode` classes are relatively simple, as they simply store the text content of the node. Both classes implement the parsing and unparsing methods, allowing them to be easily converted to and from their string representations. The LLM string representations differ slightly, as they contain marker comments for the ignored node instead of the actual node content.

The `TagNode` class handles HTML tags, attributes, and child nodes.

The `TagNode` class is the most complex class in our hierarchy, as it represents HTML tags, which can contain attributes and child nodes. Most of our implementation focuses on this class, as it is responsible for handling the majority of HTML elements in the DOM.

Parsing a `TagNode` involves recursively traversing the DOM tree and creating `DomElement` objects.

The parsing process of a `TagNode` involves recursively traversing the DOM tree and creating `DomElement` objects for each node. During this process, the `Evaluator` determines whether a child node should be included in the `TagNode`'s children. For each visited child, its attributes are also parsed into `DomAttribute` objects. The parsing process continues recursively for each child node, resulting in a tree-like structure of `DomElement` objects that represents the original DOM tree. The `TagNode`'s parsing is additionally able to handle the different style modes, optionally including the `<style>` from the documents `<head>`, or globally computed styles for each element, aligning with the settings outlined in Section 3.2.1.

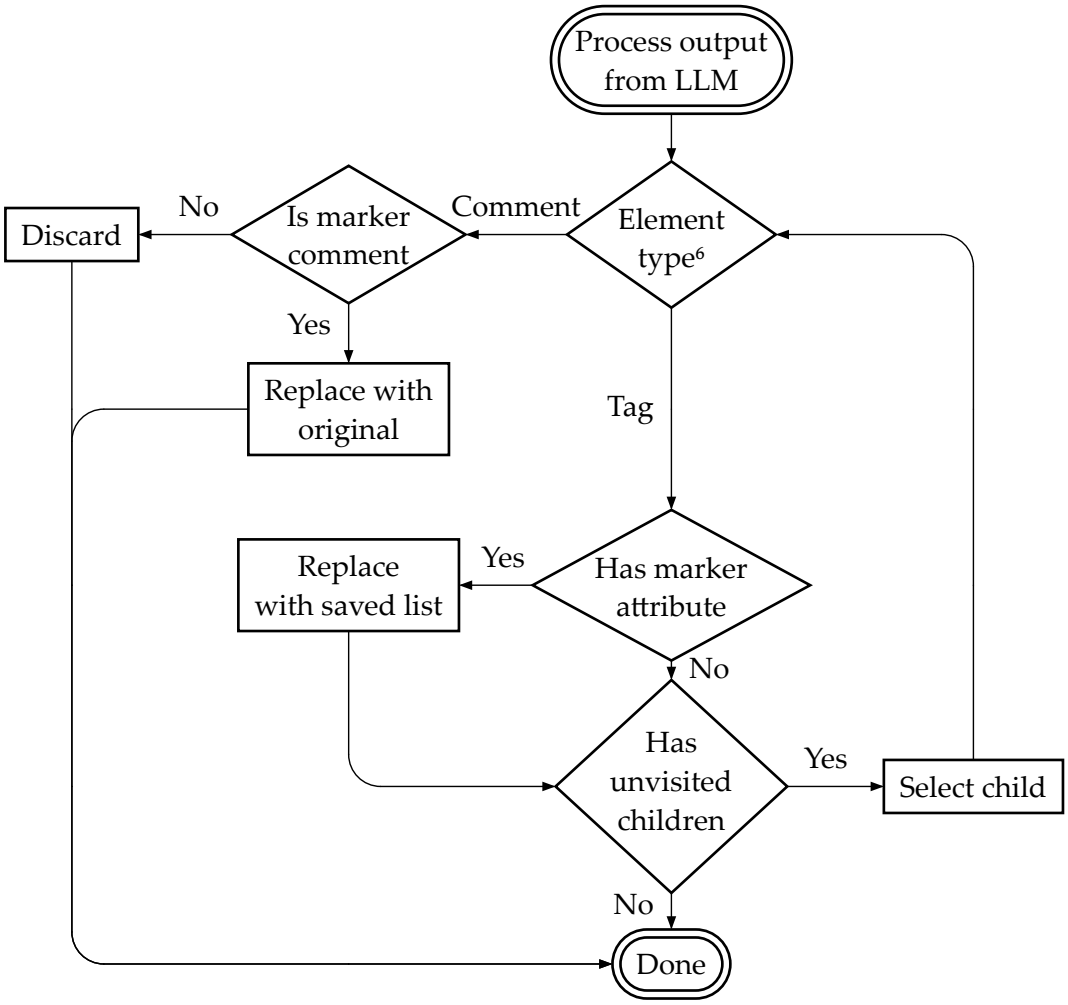


Figure 4.4: A flowchart illustrating the process of merging LLM-provided elements with elements that have not been sent for processing. For `TagNode` elements, marker attributes are replaced with saved attributes, and children are processed recursively. `CommentNode` elements that are identified as markers are replaced with their original counterparts.

The unparsing process is the reverse of parsing, converting a `TagNode` and its children back into an HTML string representation. During this process, the `TagNode` iterates over its children and attributes and calls their `unparse` method to obtain their string representations, adding them to the final output.

[6]Since `TextNode` elements are always kept as-it, they are not part of this process.

Unparsing for LLM processing involves creating and remembering marker nodes.

Furthermore, the `TagNode` implements the `parseFromLLM` and `unparseForLLM` methods, which are specifically designed for handling LLM processing. Unparsing for the LLM works similarly to the regular unparsing, but with a few key differences. First, any child nodes that are marked to be ignored for LLM processing are replaced with "marker comment nodes" and an ID, indicating that they should be replaced with their original counterpart later. We chose `"leave-{ID}"` to be the marker comment and adapted the prompt accordingly. Second, if any of its attributes are marked to be ignored, a marker attribute is added to the tag, representing all of the ignored attributes. This allows us to keep track of which attributes should be replaced later. The resulting string representation is then sent to the LLM for processing, accompanied by a mapping of marker IDs to their original nodes.

Parsing from LLM involves merging the modified tree with ignored nodes.

The parsing from LLM is more complex, as it involves merging the modified tree returned by the LLM with the original tree. The process, depicted in Figure 4.4, also behaves similarly to the regular parsing, but when encountering a marker comment or attribute, the original node or attributes are used instead of creating a new node. This ensures that any nodes or attributes that were ignored during LLM processing are preserved in the final tree structure. This process is repeated recursively for each child node until all children have been visited and merged.

All `DomElements` implement equality checking for finding differences.

At last, the `TagNode` class also provides a method for finding the smallest difference between two `TagNodes`. This is especially useful for implementing the *History Hover* feature depicted in Figure 3.3. When comparing two `TagNodes`, the method recursively traverses both trees, comparing each node and its attributes. If a difference is found, the method returns the node that is different, allowing us to easily identify which part of the element has changed.

4.2 LLM Client

The LLM client abstracts interaction with different LLM providers.

Since the LLM is a core component of the extension, we implemented a common interface for interacting with different LLM providers, called the `BaseClient`. The interface defines the basic method that any LLM client must implement: `processNode`, which takes a `TagNode`, the number of iterations to run for, and the timestamp of the request. The method returns a processed `TagNode`, along with metadata to be stored in the history. For each run of `processNode`, the client handles a series of steps. First, it unparses the `TagNode` into a string representation suitable for sending to the LLM (`unparseForLLM`), alongside its list of ignored nodes. Afterwards, it sends the unparsed string to the LLM for processing, possibly iterating multiple times. Specifically, this means that the result is piped back into the LLM for further refinement, aligning with promising research by Schäfer et al. [2025]. Once the LLM has finished processing, the client parses the modified HTML string back into a `TagNode` and merges it with the original node using the list of ignored nodes (`parseForLLM`). Since the LLM might have returned an identical element, we compute a `sha256` hash of the modified element and compare it to the original. If they are identical, we do not return any element or history information, as no changes were made. Otherwise, we return the modified element and the (history) metadata.

Local and remote LLM clients are implemented.

Currently, we have implemented two clients: local and remote. In order to be able to run LLMs locally, we used *LMStudio*[7], which provides a local API for interacting with various open-source models. The `LmStudioClient` uses LMStudio's API to send requests and receive responses. For remote LLMs, we implemented the `OpenAiClient`, which uses OpenAI's API to interact with *GPT-4o*[8]. Both clients

[7] https://lmstudio.ai/, *Accessed October 2025*

[8] https://openai.com/index/hello-gpt-4o/, *Accessed October 2025*

handle the specific details of communicating with their respective APIs while adhering to the common interface defined by `BaseClient`. This allows us to easily switch between different LLM providers without changing the core logic of the extension, for example, when adding support for more providers in the future.

4.3 Content Script

A content script is a JavaScript file that runs in the context of a web page, allowing it to interact with the page's DOM and modify its content. Specified in our manifest file, the content script is automatically injected into web pages matching the pattern `http*://*/*`, allowing it to run on all HTTP and HTTPS pages.

The content script orchestrates the extension's functionality.

We chose the content script as the core orchestrator of the extension, as its ability to interact with the DOM makes it ideal for handling user interactions, retrieving and replacing HTML elements, and coordinating communication between the different components of the extension. The content script is responsible for handling messages from the *Popup* and *Overlay*, managing the LLM processing pipeline, and updating the webpage with the modified elements. In the following sections, we will discuss its individual components and their implementation in more detail.

4.3.1 Overlay Communication

Direct injection of the overlay into the DOM has limitations.

The overlay, which allows the user to select elements and send them to the LLM for processing, is created and injected into the current webpage by the content script. Typically, the overlay would be comprised of a single `IFrame` injected into the DOM directly. This approach, however, has several limitations. The overlay is responsible for capturing user

input, such as mouse clicks and movement, to allow users to select elements on the webpage. If the overlay is injected directly into the DOM, it can interfere with the webpage's existing event listeners and styles, leading to unexpected behavior. Specifically, when trying to capture mouse events for elements that are rendered in a different context, such as within a shadow DOM, the simple overlay would not be able to access these elements directly, as the mouse events would be consumed by the hovered element.

Shadow DOM:
"[A] Shadow DOM enables you to attach a DOM tree to an element, and have the internals of this tree hidden from JavaScript and CSS running in the page."[9]

The overlay is rendered in a shadow DOM to avoid interference with the webpage.

To circumvent these limitations, we decided to render the overlay using an `IFrame` in a shadow DOM itself. This allows the overlay to encapsulate its styles and event listeners, preventing interference with the underlying webpage. However, this means that the overlay cannot access the webpage's content directly, as shadow DOMs are isolated from each other.

The overlay indirectly accesses the webpage's DOM via message passing.

To enable communication between the overlay and the content script, we implemented a message-passing system using the `window.postMessage` API. This way, the overlay can indirectly access the webpage's DOM by sending messages to the content script, which can then retrieve and modify elements on behalf of the overlay. The most important messages exchanged between the overlay and content script are:

- `iframe-mouse-move`: When the user moves their mouse over the webpage, the overlay sends this message to the content script, which responds with information about the element currently under the cursor. This allows the overlay to highlight elements as the user hovers over them.

[9] https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM, *Accessed October 2025*

- `iframe-mouse-down`: Sent by the overlay when the user clicks on an element. The content script responds by sending the selected element to the LLM for processing.
- `content-bounding-rect`: Sent by the content script as a response to `iframe-mouse-move`, containing the bounding rectangle of the element under the cursor. The overlay uses this information to position its highlight box correctly.

Consequently, messages for opening and closing the overlay also exist, as well as displaying progress information. Overall, this message-passing system allows the overlay to interact with the webpage's content indirectly, while still benefiting from the encapsulation and isolation provided by the shadow DOM.

4.3.2 DOM Interaction

The content script interacts with the DOM to retrieve and modify elements.

The content script constantly interacts with the DOM of the current webpage, as it is responsible for retrieving and replacing HTML elements, modifying CSS styles and stylesheets, and marking modified elements with a custom attribute. To retrieve an element based on the user's selection, the content script uses the `document.elementFromPoint` method, which returns the topmost element at the specified coordinates. This allows the content script to accurately identify which element the user has clicked on, even if it is nested within other elements or rendered in a shadow DOM.

Modified elements are marked with custom attributes.

When replacing an element in the DOM with its modified version, the content script additionally appends a custom attribute called `dpguard-timestamp`, which contains the timestamp of when the element was modified. Furthermore, it adds the `dpguard-timestamp-smallest` attribute, which marks the smallest modified element on the page, whose calculation was discussed in Section 4.1. To modify CSS styles and stylesheets, the content script can directly manipulate the `style` attribute of HTML elements or modify existing

`<style>` tags in the document's `<head>`. This allows the content script to change the visual appearance of elements as needed, for example, to highlight modified elements or adjust their layout.

Utility functions facilitate DOM interaction.

Furthermore, the content script provides several utility functions for interacting with the DOM, such as replacing an element by its `dpguard-timestamp`, replacing the root DOM element, or highlighting elements by their timestamp. These functions allow for easy retrieval and management of modified elements, as well as providing visual feedback to the user about which elements have been changed.

4.4 Background Script

The service worker initializes storage and computes hashes.

The background script, also called the service worker, is a JavaScript file that runs in the browser's background, separate from any web page. It is responsible for handling tasks that do not require direct interaction with the webpage, such as managing shared storage and computing hashes for element comparison. In our case, the background script does two things: Initializing the extension's storage with default settings if they do not exist yet, and providing a method for computing `sha256` hashes of HTML elements. The storage is used to persist user settings in *local* storage, while history data is only stored in *session* storage, losing its state when the browser or tab is closed, since that information is not relevant anymore.

Message passing is limited to serializable data.

Since the service worker is considered to be responsible for tasks that do not require direct interaction with the webpage's DOM, we thought about moving all LLM-related tasks to the background script, but decided against it, as message passing between the content and background script is limited to serializable data. This makes it impossible to communicate complex objects, such as our node hierarchy, between the components, which would have required additional serialization and deserialization logic.

4.5 History

The history feature tracks modifications and allows undo/ redo actions.

To keep track of modifications and revert changes if necessary, we implemented a history feature. This feature is comprised of two things: The `EditHistory` and `EditHistoryItem`. The `EditHistory` is responsible for managing a list of `EditHistoryItems`, which represent individual modifications made to elements on the webpage. Each `EditHistoryItem` contains the original and modified HTML elements, along with metadata provided by the LLM, such as the timestamp of modification, a summary of changes made, and the reason for the modification.

Undo and redo actions are performed iteratively.

The `EditHistory` keeps track of the current position in the history list, allowing users to undo and redo changes. Consequently, it also allows the user to jump to a specific point in the history, possibly performing multiple undo or redo actions at once. Rather than storing a snapshot of the whole webpage for each history item, we only store the original and modified element, as this allows the webpage to change without us undoing everything. Consequently, we need to ensure that multiple undo and redo actions are performed iteratively. For example, if a user makes three modifications to different elements on the page, and then wants to undo the last two changes, the `EditHistory` will first revert the most recent change, and then revert the second most recent change, resulting in the webpage being restored to its state before those two modifications were made. Replacing the whole webpage also deletes the history, as it is no longer valid.

The smallest modified element is highlighted when browsing history.

When the user browses the history, the content script uses the `dpguard-timestamp-smallest` attribute to identify the smallest modified element for that history item, highlighting it in the webpage (Figure 3.3). This is especially useful when the LLM only made minor modifications that did not affect the whole element, allowing users to easily identify which parts of the requested element were actually changed.

Chapter 5

Technical Evaluation

To test the effectiveness of our implementation, we conducted a systematic technical evaluation of the DPGuard extension. We designed a set of parameters and metrics to assess the tool's performance, accuracy, and usability. The evaluation was carried out in a controlled environment and partly automated. We chose to automate the evaluation as much as possible to ensure consistency and repeatability of the results. However, some aspects of the evaluation, such as categorizing the results, required manual inspection and judgment.

5.1 Methodology

Our evaluation closely follows the evaluation methodology of Schäfer et al. [2025], using the same LLM (GPT-4o) and similar parameters. However, we extended their approach by incorporating real-world web pages in addition to the prebuilt pages they used. This allowed us to assess the tool's performance in more realistic scenarios. Since Schäfer et al. [2025] already built a set of prebuilt pages, we reused one page of each category (nine deceptive websites and

The evaluation extends the work by Schäfer et al. [2025] with real-world webpages, additional metrics, and parameters.

one legitimate website). To align with their evaluation, we selected ten real-world pages, also including nine deceptive websites and one legitimate website, therefore providing the same diversity in page types. Table B.1 shows the real-world pages that were manually selected to ensure a representative sample.

We compared local and remote model performances.

In addition to using the remote GPT-4o model, we also evaluated DPGuard with `qwen2.5-32b-agi`, a local LLM said to have comparable[1] performance of OpenAI's GPT-4o. This gave us insight into how the tool performs with different LLMs, but does not allow for a direct comparison between the two models, since the local model's performance is heavily dependent on the hardware it runs on. In our case, we used a desktop PC with an Intel i5-13600KF, 32GB RAM, and an NVIDIA RTX 4090.

To then formally define the evaluation, we considered the following parameters, leading to a total of 240 evaluation runs:

- **Page Type:** Real-world or prebuilt pages (2×10)
- **LLM:** Local or remote ($\times 2$)
- **Iterations:** 1, 3, or 5 iterations of LLM processing ($\times 3$)
- **Style Mode:** None or intermediate (document's `<style>` found in `<head>`) ($\times 2$)

The results describe how well DPs were mitigated.

To be able to compare the results of the different runs, we would need a way to quantify the results. While we could use a simple binary metric (correct/incorrect), we found that this would not account for the varying degrees of correctness in the results. To address this, we split the results into five categories:

- **Removed:** The deceptive pattern was fully removed, and there are no manipulative elements left.

[1] https://dubesor.de/benchtable#cost-effectiveness, *Accessed October 2025*

- **Reduced:** The deceptive pattern was partially removed, but some manipulative elements are still present.
- **Unchanged:** The element remained unchanged.
- **Worsened:** The element was modified, but the manipulative elements were made worse.
- **No-op:** The LLM was not able to process the request, e.g., due to a timeout or an error.

We defined quantitative metrics to complement subjective judgments.

The results were decided and applied according to the Ontology by Gray et al. [2024a]. While these self-created categories provide a good overview of the results, they are still somewhat subjective and can be interpreted differently by different evaluators. In order to not only rely on subjective judgments, we additionally defined a set of quantitative metrics to measure the performance of the tool:

1. **Duration(ms):** The total time taken for the evaluation run, measured in milliseconds. This allows us to deduce the performance impact of different parameters.

2. **Total/Effective Size(yes/no):** The total and effective size (in character length) of the selected element helps us understand how much content is available and how much of it is actually processed.

3. **Unrelated Changes(yes/no):** Whether the LLM made changes to parts of the element that were not related to the deceptive pattern.

4. **Functionality(yes/no):** Whether the functionality of the element was preserved after the modifications.

For our methods of analysis, we manually inspected the results of each run and categorized them according to the defined categories. We mainly used descriptive statistics to summarize the results, such as counts and percentages for each category. For our quantitative metrics, we calculated averages and percentages to provide a more objective assessment of the tool's performance.

5.2 Results

The evaluation included a total of 240 runs, with results categorized and analyzed.

All runs were gathered and evaluated in a spreadsheet. After categorizing the results, we manually inspected the runs and outlined different trends and observations. For instance, we found that the remote LLM was generally more hesitant to make changes in either direction, resulting in more unchanged outcomes than the local LLM, but also had a higher number of removals or reductions. This is due to the local LLM producing more No-op results, likely because of its limited token length and context window when compared to the remote LLM. Table 5.1 shows a summary of the results for the different LLMs, additionally indicating that unchanged results were preferred over worsened results.

Result	**LLM: Local**	**LLM: Remote**	**Total**
Removed	14	17	31
Reduced	20	27	47
Unchanged	28	**46**	**74**
Worsened	12	10	22
No-op	**46**	20	66
Total	120	120	240

Table 5.1: A summary of the evaluation results, categorized by the outcome of each run and the type of LLM used (local or remote). Bold numbers indicate the highest occurrences in each category.

Functionality was generally preserved, and unrelated changes happened in 33% of cases.

Since preserving functionality is an important part of a successful in situ mitigation, Table 5.2 shows a breakdown of the functionality preservation and unrelated changes made by the LLMs *when making changes*, meaning that No-op results were excluded to allow for a better comparison. The results indicate that the local LLM was more likely to make unrelated changes, potentially due to its limited capabilities. However, both LLMs were generally able to preserve the

functionality of the elements, with prebuilt pages achieving a perfect score.

Parameter	Functionality	Unrelated Changes
Local	95.95%	40.54%
Remote	85.86%	26.26%
Prebuilt	**100.0%**	24.17%
Real-world	67.92%	**50.94%**
Total	90.17%	32.37%

Table 5.2: A percentage distribution of functionality preservation and unrelated changes made by the LLMs across different parameters (LLM type and page type).

Duration analysis showed the remote LLM was nearly 4 times faster than the local LLM.

To briefly assess model performances, we measured the average duration of each run, with results for the local and remote LLM being shown on a *logarithmic* scale in Figure 5.3. The boxplot indicates that the remote LLM was nearly 4 times faster than the local LLM, likely due to the difference in hardware and infrastructure. Additionally, we found that using no style mode was faster than using the intermediate style mode (as the LLM had to process less content), roughly halving the average duration.

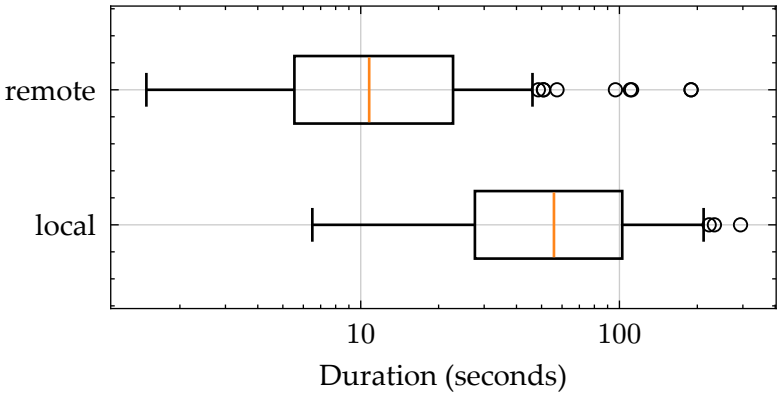


Figure 5.3: The durations (in seconds on a *logarithmic* scale) for local and remote LLMs on *all* page types and iterations. The remote LLM is nearly 4 times faster than the local LLM on average, with a much smaller interquartile range.

Page type analysis revealed prebuilt pages generally yielded better results.

Since *DPGuard* provides insight into how real-world webpages can be modified to remove manipulative patterns, we also compared results based on the type of page (real-world vs. prebuilt), shown in Figure 5.4. We found that for a majority of real-world pages, the LLM was not able to make any changes (No-op), or the manipulative elements remained unchanged. More than half of the occurrences were No-ops. It was additionally never able to **fully** remove the manipulative elements. In contrast, prebuilt pages never contained No-op results and generally produced more improvements, with almost half of the runs resulting in a reduction or removal.

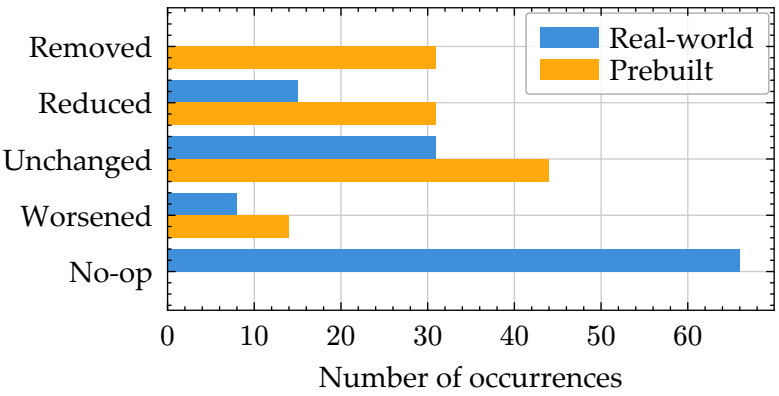


Figure 5.4: A comparison of the evaluation results for real-world and prebuilt pages across different outcome categories. The chart shows that prebuilt pages generally yielded better results, while real-world pages had a higher number of No-ops and fewer removals of deceptive patterns.

Overall, around 47.12% of all runs produced a removal or reduction of manipulative elements.

In total, around 47.12% of all runs produced a removal or reduction of the manipulative elements. For prebuilt pages, more than half (51.66%) of the runs produced such an improvement, while for real-world pages, this was only 12.5%. This indicates that while the tool is effective on simpler, prebuilt pages, it struggles with the complexity of real-world webpages, since real-world webpages often contain more complex structures, sometimes generated when using a framework, making it harder for the LLM to identify and modify the relevant elements. This suggests that further

improvements are needed, for example, by providing a more sophisticated chunking/ignoring algorithm or by using a more powerful LLM.

Smaller effective sizes led to better results.

Results have additionally shown that a higher effective size generally leads to No-op or unchanged results, probably due to the LLM being overwhelmed by the amount of content it had to process. This further supports the need for a more sophisticated chunking and ignoring algorithm.

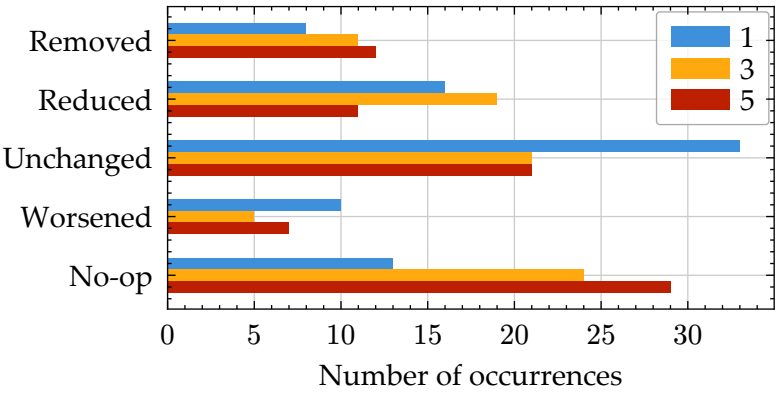


Figure 5.5: The evaluation results based on the number of iterations (1, 3, and 5). The chart highlights 3 iterations as the most effective setting, producing the highest number of removals and reductions while minimizing No-ops and worsened outcomes.

Iteration analysis identified 3 iterations as the most effective setting.

Similar to Schäfer et al. [2025], we also evaluated the impact of different LLM iterations (introduced in Section 4.2), shown in Figure 5.5. Aligning with their work, we found that 3 iterations produced the best results, yielding the highest number of removals and reductions, while minimizing unchanged and worsened outcomes. However, we also found that the more iterations were used, the more No-op results were produced. This is likely due to the LLM reaching its token limit or context window, especially for the local LLM. This suggests that while multiple iterations can be beneficial, there is a trade-off between the number of iterations and the likelihood of the LLM being able to process the request successfully.

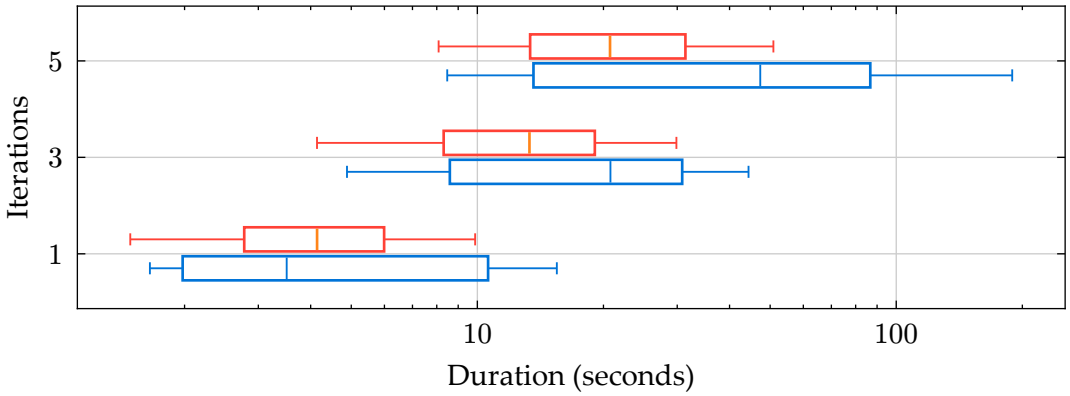


Figure 5.6: A boxplot comparison of the evaluation durations (in seconds and with a *logarithmic* scale) with *GPT-4o* based on the page type (lower: real-world, upper: prebuilt) and the number of iterations (1, 3, and 5). The plot indicates that prebuilt pages generally have shorter durations compared to real-world pages across all iteration counts, with real-world pages showing greater variability in duration.

Prebuilt and real-world pages were compared for their durations.

The duration analysis was further extended by comparing the average duration based on the page type (real-world vs. prebuilt) and the number of iterations, shown in Figure 5.6. The results were only gathered for the remote LLM (*GPT-4o*) to allow for a better comparison, since the local LLM's performance is heavily dependent on the hardware it runs on, and we only want to compare the page types here. Additionally, we only considered runs that produced a change (excluding No-op results) to focus on successful modifications.

Prebuilt pages were generally faster to process than real-world pages, which had a much higher spread.

The results indicate that prebuilt pages were generally faster to process than real-world pages, likely due to their simpler structure and smaller size. However, for a single iteration, real-world pages actually had a slightly lower median duration, potentially because the LLM was able to quickly identify and modify the relevant elements without needing to process the entire page, or because the LLM did not actually process the elements thoroughly. As the number of iterations increased, the duration for real-world pages increased more rapidly than for prebuilt pages, possibly due to the increased complexity and size, or the LLM now actually trying to meticulously process the element. For iter-

ations 3 and 5, real-world pages produced increasingly more No-op results, indicating that the LLM struggled to process the content within its token limit. Nevertheless, both page types showed an increase in duration with more iterations, highlighting the trade-off between the number of iterations and processing time.

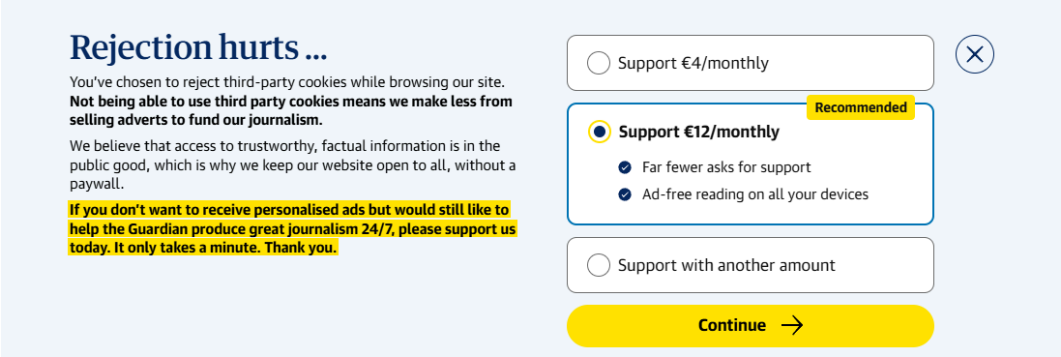


Figure 5.7: A cookie banner from The Guardian[2], a real-world manipulative webpage containing an example of *Confirmshaming* and *Visual Prominence*. The banner pressures users into either accepting personalized ads or giving monetary support through wordings such as "Rejection hurts…".

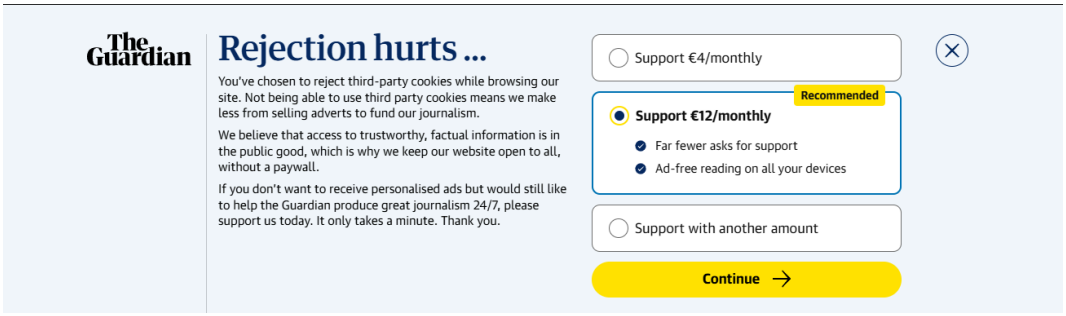


Figure 5.8: The modified cookie banner from Figure 5.7 after applying DPGuard with a local LLM, 1 iteration and intermediate styles. The Visual Prominence was reduced by removing the yellow background of the paragraph, but the *Confirmshaming* was not addressed, as the manipulative heading remained unchanged.

During the evaluation, we also kept a record of interesting cases and observations. For instance, Figure 5.7 and Figure 5.8 show the original and modified versions of a

[2]https://theguardian.com/europe, *Accessed October 2025*

A cookie banner from The Guardian demonstrated both successful and unsuccessful mitigations.

cookie banner from The Guardian[2], respectively. The original banner uses a large heading to make the user feel guilty for rejecting the cookies, implementing both *Confirmshaming* (triggering uncomfortable emotions such as guilt or shame) and *Visual Prominence* (making less relevant parts more prominent) [Gray et al., 2024a]. *DPGuard* was then applied only to the text on the left, since including the buttons on the right would have exceeded the token limit. Nevertheless, the *DPGuard* was able to remove the Visual Prominence by unifying the text style and removing the yellow background. However, the Confirmshaming was not addressed, as the manipulative heading remained unchanged. This indicates that while *DPGuard* can effectively address certain manipulative patterns, it may struggle with more subtle or complex elements, especially when operating under token constraints posed by the LLM.

The LLM introduced manipulative elements in some cases.

Although generally effective, we also observed some unexpected behaviors. For instance, Figure 5.9 and Figure 5.10 show the original and modified version of a prebuilt Fair Pattern website respectively. The original website did not contain any DPs, but after applying *DPGuard*, the LLM created a manipulative element by negating the wording of the checkbox, making users believe they needed to tick the box to **un**subscribe instead of subscribing to the newsletter. This aligns with findings highlighting the risk of LLMs generating misleading or manipulative designs [Chen et al., 2025] and emphasizes the importance of thorough review of LLM-generated content to avoid inadvertently introducing manipulative elements.

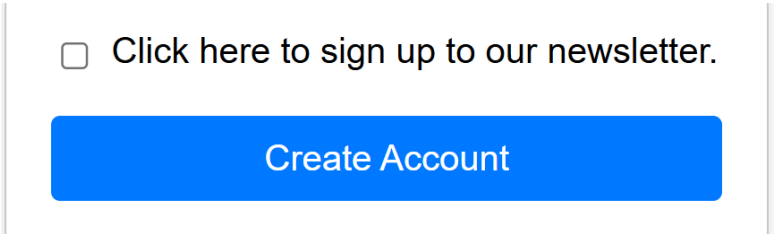


Figure 5.9: The original (prebuilt) Fair Pattern website. The website showcases a checkbox that allows the user to subscribe to a newsletter. The checkbox is not pre-ticked.

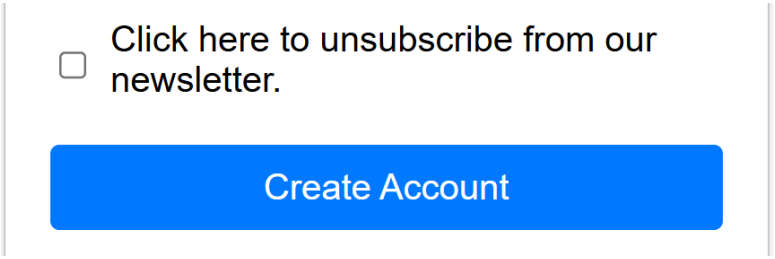


Figure 5.10: The modified Fair Pattern website from Figure 5.9 after mitigating with the local LLM, 1 iteration and no additional styles. The LLM negated the wording of the checkbox, making users believe they needed to opt out instead.

The evaluation highlighted both strengths and weaknesses of DPGuard.

Overall, the evaluation demonstrated that *DPGuard* is a promising tool for mitigating deceptive patterns on webpages. However, the results also highlighted several areas for improvement, particularly in handling the complexity of real-world webpages and ensuring that manipulative elements are effectively addressed without introducing new issues. Furthermore, the choice of LLM and configuration parameters had a great impact on the effectiveness of the tool. When using local LLMs, it is crucial to consider their limitations, such as token limits and context windows, since, currently, our results show that their usage may not be sufficient for realistic and reliable in situ mitigations.

Chapter 6

Discussion

DPGuard extends existing research on technical interventions for deceptive patterns.

Bongard-Blanchy et al. [2021] defined the category of technical intervention as an effective countermeasure against deceptive patterns. Examples of such interventions include *Insite* or the *DAPDE* project. By implementing *DPGuard*, we contribute to this line of research. Specifically, our extension can be assigned to the category of technical interventions, with a focus on realizing the scope of *elimination* [Bongard-Blanchy et al., 2021]. Since our approach leverages LLMs to identify and mitigate deceptive patterns, the natures of the intervention and its scope are not determined beforehand. The resulting visual countermeasure [Schäfer et al., 2024] aligns more closely with *design* intervention methods. For all of the applied mitigations, we ensured that the original content remains accessible to users by implementing the *SW* countermeasure by Schäfer et al. [2023]. This way, users have the ability to toggle between the original and modified content, allowing them to make informed decisions based on their preferences. This intervention falls within the scope of *elimination* and *awareness*, enhancing the user's ability to recognize and understand this type of deceptive pattern in the future.

DPGuard extends existing research by using LLMs.

Furthermore, our work extends the concept of technical interventions by incorporating *Large Language Models* (LLMs) for the detection and mitigation of deceptive patterns. This supplements existing research that has explored the use of LLMs for detecting deceptive patterns, most notably the work of Schäfer et al. [2025], by introducing a practical realization in the form of a browser extension.

DPGuard provides a flexible and extensible platform for future research.

Considering the current state of research outlined in Chapter 2, our approach aims to provide an interface for the HCI community to experiment with LLMs for mitigating deceptive patterns. By offering a flexible and extensible platform, we enable researchers and practitioners to explore various strategies for countering deceptive patterns, potentially leading to more effective and user-friendly solutions in the future, especially considering the rapid advancements in LLM capabilities[1].

DPGuard focuses on direct mitigation rather than detection and is not limited to specific DPs.

Relating our work to existing projects, *DPGuard* distinguishes itself by focusing on the direct mitigation of deceptive patterns using LLMs, rather than solely detecting or highlighting them. While tools like *Insite*[2] and *DAPDE*[3] primarily aim to identify and visually indicate the presence of deceptive patterns, our approach goes a step further by actively modifying the webpage content to reduce manipulativeness. Additionally, our approach does not suffer from the limitation of only being able to detect a predefined set of deceptive patterns, as our use of LLMs allows for a more adaptive approach to mitigation. The more flexible nature of *DPGuard* could possibly improve upon the effectiveness of deceptive pattern countermeasures when compared to other tools, as it is not limited to a fixed set of known patterns. However, this flexibility also introduces challenges, such as ensuring that the LLM-generated mitigations are appropriate and do not inadvertently introduce new usability issues.

[1] https://artificialanalysis.ai/leaderboards/models, *Accessed October 2025*

[2] https://github.com/NicholasTung/dark-patterns-recognition, *Accessed August 2025*

[3] https://github.com/Dapde/Pattern-Highlighter, *Accessed August 2025*

While *DPGuard* is more flexible in its application scope, it also faces challenges that are not present in other tools. For instance, the reliance on LLMs introduces variability in the quality of mitigations, depending on the model used and its training data. Furthermore, the modification of elements directly within the DOM can lead to unforeseen interactions with the webpage's functionality, potentially disrupting the user experience. Overall, *DPGuard* complements the existing scenery of tools by providing a promising and novel approach to deceptive pattern mitigation, addressing some of the limitations of prior work while also introducing new challenges that warrant further investigation.

DPGuard complements existing tools while introducing new challenges.

Limitations and Future Work

During the development of *DPGuard* and especially while conducting technical evaluations, we noticed some limitations of our approach that could be addressed in future work. Specifically, we identified three main areas for improvement: Handling dynamic webpage content, improving the *Evaluator*'s and *CSS* parsing capabilities, and enhancing the user interaction. Additionally, we consider the LLM to be the most notable bottleneck of our approach, although this is not a limitation of our implementation, but rather a general challenge when working with LLMs [Naveed et al., 2025]

Several limitations of DPGuard were identified during development and evaluation.

In its current state, *DPGuard* is primarily designed to handle static webpage content. However, many modern webpages feature dynamic content that can change frequently, such as animations or counters. This poses a challenge, as we are only able to take a snapshot of the webpage at a specific point in time. If the content changes after the snapshot is taken, for example, for a counter, implementing *Fake Urgency* [Gray et al., 2024a], the LLM might not be able to make appropriate modifications. Additionally, DPs that exist across different pages can currently not be mitigated. A potential solution to this problem could involve implementing a more sophisticated mechanism for tracking changes, for example, by

Handling dynamic webpage content is a challenge for DPGuard.

periodically re-evaluating the webpage content or capturing more than one snapshot over time, similar to what is done in the *DAPDE* browser extension. This would allow the extension to adapt to changes in the webpage content and provide more accurate mitigations.

Some webpages do not render their content in a traditional DOM.

Additionally, during our evaluation, we observed that a few webpages did not render their pages in a traditional DOM, but instead used a canvas element to render the entire page. This approach makes it impossible for *DPGuard* to access and modify individual elements, as the content is not represented in the DOM. Addressing this limitation would require a different approach, potentially involving image recognition techniques, similar to how Chen et al. [2023] use computer vision to detect deceptive patterns in mobile applications.

The evaluator currently lacks a sophisticated implementation.

For the *Evaluator*, we identified several areas for improvement. First, the current implementation only includes a basic algorithm for ignoring certain elements, which comes with the drawback of including irrelevant elements in the LLM processing, making it harder for the LLM to focus on the relevant parts. A more sophisticated algorithm could be developed, calculating the actual token count of the elements to be sent to the LLM, and being context-aware, for example, by ignoring elements that are not visible to the user. Furthermore, the *Evaluator* currently does not consider the size of the elements when deciding which elements to send to the LLM. This can lead to situations where very large elements are sent, exceeding the token limit of the LLM.

CSS parsing could be improved to better preserve visual appearance.

Moreover, our current implementation for processing *CSS* styles is limited, as we only support inline styles and styles defined in the document's `<head>`. Technically, we also support computed styles, but this often leads to very large elements being sent to the LLM, which can exceed token limits and impact performance. A more sophisticated approach could involve selectively including only the most relevant styles, for example, by analyzing which styles are actually applied to the element and its children, and excluding any unused styles. This would help reduce the size of the

elements being sent to the LLM, while still preserving their visual appearance.

Having the user select elements requires technical understanding and manual effort.

Following, we identified some areas for improvement in the user interaction. While the current implementation allows users to select elements on the webpage and apply mitigations, it still requires a certain level of technical understanding and manual effort. Future work could explore ways of automatically applying mitigations without user intervention. However, this would require implementing a chunking algorithm similar to *Block-o-Matic* [Sanoja and Gançarski, 2014], since sending the entire webpage to the LLM at once is often not feasible due to token and performance limitations. Additionally, the user interface does not account for moving elements, such as popups or modals, which can make it difficult for users to select the desired elements. Further complementing the research by Schäfer et al. [2023], future work could expand the range of visual countermeasures available for history hover, such as adding *HL+E* (Highlight + Explain).

Future work could address LLM limitations and expand model support.

Lastly, while we addressed privacy concerns by running LLMs locally, this approach comes with its own set of challenges. Local LLMs often have limitations in terms of performance and capabilities compared to cloud-based models. Future work could explore ways to optimize the performance of local LLMs or investigate hybrid approaches that balance privacy and performance, for example, by using local models for initial processing and cloud-based models for more complex tasks. Since *DPGuard* provides a common interface for different LLM providers, expanding the range of supported models could also be a potential area for future work, especially for researching the effectiveness of different models for mitigating deceptive patterns. Although our current implementation only allows for new LLMs to be integrated by modifying the source code, we believe that the differences in their interfaces are minimal enough to allow for a more user-friendly integration process in future iterations.

Chapter 7

Conclusion

The goal of this thesis was to explore the practical feasibility of using LLMs to mitigate deceptive patterns on real-world web interfaces. To achieve this, we developed *DPGuard*, a browser extension that implements the approach proposed by Schäfer et al. [2025] and allows users to select elements and apply LLM-based mitigations in situ. We presented the design and architecture of the extension, detailing its core components, including the *Node Hierarchy* and the *LLM Client* interface. We also described the technical implementation of the extension, highlighting key features such as the *Evaluator*, and the communication system between all of the extension's components. For evaluation, we tested *DPGuard* on a set of webpages while changing several parameters. We found that generally, the extension was able to mitigate deceptive patterns, even though the application on real-world webpages proved to only provide moderate effectiveness, with the LLM being the bottleneck. We acquired promising ideas for future work, especially for allowing effective mitigations on real-world webpages. Summarizing, while *DPGuard* demonstrates the potential of LLM-based mitigations for deceptive patterns, it also highlights the challenges and limitations of applying this approach in practice, requiring further research and development.

Appendix A

Prompt

```
Make the following html page less manipulative. Adhere to the following 12
rules from now on :
1. If there is no manipulation, then do not change anything.
2. Never remove any actions like buttons or links
3. Never make buttons look inactive or grayed out if they can be clicked.
4. Never add any new information to the page that was not present before.
5. If two buttons are on the same hierarchical level, make both the same
design.
6. Never add new functionalities to the site.
7. Never change facts.
8. Never change numbers.
9. Never hallucinate or invent new information.
10. Never invert the meaning of a statement.
11. Never include the input html code as a part of your response.
12. Provide a list with the changes you performed and why those changes
were necessary.
13. Keep all comments and attributes starting with "leave-" untouched.
14. Provide a short (2 to 4 words) summary of the changes made (or null).
15. Provide a reason why the changes were necessary (or null).
```

Listing A.1: The prompt that was used to instruct the LLM to modify deceptive webpages. It includes a set of 15 rules that must be followed when modifying the page. The original prompt was taken from Schäfer et al. [2025] and adapted to the needs of DPGuard. Note that the prompt is formatted for better readability.

Appendix B

Evaluation Webpages

On the following pages, you can find all the webpages used for the evaluation of DPGuard.

URL	Pattern	Element
https://www.opodo.de/	False Hierarchy	List of "Prime Flight Deals"
https://www.ryanair.com/de/de/lp/entdecken/mitgliedschaft	Positive Framing	Yellow banner
https://www.adobe.com/creativecloud/plans.html	Bundling	Creative Cloud Pro
https://amzn.eu/d/6S0ZFNn	High Demand	Limited time offer banner
https://www.nytimes.com/subscription	Complex Language	Payment text
https://en.wikipedia.org/wiki/Human-computer_interaction	None	First paragraph
https://help.disneyplus.com/en-GB/article/disneyplus-en-de-price	Comparison Prevention	Pricing table
https://www.google.com/search?q=ferienwohnung	Disguised Ad	Top result
https://www.theguardian.com/europe	Confirmshaming	Cookie banner
https://www.t-mobile.com/cell-phone-plans	Comparison Prevention	Leftmost option

Table B.1: All real-world websites that were used in the evaluation of DPGuard. The table lists the URL, the deceptive pattern that is present on the page, and the element that was tested. The access date for all webpages was October 03, 2025. Prebuilt websites were taken from Schäfer et al. [2025] and are not listed here.

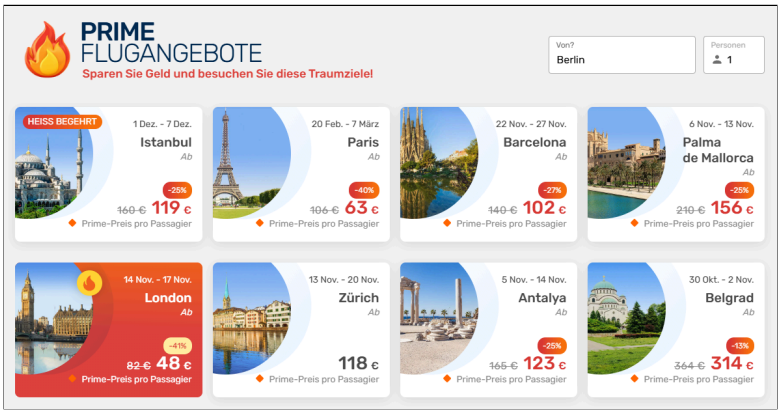


Figure B.2: Opodo, False Hierarchy, List of "Prime Flight Deals"

Figure B.3: Ryanair, Positive Framing, Yellow banner

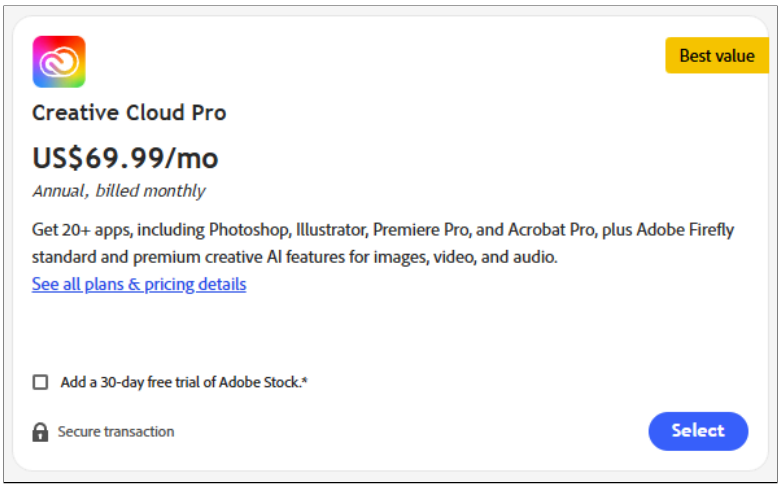


Figure B.4: Adobe, Bundling, Creative Cloud Pro

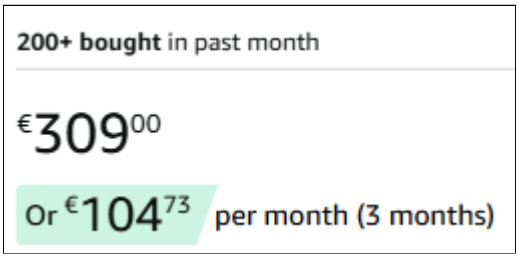


Figure B.5: Amazon, High Demand, Limited time offer banner

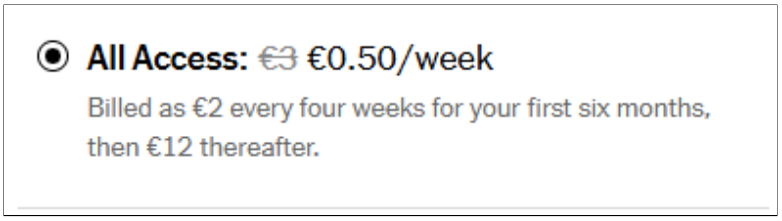


Figure B.6: New York Times, Complex Language, Payment text

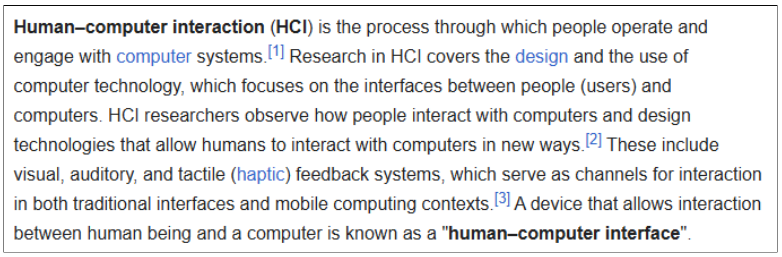


Figure B.7: Wikipedia, None, First paragraph

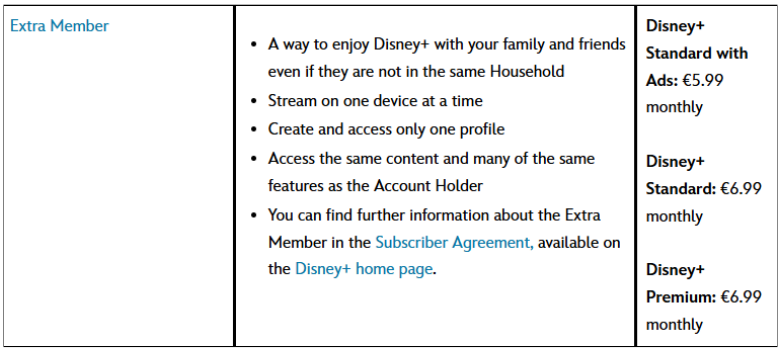


Figure B.8: Disney Plus, Comparison Prevention, Pricing table

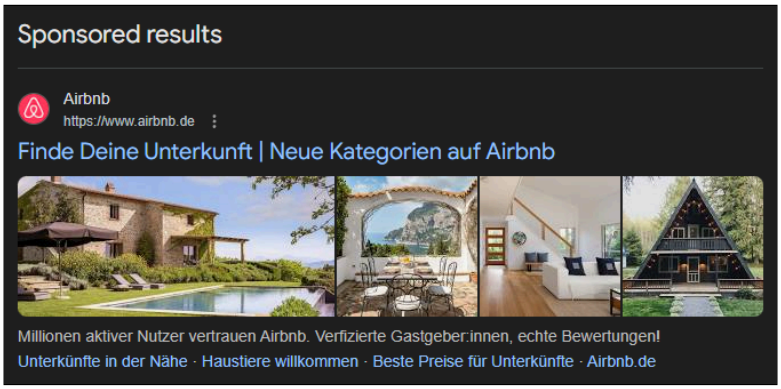


Figure B.9: Google, Disguised Ad, Top result

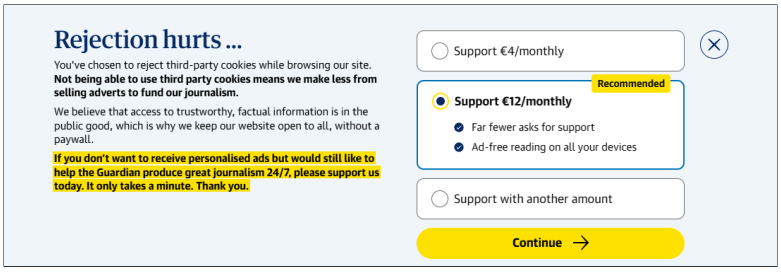


Figure B.10: The Guardian, Confirmshaming, Cookie banner

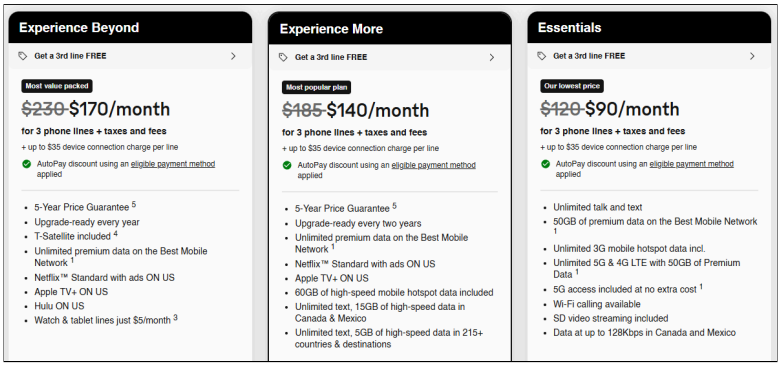


Figure B.11: T-Mobile, Comparison Prevention, Leftmost option

Bibliography

[1] Kerstin Bongard-Blanchy, Arianna Rossi, Salvador Rivas, Sophie Doublet, Vincent Koenig, and Gabriele Lenzini. "I am Definitely Manipulated, Even When I am Aware of it. It's Ridiculous!" - Dark Patterns from the End-User Perspective. pages 763–776, 2021, doi.org/10.1145/3461778.3462086.

[2] Christoph Bösch, Benjamin Erb, Frank Kargl, Henning Kopp, and Stefan Pfattheicher. Tales from the Dark Side: Privacy Dark Strategies and Privacy Dark Patterns. *Proceedings on Privacy Enhancing Technologies*. 2016:237–254, 2016. doi.org/10.1515/popets-2016-0038.

[3] Jieshan Chen, Jiamou Sun, Sidong Feng, Zhenchang Xing, Qinghua Lu, Xiwei Xu, and Chunyang Chen. Unveiling the Tricks: Automated Detection of Dark Patterns in Mobile Applications. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, San Francisco, CA, USA, 2023, doi.org/10.1145/3586183.3606783.

[4] Ziwei Chen, Jiawen Shen, Luna, and Kristen Vaccaro. Hidden Darkness in LLM-Generated Designs: Exploring Dark Patterns in Ecommerce Web Components Generated by LLMs. *https://arxiv.org/abs/2502.13499*, 2025.

[5] Gregory Conti and Edward Sobiesk. Malicious interface design: exploiting the user. *Proceedings of the 19th International Conference on World Wide Web*, pages 271–280, Raleigh, North Carolina, USA, 2010, doi.org/10.1145/1772690.1772719.

[6] Colin M. Gray, Johanna T. Gunawan, René Schäfer, Nataliia Bielova, Lorena Sanchez Chamorro, Katie Seaborn, Thomas Mildner, and Hauke Sandhaus. Mobilizing Research and Regulatory Action on Dark Patterns and Deceptive Design Practices. *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, Honolulu, HI, USA, 2024b, doi.org/10.1145/3613905.3636310.

[7] Colin M. Gray, Yubo Kou, Bryan Battles, Joseph Hoggatt, and Austin L. Toombs. The Dark (Patterns) Side of UX Design. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–14, Montreal QC, Canada, 2018, doi.org/10.1145/3173574.3174108.

[8] Colin M. Gray, Cristiana Teixeira Santos, Nataliia Bielova, and Thomas Mildner. An Ontology of Dark Patterns Knowledge: Foundations, Definitions, and a Pathway for Shared Knowledge-Building. *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, Honolulu, HI, USA, 2024a, doi.org/10.1145/3613904.3642436.

[9] Aditi M. Bhoot, Mayuri A. Shinde, and Wricha P. Mishra. Towards the Identification of Dark Patterns: An Analysis Based on End-User Reactions. *Proceedings of the 11th Indian Conference on Human-Computer Interaction*, pages 24–33, Online, India, 2021, doi.org/10.1145/3429290.3429293.

[10] Arunesh Mathur, Gunes Acar, Michael J. Friedman, Eli Lucherini, Jonathan Mayer, Marshini Chetty, and Arvind Narayanan. Dark Patterns at Scale: Findings from a Crawl of 11K Shopping Websites. *Proc. ACM Hum.-Comput. Interact*. 3(CSCW), 2019. doi.org/10.1145/3359183.

[11] Stuart Mills and Richard Whittle. Detecting Dark Patterns Using Generative AI: Some Preliminary Results. *https://ssrn.com/abstract=4614907*, 2023.

[12] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A Comprehensive Overview of Large Language Models. *ACM Trans. Intell. Syst. Technol*. 16(5), 2025. doi.org/10.1145/3744746.

[13] Asmit Nayak, Shirley Zhang, Yash Wani, Rishabh Khandelwal, and Kassem Fawaz. Automatically Detecting Online Deceptive Patterns in Real-time. *https://arxiv.org/abs/2411.07441*, 2024.

[14] Lorenzo Porcelli, Michele Mastroianni, Massimo Ficco, and Francesco Palmieri. A User-Centered Privacy Policy Management System for Automatic Consent on Cookie Banners. *Computers*. 13(2), 2024. doi.org/10.3390/computers13020043.

[15] Andrés Sanoja and Stéphane Gançarski. Block-o-Matic: A web page segmentation framework. *2014 International Conference on Multimedia Computing and Systems (ICMCS)*, pages 595–600, 2014, doi.org/10.1109/ICMCS.2014.6911249.

[16] Yasin Sazid, Mridha Md. Nafis Fuad, and Kazi Sakib. Automated Detection of Dark Patterns Using In-Context Learning Capabilities of GPT-3. *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 569–573, 2023, doi.org/10.1109/APSEC60848.2023.00072.

[17] René Schäfer, Paul Miles Preuschoff, and Jan Borchers. Investigating Visual Countermeasures Against Dark Patterns in User Interfaces. *Proceedings of Mensch Und Computer 2023*, pages 161–172, Rapperswil, Switzerland, 2023, doi.org/10.1145/3603555.3603563.

[18] René Schäfer, Paul Miles Preuschoff, René Röpke, Sarah Sahabi, and Jan Borchers. Fighting Malicious Designs: Towards Visual Countermeasures Against Dark Patterns. *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, Honolulu, HI, USA, 2024, doi.org/10.1145/3613904.3642661.

[19] René Schäfer, Paul Preuschoff, Rene Niewianda, Sophie Hahn, Kevin Fiedler, and Jan Borchers. Don't Detect, Just Correct: Can LLMs Defuse Deceptive Patterns Directly?. *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, page 11, 2025.

Index