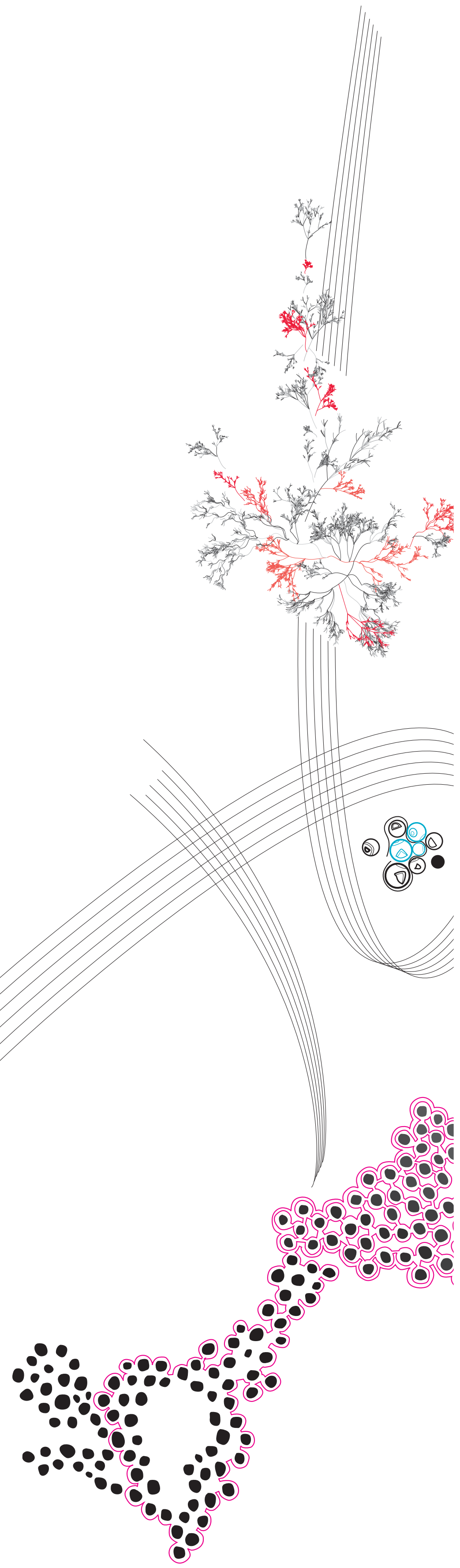
MSc Computer Science
Final Project



NAISS: Network Authentication of Images to Stop e-Skimmers

Cătălin Rus

Supervisor: Dr. Dipti Sarmah, Dr. ing. Mohammed El-Hajj and Prof. Dr. ing. Roland Martijn van Rijswijk-Deij

May , 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

The rise of payment details theft has led to increasing concerns regarding the security of e-commerce platforms. For the MageCart threat family, the attacks employ e-skimmers, which are pieces of software code that instruct clients to forward payment details to an attacker-controlled server. They can be injected into hosting providers' servers as HTML tags such as *script*, *iframe*, and *img*. By leveraging image steganography - the technique of hiding structured information inside images without visual perturbances - MageCart groups can deliver e-skimmers without raising any suspicion. In this report, we systematically review applicable solutions in the literature and evaluate their drawbacks in the setting of a compromised hosting provider. While promising, existing solutions in the literature present shortcomings such as lack of compatibility, adoptability or functionality under the presence of an attacker. Based on this review, we compile a set of features for a better solution, which we use as a foundation for designing our proposed solution - *NAISS: Network Authentication of Images to Stop e-Skimmers*. Through our solution, digital signatures of individual images are checked inside a server-side middlebox residing in the hosting provider's network to prevent the transmission of unauthorized images to clients. The signatures are provided by the e-commerce platform developer prior to uploading a website to the hosting provider. Our proof-of-concept implementation shows that *NAISS* is capable of filtering 100% of present stegoimages, regardless of their novelty, while imposing a minimal performance detriment and no client-side modifications. All of the source code material of this project has been made publicly available on `github.com/ruscatalin/NAISS`.

Keywords: Image steganography, E-skimmers, MageCart, Middlebox, Digital signatures, Network filter, E-commerce

Chapter 1

Introduction

In this report, we put forward a proposed solution for improving the authenticity of images found on e-commerce platforms. We create this solution after a thorough investigation of proposed solutions together with their strong and weak points, such that gaps are filled and improvements are made upon the strong points. We evaluate our solution using automated testing on a dummy website, under varied circumstances. The following is an introduction to the research challenge and a description of the research objectives, scope and contribution brought by our work.

E-commerce is a continuously growing market specialised in selling goods and services through the internet and is quickly becoming the preferred method of purchasing, especially through periods where customers do not leave their homes such as the recent COVID-19 global pandemic [98]. An increasing number of businesses desire and enable e-commerce capabilities to increase sales [14] and customers are opting to make online purchases more often each year. As the driving factor of e-commerce platforms is the increase in sales and reduced operational costs, technical security measures are often overlooked, especially for those who outsource their platforms to web hosting providers [98]. The hosting providers' main selling point is to ensure the availability of the stored data [11], and sadly security is an area where they fail to invest [23], leaving hosted e-commerce platforms vulnerable to exploits.

One type of dangerous exploit for e-commerce platforms is represented by the stealing of payment card details from customers. MageCart is a threat family that specializes in the theft of customer payment credentials via the use of electronic skimmers (e-skimmers) [29], [84]. These e-skimmers are typically injected into the stored data of the hosting providers through methods such as Cross-Site-Scripting (XSS) [19], yet more injection procedures are being steadily developed [64]. The prevalence of MageCart attacks has been increasing in recent years [31], [40], with numerous high-profile attacks on major companies, such as British Airways [34]. As e-commerce continues to expand its reach, the number of victims is likely to increase unless effective countermeasures are implemented [45].

As a response, the Federal Bureau of Investigation (FBI) of the United States of America has issued a warning for the rising threat posed by MageCart and the Payment Card Industry Data Security Standard has been revised after a prolonged period of stagnation [56]. With over 4,800 publicly reported attacks in 2020 alone [40], the proliferation of MageCart represents a significant threat to the trust and profitability of payment platforms. In order to understand how to protect against MageCart attacks, it is first crucial to understand the nuances of e-skimmers hiding within images.

E-skimmers deployed by MageCart are code snippets that instruct clients to forward their entered payment details to a server controlled by the attackers [47]. E-skimmers can be hidden inside various HTML tags, including *script*, *iframe*, *form*, *table*, and more others[1], but also inside Exchangeable Image File Format (EXIF) metadata [51], favicons [41] or, loaded dynamically by other pieces of code [89] embedded into the web pages. However, as these code snippets can be easily read through visual code inspection, some MageCart groups have turned to image steganography [50] to obfuscate their code from security testing tools and wary clients. Image steganography, as detailed in Section 1.1, is a technique of hiding information within images without considerably altering their appearance. The resulting images, henceforth referred to as stegoimages, present no obvious visual artifacts that would indicate tampering, yet once decoded by the clients' browsers, the embedded code would be executed.
By embedding their e-skimmers inside stegoimages, these MageCart groups have gained an average of three years before their state-of-the-art image steganography techniques can be detected [67]. Unfortunately, researchers have yet to create a practical "Universal Detector" for stegoimages [67], [18], [26] and, as a result, it is not recommended that the prevention of stegoimage e-skimmers rely solely on detection [96]. Hence, more effective solutions are required to address the stegoimage e-skimmers challenge.

1.1 Terms background

For improved clarity on this topic for non-technical readers, we define the key terms necessary to grasp the discussions and importance of our work.

- *steganography* is a procedure of hiding information in plain sight [50]. The information can be hidden in various media such as text and sound. Historical examples of steganography are long pieces of text where a meaningful sentence can be composed from the first letters of each word and such will be hidden by readers not looking to decipher any hidden meaning.

- *image steganography* is the steganography technique that involves hiding information within images. For digital images, additional bits are added such that the visual impact on the image is minimal - no significant pixel colour changes. Additionally, information can also be hidden by overwriting inside the metadata of an image, which is a set of data pertaining to the creation, characteristics and context of a file.

- *stegoimages* are the result of image steganography, where they are relatively indistinguishable visually from their original images. These images typically contain more information and therefore occupy more storage space.

- *steganalysis* is the process of detecting and extracting the hidden information created by a steganography technique. The extraction step does not always yield the hidden information in full form, especially for images [67].

- *code injection* is the process of moving a piece of software or instruction within another piece of software or computer system without the consent or permission of

[1]object, form, script, embed, ilayer, layer, style, applet, meta, img, frame, iframe, frameset, script, table, base

the receiving end. This is typically performed for malicious purposes, for example forcefully instructing a database to list all its stored information.

- *e-skimmers* are pieces of software that effectively steal payment credentials from online customers. The name originates from the card skimmers that can be attached to Automated Teller Machines (ATMs) and read the magnetic strip from payment cards. The "e" part of the name refers to "electronic", as in a skimmer that exists solely in the digital domain.
- *digital signatures* are unique bit sequences that are mathematically produced from an input (e.g., an image) and a key. The uniqueness of the resulting sequence is strictly influenced by both the input and the key, such that even the most minimal change to any of them will result in a uniquely different sequence.

1.2 Research Objective and Questions

The overarching research objective of this paper is to find a practical and efficient way to prevent stegoimage e-skimmers. To this extent, we divide this objective into the following research questions:

RQ1 What does the literature present as solutions against e-skimmers?

RQ2 How do we fill in the gaps identified in the literature?

RQ3 How does our solution compare with other solutions identified in the literature?

The first research question will be answered by performing a systematic literature review in Chapter 2. The second research question is partially answered in Subsection 2.2.7 by referring to an ideal solution, while in Subsection 3.6.2 we showcase how our proposed solution answers this question. A more in-detail view of the comparison of our solution with other solutions in the literature is presented in Subsection 3.6.1, where performance, behaviour, ease of adoption and robustness are used as evaluation criteria. By fulfilling the research objectives in this way, we bring forward our contribution to the fight against MageCart attacks.

1.3 Research scope

To better contextualise the contribution of our work, we need to draw the lines that define the capabilities of this research. Namely, our proposed solution is tested against stegoimage e-skimmers only, with a simplistic and possibly unrealistic setup that does not include mobile browser usage. We do not systematically evaluate the security posture of the proposed solution and solutions found in the literature, but we offer an opinion on that matter. We categorise solutions in the literature based on a subjective understanding of where their working mechanisms operate in the attack chain. For each category, general strong and weak points are compiled, yet for each solution in particular, we mainly comment on the weak points, so as to help create a well-documented recommendation list that addresses drawbacks in the literature. Finally, when performing the literature review, we attempted to explore the boundaries, yet this goal might have been incomplete due to methodology shortcomings. All in all, our research scope is exploratory and simplistic, while aiming to contribute maximally to the field of e-commerce security.

1.4 Contribution

We can summarize the contribution of this work in three main points:

- We perform a comprehensive literature review on the topics related to the MageCart threat, e-commerce implications and related solutions.
- We provide a list of seven important features for a solution against MageCart.
- We propose and implement a solution based on the list of seven features and outline the methodology for developing a proof-of-concept for *NAISS*. Although restricted to images, *NAISS* can be feasibly extended to include any MageCart attack vector.
- We provide a benchmark comparing *NAISS* with related solutions based on performance, robustness and behaviour.

Moreover, for both the literature review and our proposed solution, we identify and suggest future directions of improvement and research based on the identified limitations and assumptions.

The remainder of this report is structured as follows: in Section 2.1 we showcase the design and implementation of a semi-systematic literature review; and in Section 2.2, we analyse the identified literature to finally compile a list of recommendations in Subsection 2.2.7; Chapter 3 details all aspects relating to our proposed solution such as the proof of concept methodology in Section 3.4, the testing procedure along with accompanying results and experiments in Section 3.5 and the significance of those results in Section 3.6. We summarize the contribution and relevance of our study in Chapter 4 as well as future research and development directions.

Chapter 2

Literature Review

2.1 Designing the Systematic Literature Review

When conducting a literature review, one has many methodology options for doing so [39]. The chosen type is heavily influenced by aspects such as the scope and goals of the study, the study discipline and even available time. Each type of review balances out with respect to these aspects, exchanging a characteristic such as time for another (e.g. the extensiveness of explored literature).
For instance, an *Umbrella Review* (e.g. [78]) employs a methodology that restricts its scope to analyzing only secondary sources, such as other literature reviews. While this approach allows for a broad and rapid overview of the research problem and proposed interventions, it is constrained in its ability to evaluate individual studies, and consequently, may yield less nuanced and potentially outdated results that do not reflect the current state-of-the-art knowledge in the field.

Conversely, a *Systematic Literature Review* is not restricted by a fixed timeline and strives to undertake a thorough search of the literature, followed by a meticulous appraisal of each individual study. This approach provides a comprehensive overview of the current state-of-the-art knowledge, identifies gaps in the existing research, clarifies uncertainties in the findings, and suggests practical recommendations for future research and practice.

By conducting a systematic literature review, we aim to structurally collect relevant knowledge, filter and align it such that we obtain an extensive view of the best available information. Furthermore, the synthesis of this information shall yield actionable and practical insights on how to tackle the challenge of MageCart e-skimmers hiding within images. The structured workflow of this type of literature review shall ensure that the scientific standards and procedures are taken to the core of the process.

2.1.1 Methodology

Our Semi-Systematic Literature Review entails several stages (refer to Figure 2.1). Firstly, we identify relevant keywords aligned with **RQ1** to facilitate comprehensive exploration of the topic area. Next, we perform a search using two distinct techniques (i.e., programmatic and manual) utilizing these keywords. Subsequently, we scrutinize and sift through the outputs using filtering protocols to obtain a batch of pertinent and high-quality literature. Lastly, we examine and classify the filtered literature, synthesizing its content to address **RQ2**. The systematic workflow followed in our review assures that the entire process is conducted with precision and rigor.

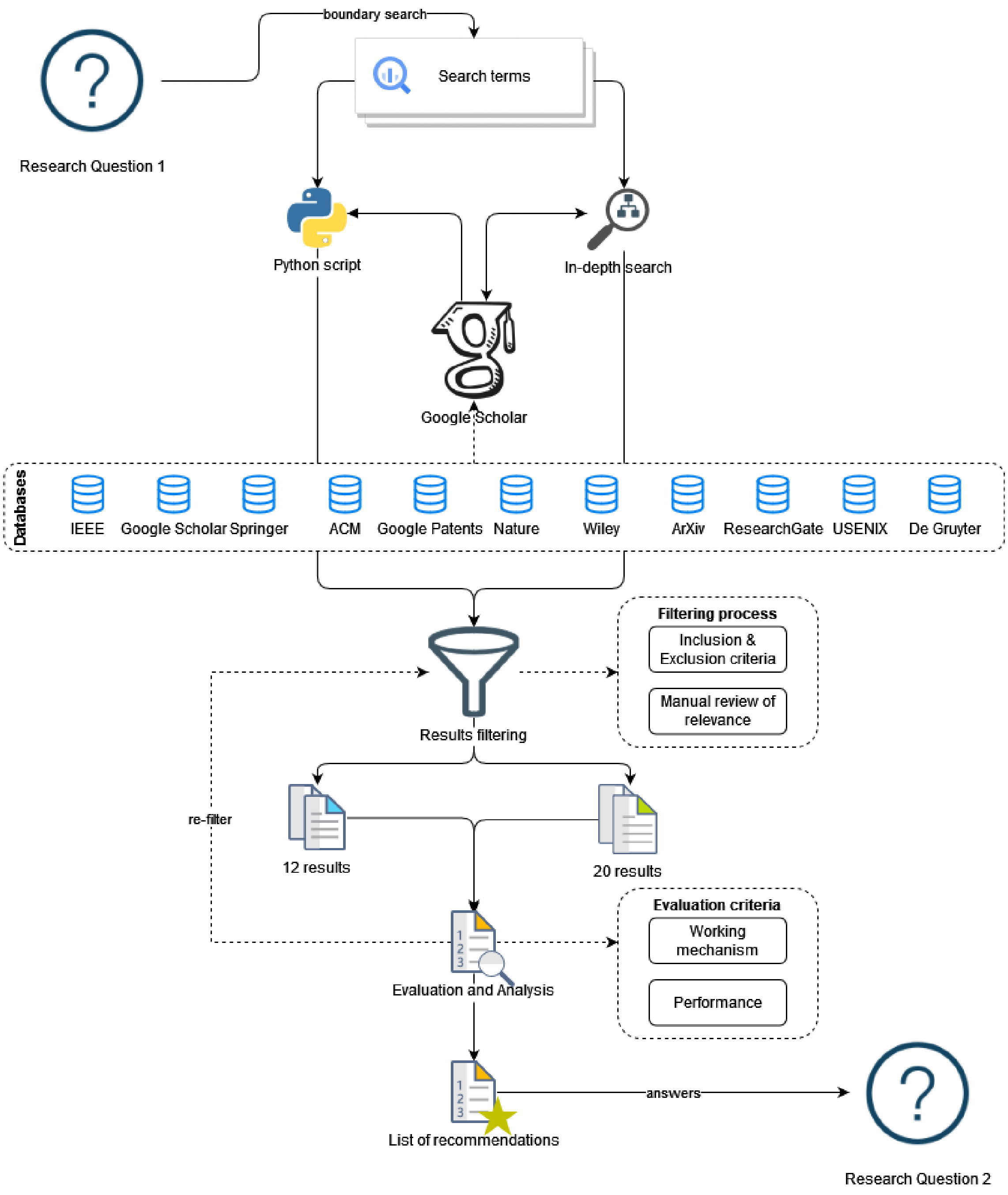


FIGURE 2.1: Overview of the literature review process

Search terms

The selected search terms were chosen to enable us to conduct an exhaustive exploration of the literature, seeking relevant solutions that address the intricate issue of e-skimmers, which is further compounded by the limited research on this specific niche topic, particularly in the context of image steganography. Owing to the multidimensional nature of the problem, potential solutions can be located in several areas of interest (e.g., prevention of code injection, secure transmission sanitization, robust web server protection, and enhanced content integrity). To cover the boundaries of the literature through these areas of interest, we designed our terms to be short and generic in order to generate a large number of hits, but to sometimes also include keywords (e.g., e-commerce, e-skimmer) that might steer the results towards works related to our topic. Therefore, we have employed the following search terms (parentheses indicate interchangeable terms):

- ”blocking image steganography e-commerce"
- ”(malicious/compromised/untrusted) hosting providers e-commerce"
- ”(e-skimmer/magecart) defense e-commerce"
- ”(image/html) authentication"
- ”end to end (integrity/authenticity) e-commerce"
- ”secure web application host"
- ”secure code (distribution/poisoning)"
- ”credential sniffing"
- ”web skimmer filter"
- ”code injection (defense/prevention)"

We believe better keywords can be compiled for this purpose, yet the search hits yielded, combined with deeper searching based on cited and citing works resulted in a satisfactory coverage of the literature boundaries.

Search Databases

In order to systematically retrieve relevant works on the topic, the appropriate research databases need to be queried. To this extent, and to homogenise the queries, the searching is performed solely through the largest database aggregator as of the date of writing, Google Scholar.

The search process involves the use of both programmatic and manual approaches. The distribution of the selected works across the publishing years and databases is depicted in Figure 2.2 and Figure 2.3, respectively. Notably, Figure 2.2 reveals that no works were selected for the time frames of 2013-2015 and 2017-2018, despite yielding results during the initial search. However, the works from these time frames were discarded during the filtering process.

In Figure 2.3 we observe that the databases with the most relevant selected works are IEEE, Google Scholar and ACM. The Google Scholar category here is comprised of works that cannot be found in web sources where querying is available - and so the sole query database to reach those is Google Scholar or other aggregators. Apart from these databases, we see a rather uniform and extended distribution, covering 11 other research and patent databases.

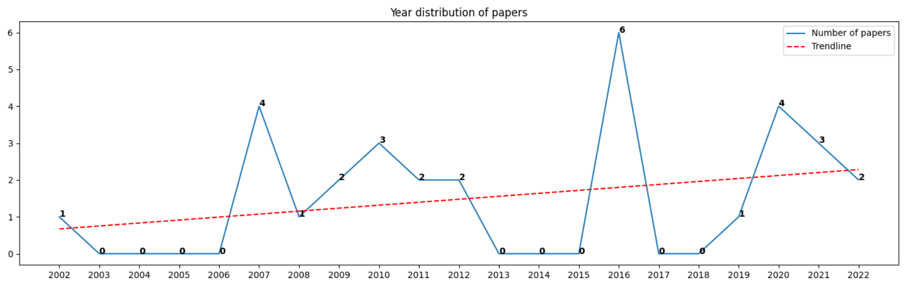


FIGURE 2.2: Year distribution of solutions identified in the literature

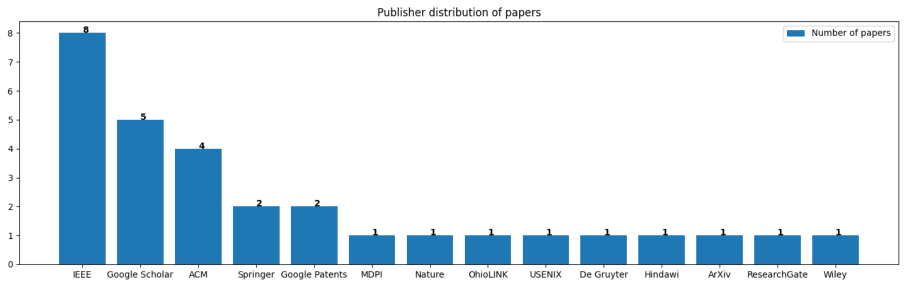


FIGURE 2.3: Distribution of research databases for selected works

Programmatic literature search

We used a Python script[1] to programmatically retrieve the top 20 hits, sorted by relevance. We attempted to further the automatisation process by filtering the results using OpenAI DaVinci [72]: in order to determine the relevance to our research challenge, the title and the abstract snippet of each paper was fed to DaVinci alongside a brief description of the research challenge. We concluded that the abstract snippet (i.e. a small, randomly chosen piece of the abstract) returned by Google Scholar was not enough to provide the right context for obtaining a pertinent answer. We then proceeded to filter the given results manually.

Manual literature search

The manual search was conducted using Google Scholar, which provided the added advantage of enabling a targeted exploration of interesting cited and citing works, thereby facilitating a deeper exploration of the citation chain. Furthermore, this step leveraged the insights obtained from prior searches conducted on the topic of "image steganography", including surveys and analyses of related challenges, which helped to guide the search for relevant literature. In some cases, keyword searching was complemented by the "Cited by" functionality of Google Scholar, which facilitated the identification of relevant papers that had cited a specific article of interest.

Filtering

We used the criteria in Table 2.1 to include or exclude found works. The same criteria were applied to papers that were not found with keyword searching, but through exploring

[1]scholar.py in the literature branch of our repository

references or citing works from specific papers. To manually evaluate the relevance of one work, we mainly investigated the title and abstract, but in some cases, the introduction and conclusion as well. Given that code injection and web-based malware are long-living threats, we considered looking into ideas and solutions that were introduced a long time ago, and which might be of good use in today's e-skimmers landscape. However, as observed in Figure 2.2, the identified solutions are predominantly dated around 2007-2011, but since 2016, there has been an increased interest in addressing the MageCart challenge.

Table 2.1: Inclusion and exclusion criteria for selecting solutions and ideas for Literature Research

Inclusion criteria	**Exclusion criteria**
Posted in journals, conferences, websites or patent databases	Electronically inaccessible
Presents an idea or solution addressing contemporary e-skimmers	The work is not written in (comprehensible) English
Presents an idea or solution that indirectly or partially addresses e-skimmers	The work does not contain (relevant) references
The authors discuss their work critically	The work has a superficial take on the topic OR is less than 4 pages long
The work was published after 2001	The work does not provide additional information on already-read points of interest
	Presented idea or solution is already discussed in a more recent paper

Before performing the analysis of the identified literature, we plot in Figure 2.4 the keyword relationship of the selected works using VOSviewer [91]. The width of the links was set to be proportional to the link strength and as such, we can tell that the keywords "attack", "user", "signature", "content", "integrity" and "server" are the most prominent ones, with "server" being the node with most connections. Moreover, we observe coloured groups which aggregate larger ideas: the red cluster is generally oriented towards describing a solution and its workings, while the blue and green groups generally refer to the issue and methods of MageCart attacks.

2.2 Analysis and categorisation

In this section, we explore the topics related to image steganography used in the context of e-skimmers to gain a grasp of the suitability of proposed solutions. We begin by reviewing the methods of detecting stegoimages and then discuss solutions based on a rough categorisation. A shortened description of found solutions and their drawbacks is given in Table 2.3. Through this incursion into the literature, we expect to gain the necessary insights to compile a list of features for an ideal solution.

We analyse the selected papers by reading the abstract and skimming through the conclusion and results sections to identify the following:

1. The unique solution proposed
2. The working mechanism
3. The performance

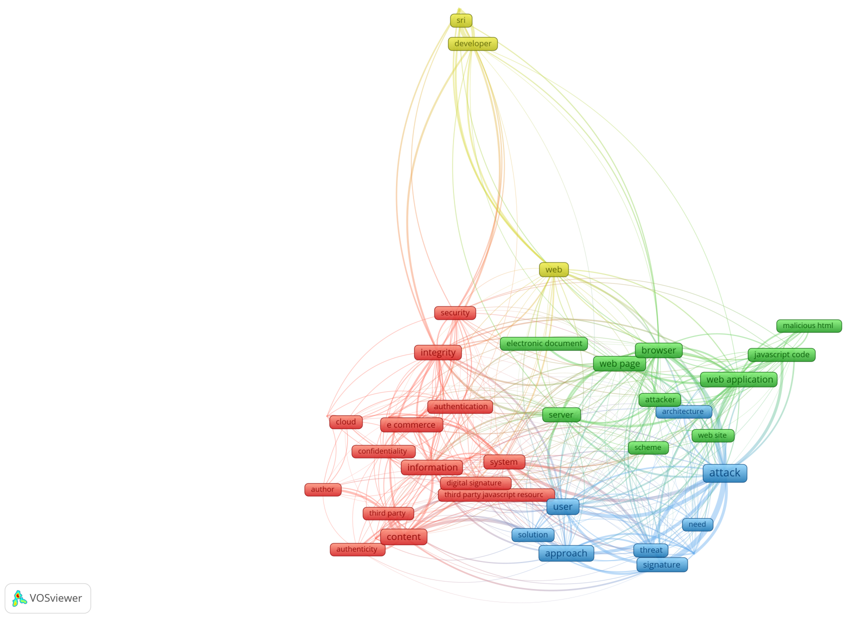


FIGURE 2.4: Keyword correlation of selected papers

If not enough satisfactory information is retrieved through skimming, a more in-depth reading is performed. During this phase, we might discover that a given work does fit the exclusion criteria, in which case said work is removed. The subsequent categorization as seen in Figure 2.2 is based on where the solution is applied on a network topology level (e.g. in the hosting provider's network) and whether they present some outstanding characteristics (e.g. using security policies instead of malware analysis). To this extent, we use the working mechanism to roughly categorise the papers like so:

1. Solutions against stegoimages
2. Protocols and frameworks
3. Server-side solutions
4. Client-side solutions
5. Solutions relying on policy systems
6. Solutions relying on third-party assistance

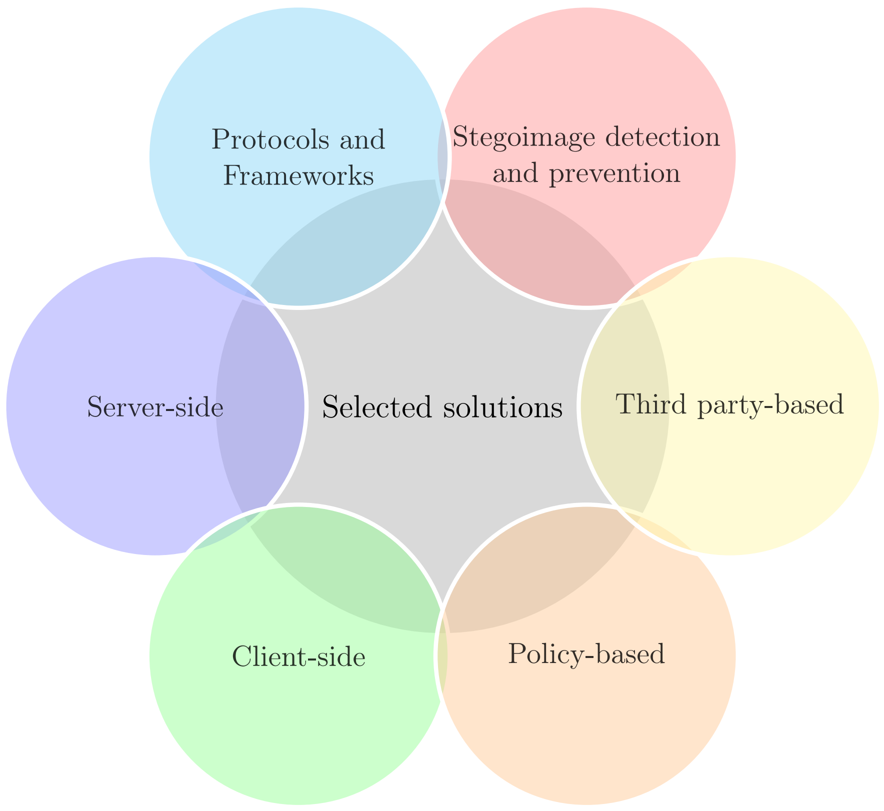


FIGURE 2.5: Proposed categorisation overview

For the readers' convenience, the analysis overview of the unique strengths and weaknesses of each category was compiled in Table 2.2.

We assess the efficacy and usefulness of a solution by contextualising the working mechanism and performance with the challenges of addressing MageCart attacks. Based on this, we identify strong points and weak points, which will be used to compile the list of recommendations.

2.2.1 Stegoimage detection

Through image steganalysis, a defender can take a suspect image and attempt to extract the payload inside it [67]. Steganalysis techniques tend to be a response to a specific steganography technique, but they take on average 3 years to develop and are hard to combine into a one-fits-all solution [67]. Although Deep Learning and Generative Adversarial Networks have shown promising developments in detecting stegoimages, the "Universal Detector" is still far from becoming a reality [67], [18], [26]. Also in [26], the authors propose a framework for analysing suspect files (incl. images) which is primarily comprised of metadata and file content analysis (using stego-toolkit [7], StegSpy [8] and StegExpose [6]) followed by file sandbox analysis (with a Cuckoo [2] environment). In this way, common attacks are blocked and newer ones get statically and dynamically checked. We remind that steganalysis tools are tailored to work only with specific image formats[2]: e.g. StegExpose [6] works with Portable Network Graphics (PNG) and bitmap (BMP) and JRevealPEG [17] works only with Joint Photographic Experts Group (JPEG/JPG).
In their conclusion, the authors of [96] state that to stop steganography-enabled malware,

[2]github.com/DominicBreuker/stego-toolkit#tools-detecting-steganography

mechanisms that do not rely on detection should be used (e.g. Content Threat Removal (CTR) [95]).

2.2.2 Protocols and Frameworks

A systemic approach requires modifications in the typical communication pattern with a client, possibly including modifications on both server-side and client-side. We searched for protocols and frameworks that propose suitable methods of protecting users from e-skimmers or other related risks such as injection. Data integrity was an aspect we focused on due to its critical importance of maintaining the web code outsourced by the e-commerce platform untouched [76].
In [101], Secure E-commerce Transaction (SET) is described to employ signatures of crucial data such as cardholder name, CVV and amount, yet the website initiating SET is not checked for its delivered content. In 2002, security tokens are proposed by [15] and discussed briefly as being useful for authenticating transmitted Simple Object Access Protocol (SOAP) [20] messages, although no protocol or further elaborations of this idea were found among citing papers. Recently, in 2020, a secure payment system model is being proposed by [44], however, it does not consider the risk of a compromised hosting provider serving malicious web pages - again, like SET, the transaction itself is well protected, while the preliminary interaction is not.

WebShield [58] proposes a middlebox (i.e. intercepting traffic) solution that runs fingerprint checking and anomaly detection on behalf of the client. It requires no client-side changes, which is crucial as the issue of client device rigidity and heterogeneity is impacting adoptance of security solutions. WebShield's main limitations are the performance bottleneck for frequently-updating web pages, lack of support for plugins (e.g. the now-unsupported Flash[3]) and incompatibility with HTTPS. One rarely found advantage of WebShield is that it filters malicious parts of the website, as opposed to blocking the whole content.
In 2016, a unified approach is proposed by [97] to collect web page information a priori, perform deep content inspection and generate signatures based on dynamic and static analysis, all within a middlebox. It imposes a delay of 960ms when accessing a web page, but is unable to cover attacks that use HTML tags instead of scripts (e.g. Client Denial of Service (DoS), Auto complete Phishing). Moreover, one more disadvantage is shared with all the other sandbox-typed solutions: steganography-based attacks can escape the sandboxed environment [62].

Verena [52] is a framework built for providing end-to-end integrity for databases in web applications. It consists of a Verena server and a client: the server works with the database and provides correctness proofs for queries, while the Verena client runs inside the client's browsers verifying and filtering data to the client based on those proofs; all this on top of web page integrity verification. The authors mention that using correctly Verena's integrity policies is a challenge for the security of the solution and that further research shall explore easing that. One last notable limitation is that their framework does not provide confidentiality, although they propose fixing it by combining it with one of their works, Mylar [74].

[3]Adobe Flash is a software program that enables a user to view interactive content on a website. Flash content is usually embedded in a web page using the Flash Player plugin.

Cryptographic signatures allow for granular authentication of content along with its delivery to the user. An initial and simplistic idea is to sign the whole HTML file transferred to the client [57]. The authors propose that the watermarking technique is used - so the signature is hidden inside the file - leading to limitations in embedding capabilities and insecure hash values [4] for large HTML files. Moreover, the client would be responsible for evaluating the watermark and, in case of any small malicious change, the whole HTML would be denied, leading to an implicit DoS. We note another similar attempt in a 2007 patent [71]. Although its design is meant for providing businesses with document authentication based on geographic data, this could be useful for securing externally-linked images (e.g. loaded through a Content Delivery Networks (CDN)) and include geographical verifiable information from the publisher side. The geographic aspect is however problematic if taking into mind cloud hosting solutions which might cluster many legitimate businesses and attackers in the same geographic location.

In [73], the author of web content can attach element-specific signatures to each structured element (e.g. HTML *div* tag). The client can then use the Public Key Infrastructure (PKI) to authenticate the origins and integrity of these elements using the public key of the author. The signature is attached at the end of the element, meaning that a malicious payload can be interpreted before the signature is read. An older and simpler alternative based on XML signatures concludes that a more generic version is needed, one which works clearly with typical HTML tags [77]. In a similar vein to [73], in 2010, [48] introduces the process of data colouring and digital watermarking which effectively ties a data resource (i.e. an image in our context) to the data publisher and the right users to access it. The interesting claim is that the computational complexity of this solution is lower than using traditional cryptography and PKI. However, this solution is more addressing the issue of authorisation and authentication for cloud data storage and it does not provide enforcement or filtering for secure data transmission to the client.

To prevent CDNs or other in-transit nodes from manipulating the web content of an application, [59] proposes that each resource be signed by the author. A browser extension will then automatically check the found signatures against the author's public key retrieved from a Domain Name Server (DNS). While effective and efficient, this solution requires client-side modifications (i.e. installing an extension) and does not filter out only malicious elements, possibly inducing DoS. Moreover, JavaScript code is not checked as the authors hold that the issue can be solved by Subresource Integrity [92] (SRI) - a recommendation which is poorly used in practice [28] and can be used by a malicious server to craft 'authentic' hash values for malicious elements.

An interesting idea to prevent a client from sending away credentials is to omit them in the first place. A complex system described in [13] has a trusted server provide a signature for a payment page which is then turned into a QR code and scanned by the client using a mobile application. A blockchain system would then assure that pre-registered client card data can be securely delivered to the verified payment server automatically, without the client having to enter them into any form. While we deem this solution elegant and assume that server-side implementation would be accepted in the industry, the opposite can be said for the client-side - convincing clients en mass to adopt and trust an experimental technology with their payment and personal data is close to impossible.

[4]breaking the second preimage resistance

We think that a promising avenue is represented by the use of homomorphic encryption [25] for e-commerce purposes. One could imagine that an encrypted website reaching a client would disable the malicious content inside images. Furthermore, a client would reply with encrypted data, yielding 'malformed' credentials for the attacker. However, this would require the introduction of yet undeveloped and slow protocols which might imply the involvement of additional parties in order to provide the right level of security.

2.2.3 Server-side

This subsection touches upon solutions that (mostly) require changes to the hosting provider and website authors. These solutions are elegant because they do not typically require client-side modifications, which should improve adoptability. Not all solutions found address the issue directly, but they nonetheless shed light on how to reduce web-based threats.

File integrity monitoring is proposed as a detection best practice against MageCart by the Payment Card Industry (PCI) Security Standards Council and the Retail and Hospitality ISAC [9]. To complement this picture, in 2010, [93] proposes a system for distributed storage which is secured against malicious data modifications (e.g. XSS) and is moreover able to detect misbehaving servers. It, however, enforces the switch of hosting providers to a distributed cloud system, which is a change unlikely to be adopted widely. Other best practices mentioned in [9] are the use of vulnerability security assessment to perform scans and engaging in periodic penetration testing. On this note, it is found that automated pentesting has an accuracy of 70% (compared to 100% for manual pentesting) [68] and that multiple automated tools (e.g. BurpSuite [1] and OWASP ZAP [5]) should be used for consistent results [86], [99]. A comprehensive survey [27] shows that not all web vulnerabilities can be identified by current mechanisms (e.g. static, dynamic analysis and black-box testing). They also show that protection techniques (such as attack-agnostic sandboxes and URL reputation lists) are attack-independent and require almost no human interaction, while returning no false positives.

Content Threat Removal [95], as discussed in [18] and [96], counters steganography-based malware by removing redundancy from the encoding of each file before sending them into the network. Here we can imagine a malicious hosting provider easily disabling CTR, as CTR's design perspectives did not consider such a complication. Moreover, CTR will not drop malicious elements that load e-skimmers dynamically on client-side. Suffering from the same drawback, but designed for stegoimages in particular, [102] introduces the use of a neural network that sterilises stegoimages created with Invoke-PSImages [37] - which only embeds PowerShell scripts into PNG images. It is able to destruct malicious payloads in 25ms without a dedicated Graphical Processing Unit (GPU). On top of the previously mentioned drawbacks, it is mentioned that this solution does not detect stegoimages, so a delay will be inevitably added to the transmission, proportionate to the number of transmitted images.

A server-side upload filter for malicious HTML elements is proposed by [35] to prevent content-sniffing XSS attacks. Regular expressions based on these elements are derived to detect malicious ones with no false negatives. However, the authors mention that this type of solution will not be able to keep up with evolving sniffers and client device heterogeneity. Another sanitising solution found is JS-SAN: sanitising web apps' inputs against JavaScript injection [42] in HTML5-based applications. The method works by clustering and identifying attack vectors based on templates. The authors test the application on just two web applications and obtain promising results, although for some types of injection attacks, their solution only achieves an 89% rate of true positives.

Intrusion Detection Systems (IDSs) typically use signatures of certain behaviour to determine whether a node within a network is malicious and [21] proposes an extension with genetic algorithms to adapt and evolve with new threats. Their initial results show that it can detect application layer attacks (e.g. XSS, Simple Query Language (SQL) Injection).

A survey on code injection countermeasures [65] shows that static analysis tools (e.g. model checking, data-flow analysis, etc.) are easily implementable and used during web application development, while dynamic analysis tools (e.g. whitelisting, runtime tainting and policy enforcement) require modifications that reduce adpotability and are hence employed during production. Approaches discussed are flexible, but the authors point out that attackers "continuously find new ways to introduce malicious code..."[65]. Moreover, a classification of XSS attacks and defences [43] regards that defences cannot protect against Document Object Model (DOM) based attacks and typically require heavy modifications on both client and server-side.

Considering supply chain compromise, the authors of [70] investigate the issue of including JavaScript from third parties and find that many high-profile websites do not take proper security measures for importing remote JavaScript, and sometimes have typos in their links, leading to unknown files being finally delivered to the client. The authors also experiment with the execution sandboxing technique for intercepting and evaluating executable code. Their experimentation concludes that fine-grained analysis profiles are better than coarse-grained, but they require constant adaptation as the malicious scripts evolve as well and are therefore hard to maintain.

2.2.4 Client-side

We will briefly discuss client-side solutions, for exploratory reasons mostly, as we do not find this category as an elegant approach against e-skimmers. Precisely, clients should not have to bear the risks of e-commerce platforms' or hosting providers' stale security posture, and such, server-side solutions are deemed better [33] and more elegant.

Clients can use tools for determining whether a website is malicious or not [12], [46]. These tools can also be implemented by third parties or by server administrators to help clients avoid attacks. For [12], Document Object Model (DOM) feature extraction is pipelined to a classification algorithm (i.e. XGBoost [30]) yielding an accuracy of 98.48%. However, the authors mention that their solution might fail to detect malware enabled by JavaScript, Flash or images.
A detection rate of 99% is achieved [46] with a K-Nearest Neighbor (KNN) model trained on features extracted using static and dynamic analysis from JavaScript code and respectively the DOM changes. The author suggests that a hybrid type of analysis should be developed to compensate for the shortcomings of existing static and dynamic analysis systems.

In the context of securing banking clients from phishing websites, Pixastic [90] is proposed - a web extension that decodes a steganographic message hidden by the bank in a website. The web extension can then decide at runtime whether the website is clean or not based on the extracted payload. Although interesting, one can imagine that a compromised distribution server can be used to deploy malicious versions of the websites in question, yielding positive checks. Moreover, it is heavily dependent on the client synchronising the extension with any updates, which adds even more effort on the part of the user. Another proposed web extension is made in [19], where a Chrome extension does dynamic analysis to prevent loading JavaScript skimmers and blocks outgoing requests going to attacker domains. It has a reported 97.5% detection rate and a 12% increase in load time, but it limits itself to being able to stop only JavaScript skimmers and is prone to be left out-of-date or one step behind new attacks by relying on a blacklist.

Execution-based web content analysis is explored with SpyProxy [66]. It involves a middlebox (running a Virtual Machine) residing inside the client or a network, intercepting (unencrypted) traffic and applying on-the-fly execution analysis to block malware transmissions to the client. It shows good results, adds 600 milliseconds of latency for page rendering, but theoretically struggles to work correctly with non-deterministic websites (i.e. involving session-based randomness).
Another, newer version of this approach is found in Cujo [55], a learning-based approach which uses static and dynamic analysis inside a proxy/middlebox to identify JavaScript attacks before reaching the client. Their empirical evaluation showed that out of 200,000 web pages, Cujo detects 94% of the attacks.
Cujo's signature-based counterpart is found in JSSignature [69] - instead of JavaScript code processing, the authenticity of the third-party JavaScript is verified using digital signatures and an in-browser agent. This approach can protect clients efficiently if pieces of code are modified, while not imposing technical or compatibility drawbacks on clients or third parties involved. The authors mention intending to extend and transform JSSignature into a World Wide Web standard for all third-party resources, not just for JavaScript ones.

2.2.5 Policy-based

Content Security Policy (CSP) allows developers to specify permissions for loading and running JavaScript resources [81], but does not stop a malicious server from modifying the existing CSP to allow 'self' in-line scripts to be executed. An earlier study concluded that HTML security policies "today have too many problems to be used in real applications" - primarily functionality restrictiveness and performance clogging [94].
Another policy-based solution named PHMJ [60] is proposed as an improvement over CSP, where the developer outsources an additional policy for the web page which can be enforced by the browser. It provides an efficient and comprehensive protection mechanism against (dynamically loaded) malicious HTML and JavaScript code and allows developers to better control high-risk JavaScript Application Programming Interfaces (APIs). Yet, as with CSP, we foresee issues with an attacker that can modify the PHMJ website policy to allow malware execution.

2.2.6 Third-party-based

This brief category of solutions relies on or is a service offered by a third party. These benefit from not necessarily relying on either client or server-side modifications, but also from an alleged better security posture than the client and server. However, the trust investment into a third party's secure operations and decision-making brings risks that some parties cannot tolerate.

A patent from Google proposes a Security as a Service (SECaaS) solution [24] that passes web pages through their security modules and, if benign, tracks their changes over time, distributing hash lists of non-malicious web pages and their URL. These hash lists are stored locally by clients and used to gain faster access time only to verified and authenticated web pages. One drawback we observe is the same as for [59]: web pages containing malicious code are not delivered at all, leading to implicit DoS for trivial elements such as images.
MageReport [4] is a website which provides reports of MageCart presence in Magento [3] platforms based on behaviour-based identification patterns. While not state-of-art, tools like this can help to easily identify older e-skimmers living in platforms [82].

Table 2.2: Strengths and weaknesses for each category of solutions

Category	**Strengths**	**Weaknesses**
Stegoimage detection	Capability of evaluating malicious intent; Allows for granular action against e-skimmers.	Unable to defend against new methods (3 years of delay); Can be bypassed by dynamically loading stegoimages at client-side.
Protocols and Frameworks	Capability for improving security beyond defending against e-skimmers.	High degrees of complexity; Slow implementation and adoption process.
Server-side	Prevents the spread of malicious content over the internet; Effective and relatively easy implementation.	Can be invalidated/deactivated by an intruder; single point of compromise; Some solutions only address a single aspect .
Client-side	Clients have control over the security posture they get.	Impractical adoption process, therefore with low real-world impact.
Policy-based	Logically efficient; Relatively easy to implement.	Susceptible to deactivation on server-side by an intruder; Restrict functionality and can heavily reduce performance.
Third-party based	Provides a safety net for both the involved parties; Parties can rely on the security specialty of the third party.	Sacrificing security autonomy; Few solutions found.

Table 2.3: List of solutions drawn from the literature and their drawbacks

Category	***Solution name* / work title**	**Drawbacks**
Protocols and frameworks	Secure E-commerce Transaction [101]	The website initiating the transaction is not protected by the same integrity measures as the transaction itself.
	WS-Security tokens [15]	Cannot apply to modern context because it is SOAP-based. Furthermore, it was not explored further in the literature.
	An efficient secure electronic payment system for e-commerce [44]	It disconsiders the risk of a compromised hosting provider server.
	WebShield [58]	Incompatibility with HTTPS and performance bottlenecking for frequently-updating web pages The middlebox can be skipped by steganography-based attacks.
	Unified Detection and Response Technology for Malicious Script-Based Attack [97]	Cannot prevent attacks that use HTML tags instead of scripts. The middlebox can be escaped by steganography-based attacks.
	Verena [52]	Enforces integrity for databases only, not images/websites. Involves client-side modifications and does not provide confidentiality out-of-the-box.
	HTML integrity authentication based on fragile digital watermarking [57]	Has embedding limitations. It can lead to indirect DoS. Relies on client-side filtering.
	Method and apparatus for providing geographically authenticated electronic documents [71]	Unreliable in the context of cloud hosting providers. Relies on end-user for filtering.
	Authenticity and revocation of web content using signed Microformats and PKI [73]	Involves client-side modifications. Malicious scripts can be interpreted before element checking occurs.
	Trusted cloud computing with secure resources and data colouring [48]	Does not provide a method of enforcing end-to-end integrity of transmitted data.
	Why HTTPS Is Not Enough–A Signature-Based Architecture for Trusted Content on the Social Web [77]	Does not apply to modern context because it is Extensible Markup Language (XML) based. It does not work with HTML tags and is not generic enough.
	Ensuring Web Integrity through Content Delivery Networks [59]	Involves client-side modifications (browser extension). It filters out an entire transmission instead of only malicious elements.
	Subresource Integrity [92]	It is poorly used in practice and allows a malicious hosting server to authenticate malicious elements.

	Protecting Users from Compromised Browsers and Form Grabbers [13]	Complex and reliant on experimental technologies that are yet to be fully researched. Hard for users to adopt.
	Fostering the Uptake of Secure Multiparty Computation in E-Commerce [25]	Still in the research phase. Fundamentally slow. Might involve additional parties.
Server-side	Automated pentesting	Multiple automated tools should be used for consistent results. Manual pentesting has been found to be more accurate.
	Toward secure and dependable storage services in cloud computing [93]	Not all hosting providers can or want to switch to a distributed cloud storage system.
	Content Threat Removal [95]	Can be disabled by a malicious hosting provider as it is an egress solution. Moreover, some e-skimmer loaders can still be effective after being reconstructed by CTR.
	Sanitization of Images Containing Stegomalware via Machine Learning Approaches [102]	Works solely for PowerShell scripts and PNG images. Adds scaled delay per each image.
	A robust defense against Content-Sniffing XSS attacks [35]	Only addresses content sniffers injected by XSS. Cannot keep up with evolving sniffers and client device heterogeneity.
	JS-SAN [42]	Limited testing. 89% rate of true positives for some types of injection attacks.
	A signature-based intrusion detection system for web applications based on genetic algorithm [21]	Involves client-side modifications and addresses injection attacks only.
Client-side	An effective detection approach for phishing websites using URL and HTML features [12]	Might fail to detect malware embedded in images, despite yielding an accuracy of 94.49%.
	A novel approach for analyzing and classifying malicious web pages [46]	Comes with shortcomings of relying solely on static and dynamic analysis.
	Pixastic [90]	Relies on synchronising updates. A compromised server can update the plugin to work for a malicious version of the website.
	Identifying JavaScript Skimmers on High-Value Websites [19]	Relies on a blacklist, which will not catch zero-day attacks. 12% increased load time. Limited to JavaScript only.
	SpyProxy [66]	Does not work correctly with websites involving session-based randomness. Being a middlebox, it can be escaped by steganography-based attacks.
	Cujo [55]	Can be escaped by steganography-based attacks.
	JSSSignature [69]	Only addresses JavaScript code provided by third-parties.

Policy-based	*Content Security Policy* [81]	A malicious hosting provider can allow its own added scripts to be executed. Moreover, it restricts functionality and bottlenecks performance.
	PHMJ [60]	Although an improvement (in other areas) over CSP, it suffers from the same essential drawbacks.
Third-party based	Providing a fast, remote security service using hash lists of approved web objects [24]	It filters the whole transmission, instead of only malicious elements, resulting in an indirect DoS.
	MageReport [4]	Addresses only Magento-based platforms and relies on (not state-of-art) behavior patterns.

2.2.7 Recommendation

The literature shows (partial) solutions for the problem of e-skimmers, varying from server-side to client-side, Machine Learning and middlebox/sandbox approaches. While we deem these solutions good, we observe drawbacks and limitations that make them unsuitable for the setting of a hosting provider delivering malicious payment pages. We also see that hosting providers need and are the best suited to combat the issue of malicious web pages [33].
Given that the solution and attack both reside in the same place, we propose the following features list of an ideal solution for stopping e-skimmers :

1. Maintains content integrity under malicious hosting provider.
2. Works with all attack surfaces of e-skimmers (including dynamically-loaded content).
3. Requires no client-side modifications.
4. Integrates with TLS/HTTPS.
5. Involves both manual and automated pentesting prior to web page publication in order to detect zero-day attacks.
6. Stops only the transmission of malicious elements to avoid indirect DoS.
7. Allows the client to verify the authenticity of received data.

Recommendation 1 helps assure that the client receives only data intended by the e-commerce platform developer and furthermore Recommendation 7 allows the client to verify that the data indeed belongs to the developer. An ideal solution (Recommendation 2) would work not only with images, but with code blocks, other media assets and even scripts that load malicious content at runtime. In order to be implemented in a practical and effective manner, the ideal solution should not ask for any adaptation on the client-side (Recommendation 3). Transmitted data needs to be always secured, so any ideal solution should not interfere with the best practice of relying on TLS/HTTPS for data transmission (Recommendation 4). To properly ensure that the developer itself does not push the malicious content to the hosting provider, systematic security testing should be performed (Recommendation 5). Finally, Recommendation 6 assists in maintaining service availability and protecting the client at the same time by only denying malicious data to reach the client.

Additionally, future researchers should read about the weak points of our methodology in Subsection 2.2.8 to improve upon the literature review conducted here.

2.2.8 Methodology limitations

Let us now discuss the shortcomings of our literature review process. These should be taken into account by future research to build upon the work presented here, but also by readers wanting to assess the quality of the process itself.

First, a trivial limitation, but relevant nonetheless, is the implicit bias of the authors due to an incomplete or skewed perspective on the subject. This is especially relevant in building the search terms and in conducting the in-depth manual search. As an effect of building incomplete or biased search terms, the search process might not be completely covering the boundaries of the research topic, leaving interesting ideas undiscovered.

The second limitation is related to the words used in the search terms for querying a single aggregator. It can be that better results would have been obtained if slightly different or synonym terms would have been used instead or that a different aggregator might have given different results. We deem that some variations (in terms and/or aggregators used) would prove to be slightly beneficial for the obtained results.

The third limitation again addresses the search process, but is referring to the inability to programmatically apply the inclusion/exclusion criteria based on the title and abstract retrieve through the Google Scholar API. As discussed in the section regarding the programmatic search, one can only retrieve a snippet of the abstract, which has proven to us insufficient as an input for OpenAI DaVinci (the 'smartest' GPT3 [22] available through an API at the time of writing) to determine the relevance of one work. A method of obtaining the full abstract would greatly benefit the automated literature review process, reducing the time needed to filter thousands of works in a matter of minutes.

Lastly, the performance evaluation of selected works might be imperfect and therefore skew the opinions on the strong and weak points of each solution. Although we deem that this would not greatly impact the final recommendations made, it is important for future researchers to check for themselves whether an individual assessment of one solution is fitting.

2.3 Conclusion

We reviewed many research works to gain a comprehensive overview of addressable solutions and ideas against MageCart threats. Although the literature was not specifically oriented to these types of threats, by checking on the boundaries of this topic, we were pleased to find a diverse range of applicable measures, which were diverse enough that we could compile recommendations out of their strong and weak points. These recommendations are relevant for fellow researchers and for establishing targets for our proposed solution's design.

Chapter 3

Proposed solution

Based on the recommendation list compiled by broadly analysing the literature in Subsection 2.2.7, we create our solution to fill in the existing gaps with a simplistic and compatible approach that provides integrity under the premise of a compromised hosting provider. We begin by discussing our threat model to understand the assumptions and limitations in place for the parties involved. We follow with a brief summary of the usefulness and relevance of cryptographic digital signatures, then describe the inner workings of our solution and conclude by evaluating the identified research gaps against the proposed solution.

3.1 Threat model

Understanding the picture we composed before designing our solution is essential for evaluating the relevance of this work. The attacker, hosting server and the e-commerce platform developer are assumed to have certain limitations, modus operandi and responsibilities.

First, the e-commerce platform is ultimately the one that holds responsibility for the security of the customer's payment credentials, even if a hosting provider is compromised. The developer is assumed to take responsibility for all security aspects regarding their web page source code, including third-party source code, media or resources used for their service. In more technical means, this implies that the developer has a form of security testing for their web page, especially for modifications that relate to its source code and images. This testing would include specific tests for images, including steganalysis techniques, meant to make sure that stegoimages are not being published from their repository to the hosting provider. However, we do not assume that these security testing procedures will catch or manage to solve all the problematic vulnerabilities, leaving the source code exploitable by (novel) injection attacks.

Second, we assume that the hosting provider has a precarious external security posture due to the low return on investment compared to other areas such as performance, connectivity, uptime and multi-tenant isolation [23]. On top of this, injection and remote code execution vulnerabilities from their clients' source code can open the door to an attacker gaining access to the physical server, possibly compromising all services running on that said unit. In this context, attackers have multiple options for gaining and elevating their access to deliver e-skimmers and propagate further. We assume the extent of security monitoring performed by hosting providers is to guarantee service availability and cross-tenant contamination, yet system or network-wide monitoring is not as thorough and therefore might allow an intruder to maintain elevated privileges for an extended or indefinite period of time. The result of either the injection or the intrusion into the server is that stegoimage e-skimmers are placed into the payment page.

Third, the attacker is assumed to have gained elevated access in the hosting server and therefore influence the running services on it. Although this would possibly allow the attacker to perform lateral movement within the hosting provider's network, we limit those capabilities of our attacker model in favour of exploring it in future works. Hence, the hosting server is assumed to be fully malicious under the intruder's command, but the network it resides in and its nodes are unaffected. The affected websites are assumed to contain stegoimage e-skimmers created using novel or state-of-art techniques (e.g., Generative Adversarial Networks) and therefore be virtually impossible for most related works to detect or stop. We also assume that the attacker would be mainly interested in modifying the affected websites such that minimal suspicions are being raised from the clients, hosting provider or developer.
Therefore, our solution is designed to counteract additions and alterations of web elements, not necessarily other types of changes such as structural, logical or aesthetical ones. To this end, digital signatures are leveraged to identify deviations from a developer-validated form.

3.2 Digital signatures

A digital signature is a unique string of bits generated by a mathematical algorithm that depends on two parameters: the private key of the signer and the input message or document [53]. Any changes to the input will result in a different signature, ensuring its uniqueness and providing integrity and non-repudiation. Digital signatures are widely used in e-commerce and other applications where message authenticity is crucial [79]. Their use has also been noted in related works in the literature and inspired our own research.

The uniqueness property of digital signatures lies at the heart of our proposed solution. We leverage this uniqueness to tie every image of a website to its developer's private key and append the resulting signatures to the website. This results in a collection of signatures that can be checked for authenticity by any party through the developer's public key. To prevent any misbehaviour of a hosting provider to affect a client, a reverse proxy is attached to the hosting server and filters downstream traffic by checking the found images and their signatures against the public key of the developer. Finally, the result is that even if an intruder injects stegoimages or modifies the attached signatures on the server-side, the client will not receive the malicious changes. We coin this solution *NAISS: Network Authentication of Images to Stop e-Skimmers*, but we acknowledge the future possible extension of its capabilities to cover other attack vectors of MageCart [80], [35].

Figure 3.1 shows the flow and architecture envisioned for *NAISS* to function properly. It first begins on the developer's side where a new element (e.g. image) has been changed. The element should pass through an evaluation process to determine whether it is malicious or not. Typically this involves security testing [68], where additional to the current best practices for security testing, we assume that stegoimage detection tools can be implemented as well. Following the decision that an element is not malicious, its signature will be computed by an employee that has access to a secure key. For example, this key can be one which is higher up in the key hierarchy of the developer. By choosing the key in that way, it would increase the difficulty of an attacker to obtain it, as it would be protected better than the other keys lower in the developer's key infrastructure. This step is needed to minimise the accidental automated signing of a malicious image, which might, for example, be committed by a compromised developer account or an injection attack.

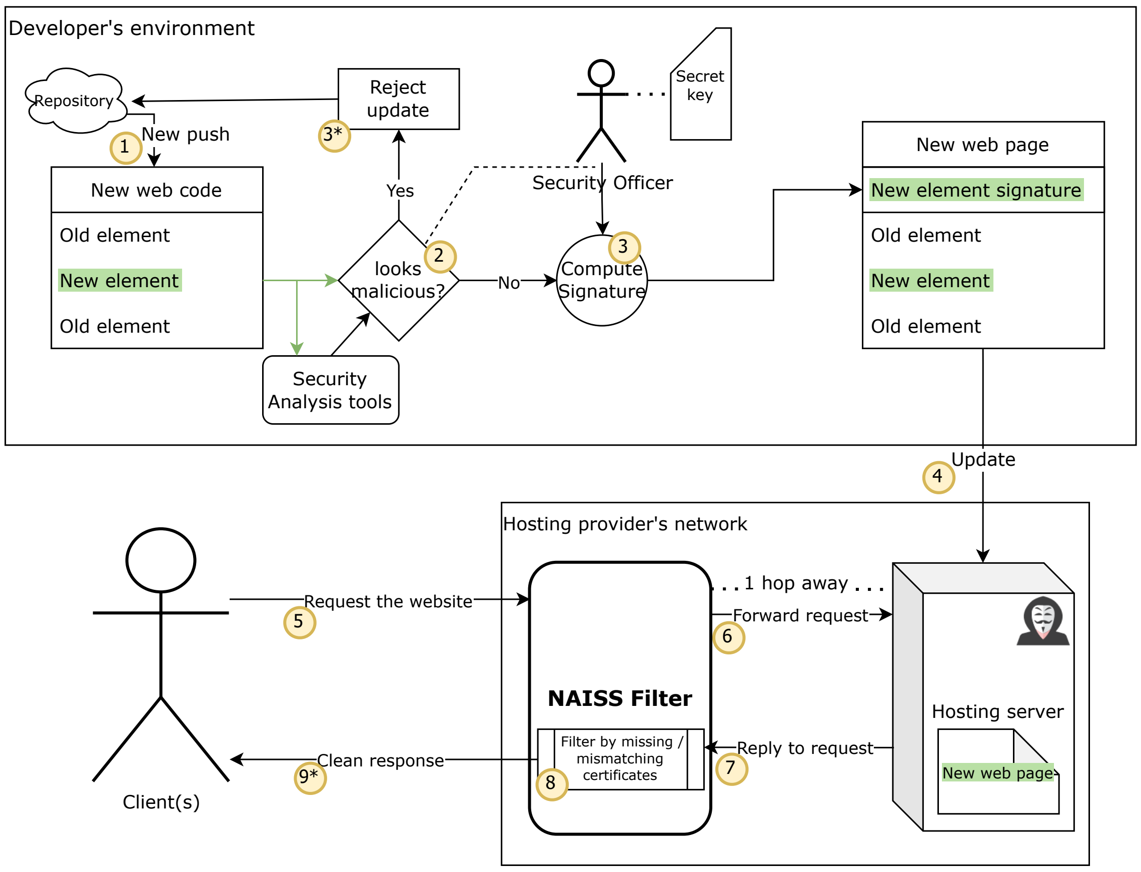


FIGURE 3.1: NAISS Flowchart

The fact that our solution requires manual interaction slows down automated pipelines and enables powerful insider attacks. However, integrity being the most important value of a payment platform [76], a time tradeoff can be a cheap price to pay for high levels of integrity. In fact, automated pentesting tools suffer from false alarms that can only be found or reduced by involving people in the process [68], [86], [27], [49], [16]. Also, as automation increases year-by-year [49], the time spent on manual inspection will be lesser as time passes.
Another factor of the time investment is the frequency and size of updates of the payment page. Considering that these mainly contain a couple of text fields (i.e., where one enters credit card number and cardholder details) and buttons (i.e., for submitting the details) with limited functionality, changes made to the page would be atomic, rare and require little background knowledge to be comprehended by reviewers. Hence, involving human-led and automated testing will strengthen the confidence that signatures will only be produced for trustful images.

The input we are using for creating the image signatures is of two types: either it is the byte string content of an image, in the case of locally-referenced images, or, in the case of images referenced by external URLs, the URL string itself. Note that in the later case, *NAISS* can only validate signatures based on the correct URL, not on the media content referenced by that URL, and hence it falls under the responsibility of the developer to ensure that external URLs do not point to malicious resources at any point in time. The generated signatures are collected and added to the *head* tag of the web page in a new tag named *naiss_ signatures*. The website is then published to the hosting provider as natural. After the new website and its content is stored at the hosting provider, we can assume that it can be compromised by an intruder inside or outside the hosting server.

The *NAISS* filter is located one hop away from the server in the hosting provider's network. It acts as a reverse proxy, and hence its Internet Protocol (IP) address is where the client finally connects instead of the server. In this way, we make sure that the supposed malicious hosting server is obliged to pass its communication through the reverse proxy filter so that its response eventually reaches the client back. The reverse proxy would represent an additional layer of difficulty for the intruder to achieve complete control over the delivered websites. The filter itself is a server modified to perform the filtering process on the GET requests for the stored websites and could be hardened to prevent lateral movement intrusion (refer to Section 4.2). It is similar to solutions identified in the literature, yet it fills in crucial gaps in a simple manner.

3.3 Research gaps comparison with *NAISS*

We can briefly showcase how our proposed solution *NAISS* generally compares to the related work based on the shortcomings identified in the literature:

1. Digital signatures can be effectively used by *NAISS* to ensure integrity as long as the input is unique enough. *NAISS* does not have any limitations for image formatting nor for any other attack vector of MageCart, as long as it can be represented through a unique byte string.

2. *NAISS* does not require any client-side modifications because it is a server-side solution. Moreover, the client will not need to change any previous behaviour, greatly increasing its adoptability.

3. Any change made (i.e. even a single bit change) to the hosted files or signatures will eventually ripple out due to the uniqueness property of digital signatures and be finally detected by the *NAISS* filter. Therefore, the only avenue for an intruder to deliver stegoimage e-skimmers is to not modify anything.

4. Regardless of how recent an attack is, if it is reliant on the modification of the already pushed data at the hosting provider, it will be caught by *NAISS*, as explained in the point above. This aspect is nuanced in 3.4.2.

Fulfilling these design gaps requires modern and reproducible methods of implementation and testing in order to provide a valuable and provable contribution to the fight against MageCart.

3.4 Methods

This section shall describe the materials, algorithms and other technical details used in our proof-of-concept implementation of *NAISS*.

We used Python [75] as our main programming language and Docker [32] as our method of building our test environment. The hosting server is a simple Python HTTP server, while the *NAISS* filter is using Flask [36] as its baseline. The decision of using HTTP instead of HTTPS was done in favour of the ease of development and testing *NAISS*, with the security implications of it being of second importance to our research objective overall. Moreover, we do not think that using a HTTPS communication would produce results that significantly differ from the HTTP counterpart.
In our testing, both the hosting server and filter containers ran on the same machine inside a shared Docker network, while the automated tests were run from the same machine, but not from within the Docker network, so as to emulate a real client as much as possible.
The source code of our work can be accessed on GitHub [83], where further usage instructions can be found. The code is mainly split into four directories:

- *client*: Contains a script for running automated tests and one for visualising the results of those tests. Additionally, the resulting plots are stored alongside the collected measurements under the *test_results* subdirectory.
- *filter*: Contains the scripts and the Dockerfile to spin up the *NAISS* filter container.
- *server*: Contains multiple website variants and the Dockerfile to run the hosting server container.
- *utils*: Contains various useful scripts that a developer/attacker would use to finally attach signatures to a website. It also contains scripts used to generate websites, which are used for testing.

3.4.1 Signatures

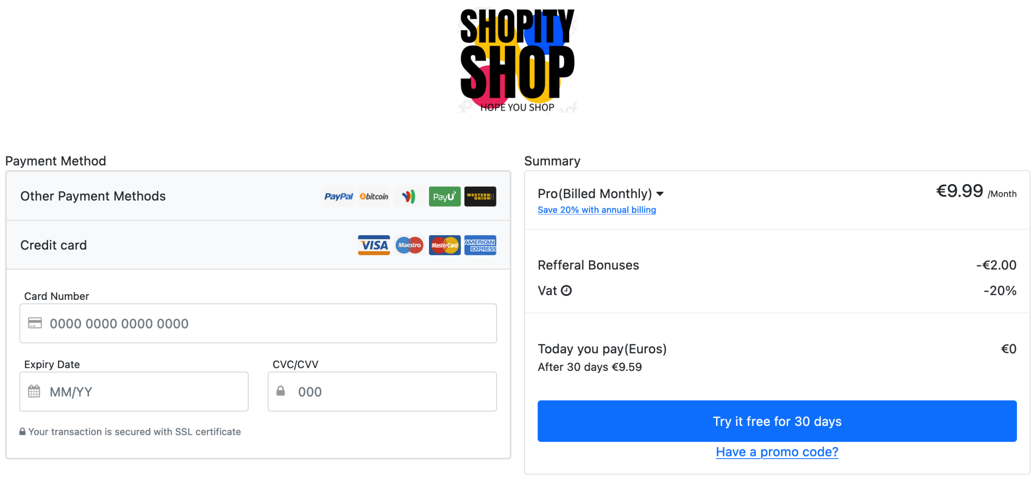
For generating and validating the signatures and their corresponding key we used python-ecdsa [38] with the NIST256p curve [10]. This curve was chosen for the practical key size (128 bits of security). Also, the computational performance of this curve is better than other alternatives provided by the library for the equivalent security level.
The input for the signatures will be the byte string representation of the images stored on the hosting server, which we coin as 'internal' images; while for images stored on external hosting services, we use the links in their respective HTML tags as input.

3.4.2 Stegoimage generation

To validate that our proposed solution can counter stegoimages generated with state-of-art techniques (**RQ5**), we choose to generate our own stegoimages by using SteganoGAN [100]. SteganoGAN leverages Generative Adversarial Networks to embed payloads, a technique which is considered the most difficult to detect [67]. In our testing, we include the image format types *.png*, *.jpg* and *.ico* as these are the most commonly used in practice for e-commerce platforms. Moreover, as touched upon in 3.4.1, our testing also includes the use of both internally and externally loaded images. The payloads accepted by SteganoGAN can be of any size, as it has an internal cutting point of 4 bits per pixel for embedding data. We tested this limit in the steganogan_threshold branch of our GitHub repository, where we embedded payloads of 45 megabytes (MB) into stegoimages, yet the resulting stegoimages would occupy an average of 25MB. In comparison, stegoimages using only 11 bytes as payload occupy 24.8MB. Stegoimages will finally be displayed in different sizes and places (e.g., as a favicon) on our test websites.

3.4.3 Website generation

Multiple variants of websites were created stemming from a single template. This template is representative of a typical website where e-commerce customers would introduce their credentials. The websites contain multiple pictures (attack vectors of stegoimage e-skimmers) of different sizes which are placed in different parts of the website. We used a royalty-free icon package [88] to represent payment method icons and LogoAI [61] to create a logo for our fictitious e-commerce platform. The images used have the following sizes: 128x80 pixels for payment method pictograms, 256x256 for favicons and 505x446 for the website logo. These sizes were chosen to reassemble a payment page where large images of different sizes are scaled down to ensure visual clarity, but also to evaluate how large stegoimages would affect the filtering. Figure 3.2 exemplifies how the website with all its images, except the favicon, would appear to a client. Each such website will represent one specific test case, which will be automatically accessed and measured for performance.


SHOPITY SHOP
HOPE YOU SHOP

Payment Method

Other Payment Methods

Credit card

Card Number
0000 0000 0000 0000

Expiry Date
MM/YY

CVC/CVV
000

Your transaction is secured with SSL certificate

Summary

Pro(Billed Monthly) €9.99 /Month
Save 20% with annual billing

Refferal Bonuses -€2.00
Vat -20%

Today you pay(Euros) €0
After 30 days €9.59

Try it free for 30 days

Have a promo code?

FIGURE 3.2: Fictitious e-commerce platform

3.4.4 Automated testing

To improve the reproducibility and execution time of our tests, we opted to use Selenium Webdrivers [85] to automate the accessing of websites and collect data from those interactions. Additionally, we used selenium-wire [54] to be able to inspect the exchanged network requests and responses. To maintain consistency of the results, all tests were run on the same MacBook Pro M1 with 8GB of RAM, with no unnecessary applications running in the background, connected over 5GHz Wi-Fi, and at a higher than 80% battery remaining.

3.5 Results

In this section, we present the testing variables, the collected data and the aggregated results. Moreover, we showcase how the change of one type of parameter affects the performance or behaviour of *NAISS*.

3.5.1 Test parameters

We aim at creating a comprehensive test through the tuning of many relevant website parameters, such that we can retrieve information about the behaviour of the *NAISS* filter under different circumstances. A website variant would contain a complete representation of the chosen parameter, with no mixed values, apart from where that was technically unfeasible. These tuned parameters are:

- The format of the loaded images: PNG or JPG; the ICO format is reserved for the favicon
- The method of loading images: through internal or externals links (i.e. a file path or an URL)
- The embedded payload inside images: none (i.e. clean images) or any, up to 4 bits per pixel [100] (stegoimages)
- The type of signatures attached: none (coined "nosig"), produced with the developer key ("sig") or an attacker's key ("evilsig")

Additionally, we investigate how the use of different web browsers would impact the interaction with *NAISS*. To this end, we ran our tests using Google Chrome, Mozilla Firefox and Microsoft Edge, as these are typically the most popular desktop browsers [87].
One last, but essential parameter we tuned was whether the webdriver would connect to a website directly (i.e. unfiltered by *NAISS*) or through the filter itself. Hence, in half of the test cases, one would expect *NAISS* to filter everything necessary, while for the other half, no filtering would occur.
Finally, all the combinations of these parameters yield a number of 54 test cases.

3.5.2 Measurements

For each test case, we are interested in collecting data to evaluate the behaviour and performance of the *NAISS* filter. These measurements will also serve as a base for experimentation with specific types of parameters (e.g. encryption curve). Through automated scripts, we collect the following interaction data:

1. The time (in seconds) to access a website.
2. The transferred data (in kilobytes) when accessing a website.
3. The percentage of images that reach the client (e.g. 0% if no images are loaded whatsoever).

The third measurement will be used to determine whether the *NAISS* filter functions as intended and the former measurements are used for performance evaluation. Multiplying by the number of test cases, we take a total of 162 performance and behaviour measurements.

In order to compile results with a degree of confidence in our measurements, we ran each test case 10 times. In this way, we see for each result category how much one can expect the measured values to vary. To this extent, the 96% confidence interval and the standard deviation of each result category is computed and displayed (refer to Subsection 3.6.1).

3.5.3 Baseline results

The collected measurements were plotted to obtain the final results. In our baseline experiment, we used images of certain pixel size (refer to 3.4.3), the NIST256p curve for signatures and the text "RENAISSANCE" (11 bytes) as the payload for stegoimages. Due to the simplicity and relatively small size of these parameters, we name the results as baseline. Subsequent experiments in 3.5.4 explore what effects any of these previously mentioned parameters have on the measured performance and behaviour.

The results from the baseline experiment are averages of measurements taken from all the collected data, grouped by the parameters in 3.5.1 (e.g., we collect and average all the measurements of all the website variants that have JPG images). The average values are computed from all 10 repetitions of each test case and displayed alongside their associated confidence interval and standard deviation. For each measurement in 3.5.2, a different plot is produced: Figure 3.3 for the access time, Figure 3.4 for the transferred data and Figure 3.5 for the percentage of unfiltered images, which is only grouped based on whether the correct signature for the images is present or not.

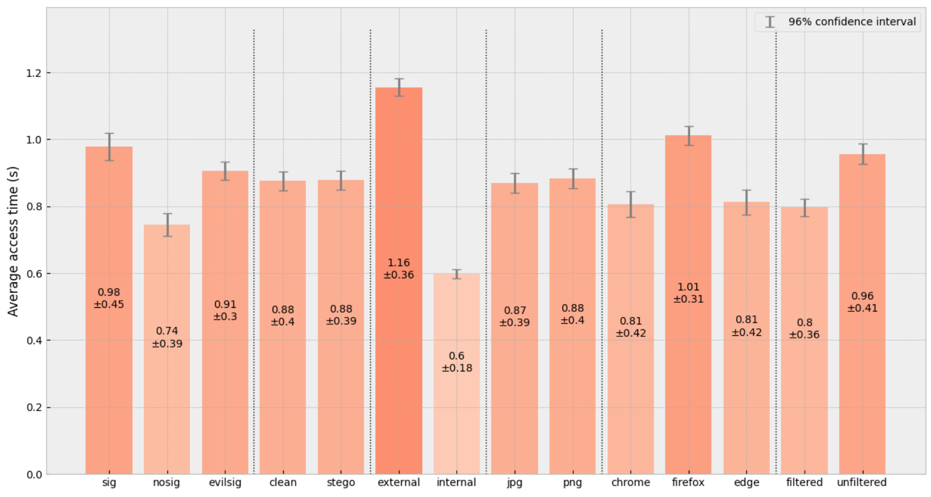


FIGURE 3.3: Baseline experiment - access time per parameter

The most significant result is the one related to the behaviour of the *NAISS* filter, the percentage of unfiltered images. We see that for all of our test cases, 100% of images with the correct signatures arrive at the client, compared to 50% for the incorrect signatures, which is precisely expected as only half of the test cases involve connections through the *NAISS* filter. These results confirm that the method of filtering images based on attached signatures is working as intended.

The performance results show a connection between the transferred data and the access time of a website, hence we can expect that an increase in transferred data will create a slight increase in access time. The transferred data values for each category of parameters are topologically identical to the access time values.
We observe that the websites with no signatures have the lowest values, while the ones with a correct signature have the highest - logical given that verifying and transporting

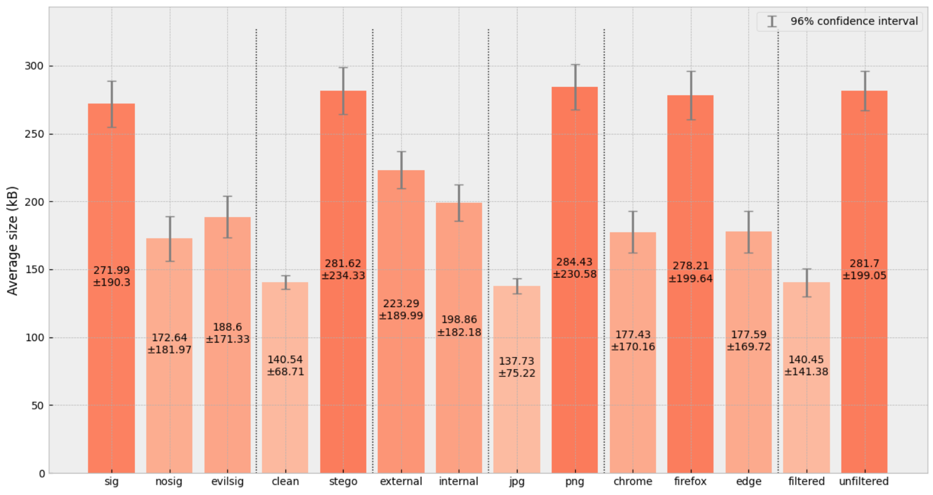


FIGURE 3.4: Baseline experiment - transferred data per parameter

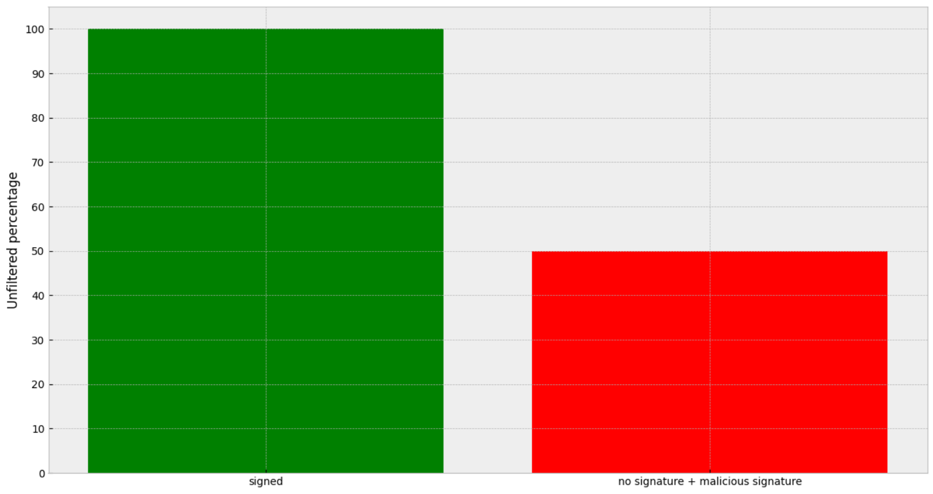


FIGURE 3.5: Baseline experiment - percentage of images reaching the client

signatures are additional steps to be taken. Although the size of websites with stegoimages is close to double that of the websites with clean images, the time difference in accessing them is orders of magnitude smaller. The difference between websites with externally and internally loaded images follows the same pattern, with the external images yielding higher values. The same is observed for the image format category: as expected, the JPG images are more compressed and therefore require less bandwidth to be transferred. Regarding browsers, we see that the Chromium-based browsers are both faster and transfer less data. Finally, the websites accessed through the *NAISS* filter have transferred close to 50% less data and 16% faster than those directly accessed by the client.
These results are called baseline results due to the simplicity and smaller sizes of the parameters used. Experimentation is needed to assess how a value increase in any of these parameters affects *NAISS*.

3.5.4 Experiments

Each experiment is designed to investigate the significant changes of one single parameter compared to the baseline experiment. For this purpose, we chose the following three parameters: image size, stegoimage payload and encryption curve. The image size is interesting because a website is expected to have images of varied sizes, which are easily changeable and resized during a webpage's lifetime; the stegoimage payload is used to observe how would the *NAISS* filter work with stegoimages containing real e-skimmers; and the encryption curve is investigated to probe how much of an impact increasing or decreasing the key and signature size has on our solution. Additionally, we ran an experiment with all these previously mentioned changes occurring simultaneously (Figure 3.6, 3.7 and 3.8). In GitHub, each experiment has its own separate branch, so navigating the results is uniform across all experiments.

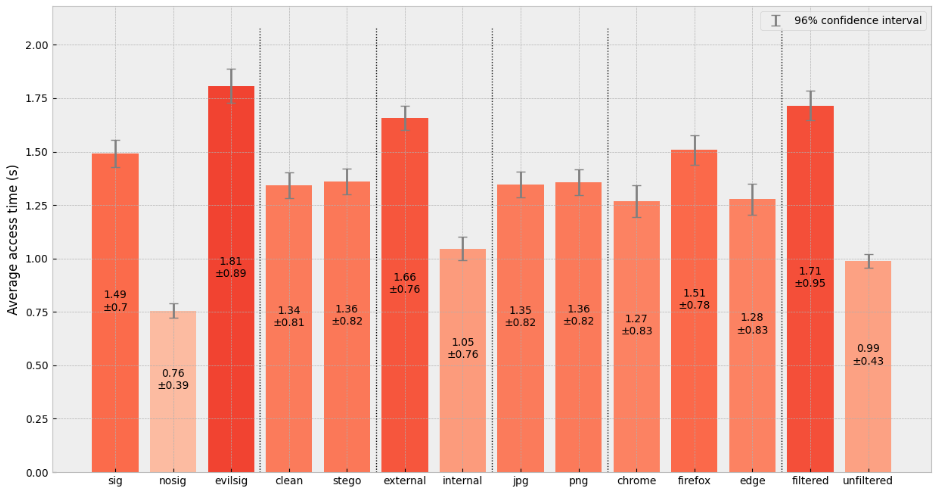


FIGURE 3.6: All parameter experiment - access time per parameter

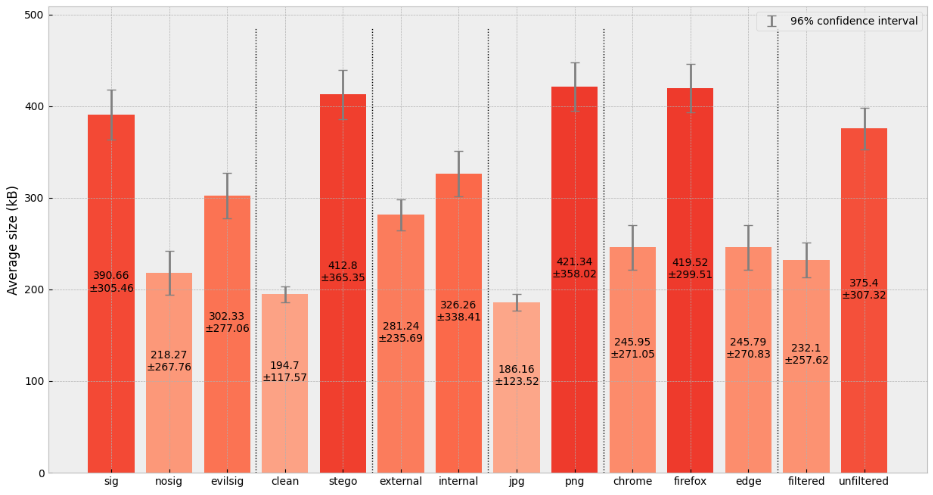


FIGURE 3.7: All parameter experiment - transferred data per parameter

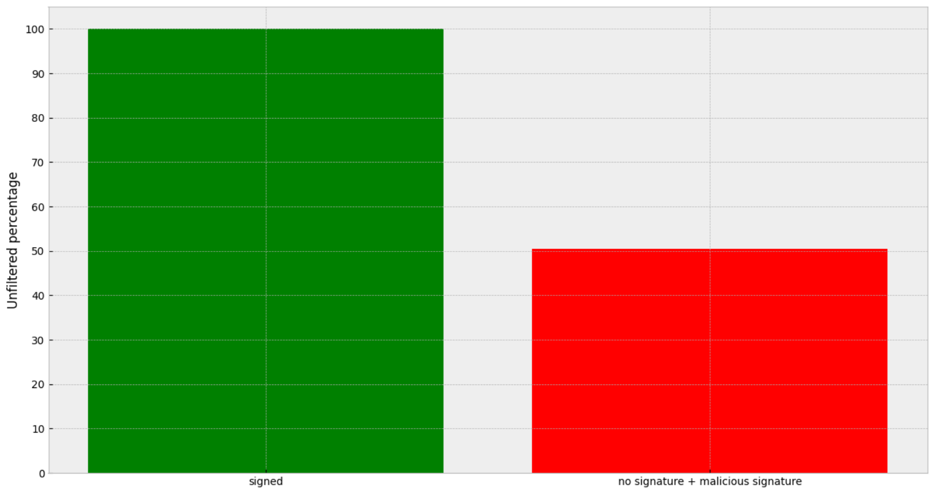


FIGURE 3.8: All parameter experiment - percentage of images reaching the client

Image size

In this experiment, the pixel size of all the used images is doubled. The results (Figure 3.9, 3.10 and 3.11) show the same topology as for the baseline experiment, except a 3% increase for the access time in the stegoimage category. The main difference between the baseline experiment is higher values, especially for the transferred data. This experiment further contributes to the idea that transferred data insignificantly increases the time taken to load a web page.

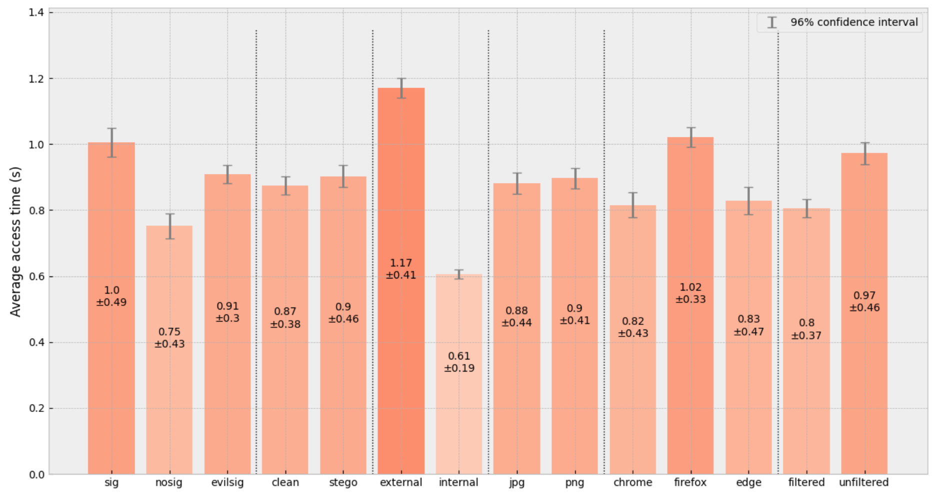


FIGURE 3.9: Image experiment - access time per parameter

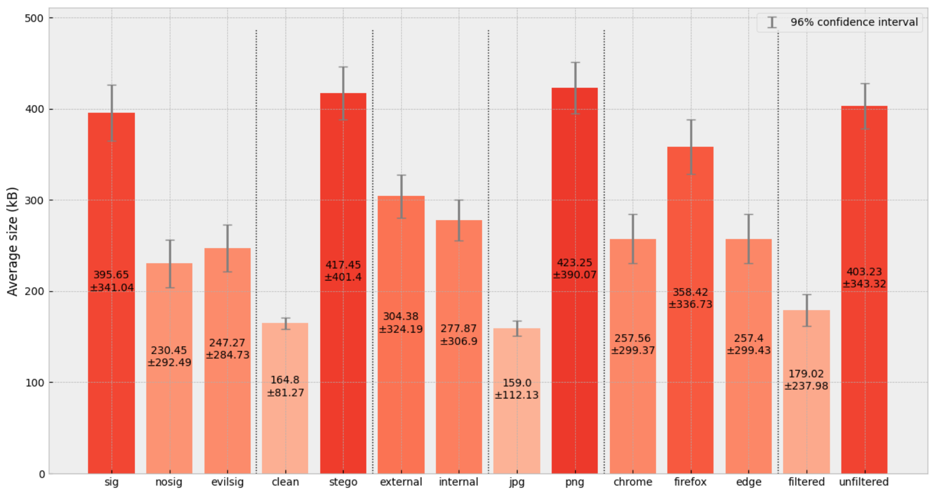


FIGURE 3.10: Image experiment - transferred data per parameter

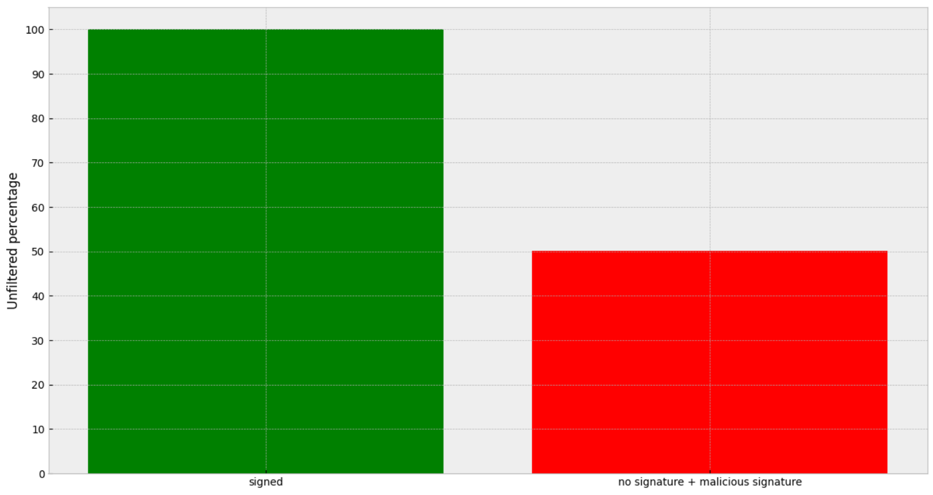


FIGURE 3.11: Image experiment - percentage of images reaching the client

Payload size

By changing the payload to a realistic one, we emulate how *NAISS* would react to real e-skimmers hidden in images. As a realistic payload, we used an e-skimmer script (4,5 kilobytes) caught in the wild [63]. By increasing the size of the payload, we also slightly increase the storage size of the stegoimages. The results (Figure 3.12, 3.13 and 3.14) are similar to the image size experiment, with all values increasing by up to 11% when compared to the baseline.

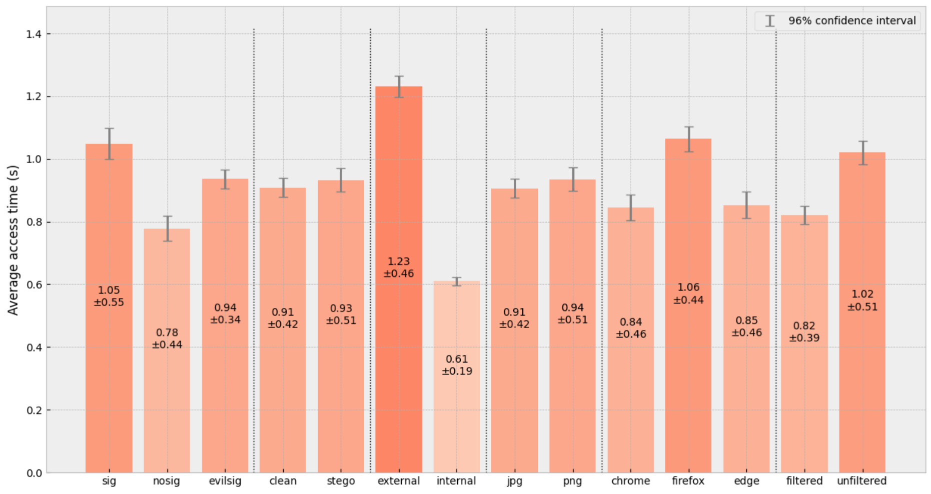


FIGURE 3.12: Payload experiment - access time per parameter

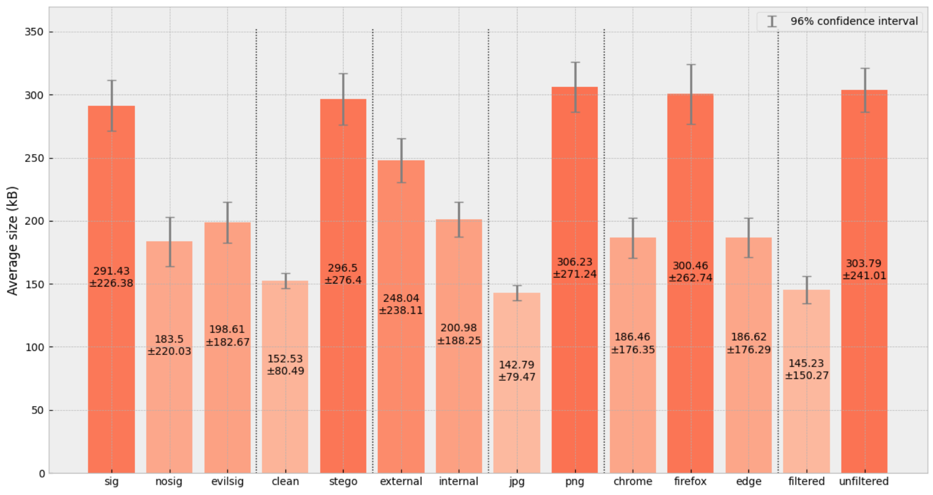


FIGURE 3.13: Payload experiment - transferred data per parameter

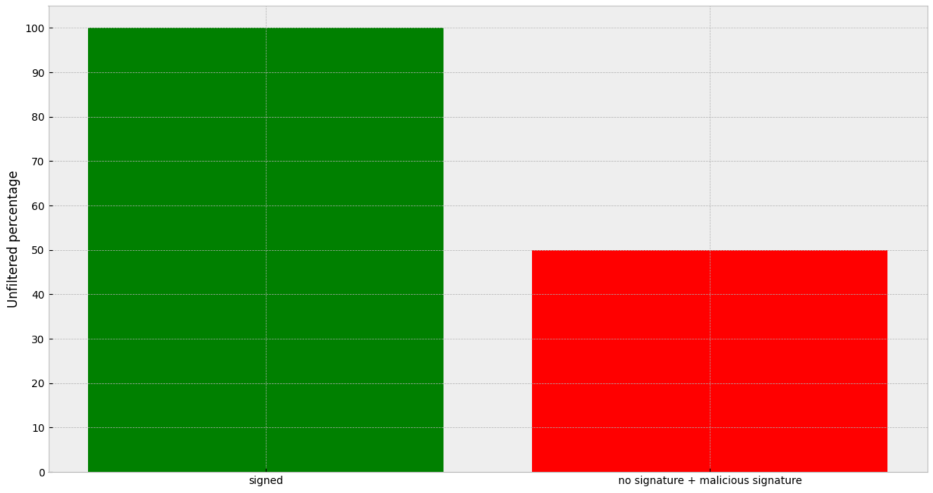


FIGURE 3.14: Payload experiment - percentage of images reaching the client

Encryption curve

For this experiment, we changed the elliptic curve used to NIST512p, increasing both the storage size of attached signatures and the time to verify a signature. The purpose of this experiment is to evaluate how this technical aspect would impact our solution, such that an implementer can find the desired balance between security and performance. The results (Figure 3.15, 3.16 and 3.17) are topologically different in the signature and connection type categories. The websites signed with an attacker key now have the longest access time, although not the highest transferred data size. The other significant change is observed in the websites that are accessed through a *NAISS* filter, where the access time has more than doubled and transferred data increased by approximately 40% compared to the baseline results. These large shifts in measurements are due to the fact that by doubling the bit size of the signature and the key (i.e., from NIST256p to NIST512p), one effectively doubles the computation time needed to validate a signature. One interesting finding is that the "evilsig" category has the largest latency, hinting that verifying an invalid signature is a slower process than verifying a valid one.

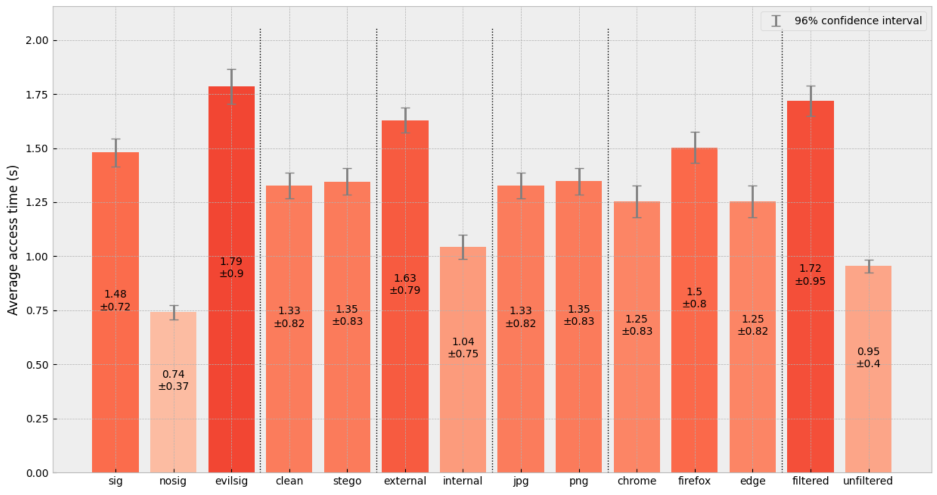


FIGURE 3.15: Encryption experiment - access time per parameter

For all the performed experiments, the percentage of images reaching the client (measurement 3 from Subsection 3.5.2) remained the same as for the baseline experiment, highlighting that the behaviour of the *NAISS* filter is consistent and correct under any parameter tuning (refer to Subsection 3.5.1). The time taken to access websites through the filter is maximally double when compared to a direct connection, with the externally-loaded images and the elliptic curve size contributing the most to this increase. Further similar connections are made in Section 3.6.

3.6 Discussion

In this section, we interpret the results and discuss their implications for the proposed solution. We use the payload experiment as our representative scenario of the real-world

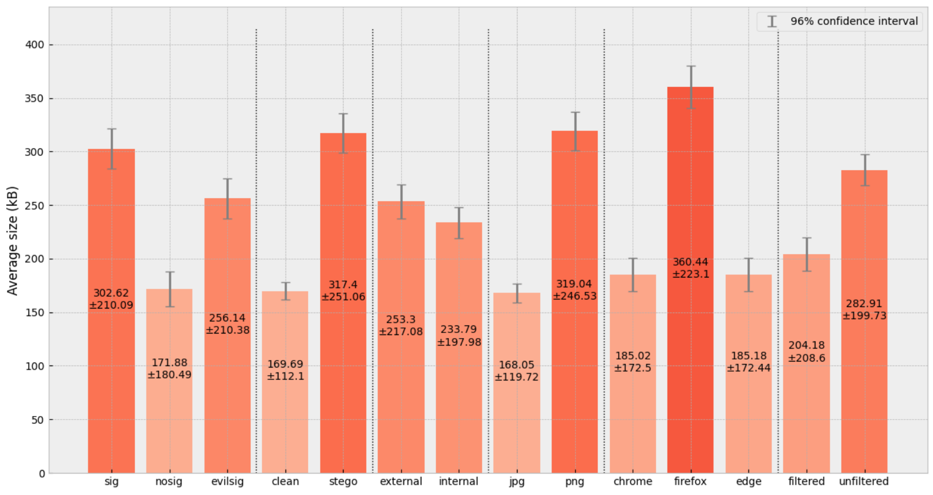


FIGURE 3.16: Encryption experiment - transferred data per parameter

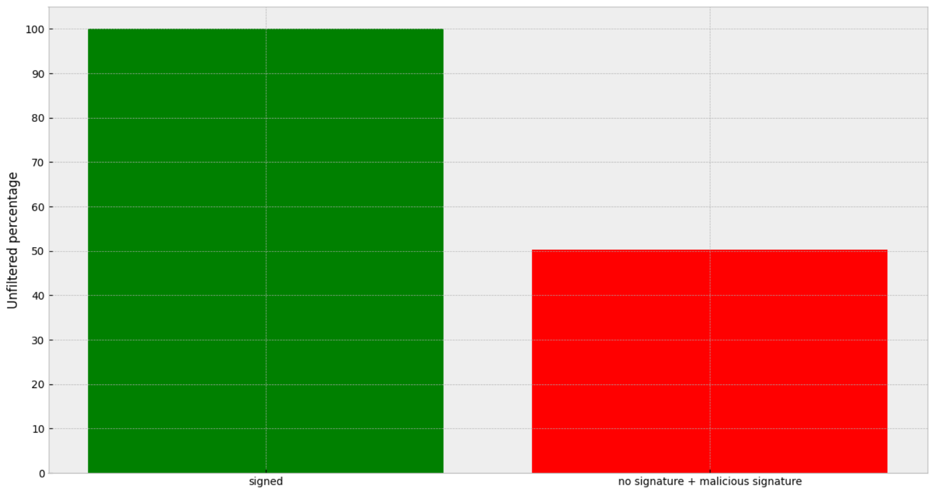


FIGURE 3.17: Encryption experiment - percentage of images reaching the client

context. Our findings indicate that the behaviour of the *NAISS* filter is effective in discarding all images that are missing or presenting a malicious signature, regardless of any parameter combination used. If malicious signatures are present in a filtered connection, it naturally results in a faster access time on the client-side by up to 11% compared to valid signatures and 25% in the case of no signatures.

The difference in access time between "sig" and "nosig" is due to the fact that the filter prevents the loading of external images for "nosig" websites, which have been found to contribute the most to the detriment of the loading time. The presence of signatures minimally impacts access time, as websites with signatures show access times very similar to those with no signatures when using unfiltered connections. Therefore, the differences in the signature type category occur solely from the filtering process.

Our study found that the filtering process will either significantly reduce the load time due to loading less content or will add additional time because of the signature verification process. The added time for "evilsig" variants is up to 50%, while for "sig" variants, it is less than 10% compared to an unfiltered connection. Verifying an invalid signature is slower than verifying a valid one, possibly allowing an intruder to degrade the access time by attaching multiple incorrect signatures.

Transferred data is found to be correlated with access time, although it does not highly influence it. The factors that most impact the loading time of a website through the *NAISS* filter are, in order: the use of a slower signature scheme, the loading of external images, the use of Mozilla Firefox, and the signature verification process itself. The two most significant factors are modifiable on the developer side and do not highly impact the level of security. Hence, the performance of using *NAISS* can be appropriately tuned such that clients can be better protected from MageCart attacks without noticing the presence of the filter.

3.6.1 Benchmark

To better compare and evaluate the performance and characteristics of *NAISS* with the solutions identified in the literature, a comparison benchmark has been compiled in Table 3.1. The comparison criteria are meant to evaluate one solution's performance, behaviour, robustness and ease of adoption:

- **Imposed latency**: the delay incurred by a client retrieving the webpage, as reported by the authors of the work (i.e. either in percentage or in absolute value)
- **True Positive rate**: the percentage of correctly identified malicious elements
- **Flexibility**: the capability of functioning correctly with varied inputs and contexts, including novel attacks
- **Adoptability**: the ease of adoption and use from the servers' and, but especially, clients' side
- **Bypassable by intruder**: whether an intruder inside the hosting provider can disable, alter or ignore one solution such that e-skimmers can be delivered to clients
- **Can cause indirect DoS**: whether the correct functioning of one solution can stop clients from accessing the e-commerce service

In case a criterion is not applicable or assessable from one paper, its corresponding cell is coloured grey and marked with a ⁄. Flexibility and adoptability, were evaluated as strong or weak points. For example, if flexibility is one of the strong points of a solution, its corresponding cell will be coloured green and marked with a △. Conversely, a solution's weak point is noted with a red cell marked with a ▽ and if a criterion is neither a strong nor weak point, its cell is coloured yellow and marked with a ◊.

The imposed latency values reported by the authors, as shown in Table 3.1, are either in percentages or absolute values. We chose to report them as presented by the authors, without attempting to normalise or convert the value from one type to the other, so as to provide an incongruent, yet error-free benchmark. We acknowledge the incongruence could be problematic for assessing the suitability of the solutions, yet for some, normalising or converting values without adding biases or estimations is not feasible. The most notable biases would arise from the heterogeneity of the accessed websites, where some authors used real-world applications for latency measurements, and others, dummy websites. The difference between these categories can be explained by the less complex context the dummy websites operate in, but also by their inconsistent sizes across found solutions. One could, in some instances translate an absolute value to a percentage or vice-versa, yet the large selection of solutions identified hinders the homogenisation of the latency value types through its diverse approach to reporting the values. Although *NAISS* ' value was reported as a percentage for the ease of quantifying an easy-to-comprehend value, the average delays are reported within figures as absolute values such as in Figure 3.12.

***Solution name* / work title**	**Imposed latency**	**True Positive rate**	**Flexibility**	**Adoptability**	**Bypassable by intruder**	**Can cause indirect DoS**
Secure E-commerce Transaction [101]	⁄	⁄	△	◊	yes	no
WS-Security tokens [15]	⁄	⁄	◊	▽	⁄	⁄
An efficient secure electronic payment system for e-commerce [44]	⁄	⁄	◊	◊	yes	⁄
WebShield [58]	15%	⁄	▽	△	no	no
Unified Detection and Response Technology for Malicious Script-Based Attack [97]	960 ms	⁄	▽	△	no	no
Verena [52]	⁄	100%	◊	▽	no	no
HTML integrity authentication based on fragile digital watermarking [57]	⁄	100%	◊	▽	no	yes
Method and apparatus for providing geographically authenticated electronic documents [71]	⁄	⁄	◊	▽	no	⁄

Authenticity and revocation of web content using signed microformats and PKI [73]	/	/	△	▽	no	no
Trusted cloud computing with secure resources and data colouring [48]	/	/	△	◊	no	no
Why HTTPS Is Not Enough - A Signature-Based Architecture for Trusted Content on the Social Web [77]	/	/	▽	◊	/	no
Ensuring Web Integrity through Content Delivery Networks [59]	10%	100%	△	▽	no	yes
Subresource Integrity [92]	/	/	△	△	yes	no
Protecting Users from Compromised Browsers and Form Grabbers [13]	/	/	◊	▽	no	no
Fostering the Uptake of Secure Multiparty Computation in E-Commerce [25]	/	/	◊	◊	no	no
Towards Secure and Dependable Storage Services in Cloud Computing [93]	/	/	△	◊	no	no
Content Threat Removal [95]	/	100%	◊	△	yes	no
Sanitization of Images Containing Stegomalware via Machine Learning Approaches [102]	25 ms/image	80.64%	▽	△	yes	no
A robust defense against Content-Sniffing XSS attacks [35]	/	100%	▽	▽	yes	no
JS-SAN [42]	/	89%	◊	△	yes	no
A signature-based intrusion detection system for web applications based on genetic algorithm [21]	/	100%	▽	◊	no	no

An effective detection approach for phishing websites using URL and HTML features [12]	2-3 s	94.49%	▽	▽	no	yes
A novel approach for analyzing and classifying malicious web pages [46]	/	99%	◊	▽	no	no
Pixastic [90]	/	/	◊	▽	yes	no
Identifying JavaScript Skimmers on High-Value Websites [19]	12%	97.5%	▽	▽	no	no
SpyProxy [66]	600 ms	100%	◊	▽	yes	no
Cujo [55]	500 ms	94%	◊	▽	yes	no
JSSSignature [69]	30 ms	100%	◊	◊	no	no
Content Security Policy [81]	/	100%	◊	△	yes	yes
PHMJ [60]	2.6%	100%	△	△	yes	no
Providing a fast, remote security service using hash lists of approved web objects [24]	/	100%	△	◊	yes	no
MageReport [4]	0	/	▽	△	yes	no
NAISS [83]	**10%**	**100%**	▲	▲	**no**	**no**

TABLE 3.1: Comparison between solutions identified in the literature and *NAISS*

According to this benchmark, *NAISS* situates itself as a relatively low-latency solution, that is efficient in preventing attacks and preserves the core e-commerce service under MageCart attacks. However, we believe *NAISS* has additional improvements over the identified solutions.

3.6.2 Strong points

Let us briefly summarize our findings relating to the improvements brought forward by implementing *NAISS*:

1. Logically sound filtering based on the developer's validation of content. As the results show, the percentage of images reaching the client remains correct under any parameter we tested for.

2. 100% true positive rate, assuming the developer or filter is not compromised. Once an image or web element is validated through a provided signature, any alteration or substitution that effects a change in the byte representation of the image will produce a different signature and hence be spotted by the filter as being different. Moreover, an image is not allowed to be transmitted if its accompanying valid signature is missing.

3. Agnostic of the image steganography technique used. Can stop zero-day techniques. Any e-skimmer using a (novel) image steganography technique used will be caught as long as it relies on modifying any bits of the image (including image metadata).

4. Does not require any client-side modifications. The design of *NAISS* is strictly server-side, but involves developer-side modifications as well. Our testing emulated a simple client connecting to a hosting provider through the *NAISS* filter and showed that no modifications were required on the client-side.

5. Server-side modifications are minimal. It involves the addition and maintenance of a reverse-proxy to the hosting server. In case one is already present, it would require modifications such as filtering is performed on the downstream traffic.

6. The authenticity of received images can be verified on the client-side through the PKI. Moreover, using the public key of the developer, any party can validate whether a signature is correct by learning the public key from a trusted third party.

7. Does not cause a DoS if the attached signatures are altered or removed. Compared to other solutions in the literature, *NAISS* filters only the elements that fail to be authenticated, as opposed to the whole web page. As such, the core e-commerce service can still be available to the customer if an image is denied transmission.

3.6.3 Limitations

When reviewing our work, readers have to be aware of the limitations present within our implementation and testing procedure. Addressing these limitations shall yield an improvement in the proposed solution.

Implementation

Limitations specific to the implementation process can obscure the usefulness of our proof-of-concept as it might not perfectly fit into a real-world e-commerce platform:

- When accessing a website through *NAISS*, the favicon is loaded, but not displayed.
- The communication is done through simple HTTP, although upgrading to HTTPS should not be technically unfeasible.
- An intruder can add multiple incorrect signatures to slow down a website's loading time.
- Does not offer a clear method of protecting the filter against the intruder inside the hosting provider.
- Does not provide a method of safely updating the filter container in the presence of an intruder.
- Currently supports only images, but the extension to all types of MageCart attack vectors is technically feasible.

Testing

These limitations regarding testing might be able to hide or introduce biases into our results:

- Scalable Vector Graphics (SVG) images could not be included due to SteganoGAN incompatibility.
- The ICO images are not loaded from external sources due to incompatibility with online hosting services (e.g. `imgur.com`).
- The Safari browser was not included due to data collection issues.
- No mobile browsers were tested.

3.7 Conclusion

To fill the gaps identified in the literature, we have designed *NAISS* based on the recommendation list in Section 2.2.7. We have devised comprehensive tests and experiments to emulate the effectiveness of *NAISS* under many changing conditions. Despite all these modifications, the filtering process remained sound, while the accompanying performance detriment is comparable to or lower than related works. All in all, our proposed solution is found to be a simple and yet powerful approach for stopping the delivery of stegoimage e-skimmers.

Chapter 4

Conclusions and Future Work

In this report, the significance of the MageCart threat was outlined, especially the challenge of tackling stegoimage e-skimmers to protect e-commerce platform customers. In Chapter 2, we presented the methodology and results of a semi-systematic literature review, which explored the boundaries of the literature to identify suitable solutions for the stegoimage e-skimmers challenge. The analysis and categorisation of these solutions revealed various shortcomings which deem them ineffective or impractical against the stegoimage e-skimmer threat. The analysis additionally yielded a recommendation list (refer to 2.2.7) of features for an effective and practical solution, which served as a foundation for designing our proposed solution in Chapter 3 .

We have proposed a server-side middlebox solution, named *NAISS*, which leverages digital signatures to prevent the transmission of unauthorised images to clients. The authorisation process is fully in the control of the e-commerce platform developer and involves creating a signature for each image on the website, which will then be validated by a middlebox residing between the hosting server and the client. Based on the collected results in Section 3.5, our proof-of-concept implementation demonstrates the effectiveness of *NAISS*, as it is capable of granular filtering 100% of injected stegoimages with minimal impact on loading times. As highlighted in Subsection 3.6.2, *NAISS* is a relevant solution due to its simplicity, efficacy, and ease of implementation, making it a promising approach for mitigating MageCart attacks based on stegoimages.

We conclude our research by pinpointing the answers to our research questions and casting a light towards the future avenues of our research.

4.1 Answered Research Questions

Let us briefly provide and point out the answers to the research questions in Section 1.2. As mentioned in that specific section, **RQ1** is answered by the systematic literature review, while **RQ2** is complementarily answered in the literature review and through the design of NAISS. **RQ3** is bluntly answered through a benchmark.

RQ1 is answered in Section 2.2 and summarized in Table 2.2 and Table 2.3.

RQ2 is answered in Subection 2.2.7 and Subsection 3.6.2.

RQ3 is answered through the benchmark in Table 3.1 from Subsection 3.6.1.

Despite its many advantages, it is important to note that *NAISS* may still face challenges related to its security and deployment, such as the need for additional processes to support its operation. Therefore, further research is needed to evaluate the scalability and practicality of NAISS in real-world scenarios and explore possible enhancements to its design.

4.2 Future directions

We identify future directions for researchers and developers, based on the assumptions and limitations of the literature review and the proof-of-concept implementation.

4.2.1 Literature Review

To enhance the literature review process, we recommend the following improvements for researchers looking to continue or take inspiration from our work:

- The use of a better method to programmatically search more aggregators and follow the chain of citations.
- The better use of advanced Natural Language Processing AIs such as GPT4 for automatic filtering of a large number of search hits.
- Reading reports from security and forensics labs on the MageCart topic to grasp the state-of-art modus operandi and possible defence measures.
- Consider investigating the MageCart families, their modus operandi and impact in the wild to assess which family/group presents the highest risk for the e-commerce market. This could yield better research focused on one particular type of attack.

4.2.2 Proposed solution

Through critical analysis, we conclude that there is room for improvement to elevate the capabilities of *NAISS* such that it becomes industry-ready. Specifically, we indicate the following directions for future research and development:

- Extend *NAISS* and its testing to include more MageCart attack vectors.
- Propose an integrated Continuous Integration/Continuous Deployment (CI/CD) process for the developers to validate and sign web elements.
- Upgrade the connection type to HTTPS.
- Address the slowdown attack by adjusting the filtering algorithm.
- Improve the access time by making use of faster programming languages, algorithms and digital signature schemes.
- Study the security posture of the reverse proxy filter and approaches to better protect it from intruders.
- Improve the testing by running test cases on mobile browsers as well.
- Conduct tests on real-world platforms to test the scalability and adoption process.

The authors believe that the above points of improvement will elevate the contribution of this work and will bring forward an industry-ready solution able to defend e-commerce customers in a *NAISS* manner.

Bibliography

[1] Burpsuite - application security testing software. https://portswigger.net/burp.

[2] Cuckoo - automated malware analysis. https://cuckoosandbox.org/.

[3] Magento digital commerce. https://magento.ca.

[4] Magereport. https://www.magereport.com/.

[5] Owasp zap. https://owasp.org/www-project-zap/.

[6] Stegexpose. https://github.com/b3dk7/StegExpose.

[7] stego-toolkit: Collection of steganography tools. https://github.com/DominicBreuker/stego-toolkit.

[8] Stegspy - steghunter. http://www.spy-hunter.com/stegspy.

[9] Two leading cybersecurity organizations issue joint bulletin on threat of online skimming to payment security. https://www.pcisecuritystandards.org/about_us/press_releases/two-leading-cybersecurity-organizations-issue-joint-bulletin-on-threat-of-online-skim May 2022.

[10] Mehmet Adalier and Antara Teknik. Efficient and secure elliptic curve cryptography implementation of curve p-256. In *Workshop on elliptic curve cryptography standards*, volume 66, pages 2014–2017, 2015.

[11] Ghazi Ayed Alghathian. Website hosting contract. *The Lawyer Quarterly*, 11(4), 2021.

[12] Ali Aljofey, Qingshan Jiang, Abdur Rasool, Hui Chen, Wenyin Liu, Qiang Qu, and Yang Wang. An effective detection approach for phishing websites using url and html features. *Scientific Reports*, 12(1):1–19, 2022.

[13] Sirvan Almasi and William J Knottenbelt. Protecting users from compromised browsers and form grabbers. 2020.

[14] H Alzoubi, M Alshurideh, B Kurdi, K Alhyasat, and T Ghazal. The effect of e-payment and online shopping on sales growth: Evidence from banking industry. *International Journal of Data and Network Science*, 6(4):1369–1380, 2022.

[15] Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach, John Manferdelli, Hiroshi Maruyama, et al. Web services security (ws-security). *Specification, Microsoft Corporation*, 2002.

[16] Murat Aydos, Çiğdem Aldan, Evren Coşkun, and Alperen Soydan. Security testing of web applications: A systematic mapping of the literature. *Journal of King Saud University-Computer and Information Sciences*, 2021.

[17] Charles A Badami. Jrevealpeg: A semi-blind jpeg steganalysis tool targeting current open-source embedding programs. 2021.

[18] Kristoffer Björklund. What's the deal with stegomalware?: The techniques, challenges, defence and landscape, 2021.

[19] Thomas Bower, Sergio Maffeis, and Soteris Demetriou. Identifying javascript skimmers on high-value websites. *Imperial College of Science, Technology and Medicine, Imperial College London*, pages 1–72, 2019.

[20] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[21] Robert Bronte, Hossain Shahriar, and Hisham M Haddad. A signature-based intrusion detection system for web applications based on genetic algorithm. In *Proceedings of the 9th International Conference on Security of Information and Networks*, pages 32–39, 2016.

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[23] Davide Canali, Davide Balzarotti, and Aurélien Francillon. The role of web hosting providers in detecting compromised websites. In *Proceedings of the 22nd international conference on World Wide Web*, pages 177–188, 2013.

[24] Justin Cappos, Nasir Memon, Sai Teja Peddinti, and Keith Ross. Providing a fast, remote security service using hashlists of approved web objects, January 26 2016. US Patent 9,246,929.

[25] Octavian Catrina and Florian Kerschbaum. Fostering the uptake of secure multiparty computation in e-commerce. In *2008 Third International Conference on Availability, Reliability and Security*, pages 693–700. IEEE, 2008.

[26] Rajasekhar Chaganti, Vinayakumar Ravi, Mamoun Alazab, and Tuan D Pham. Stegomalware: A systematic survey of malware hiding and detection in images, machine learningmodels and research challenges. *arXiv preprint arXiv:2110.02504*, 2021.

[27] Jian Chang, Krishna K Venkatasubramanian, Andrew G West, and Insup Lee. Analyzing and defending against web-based malware. *ACM Computing Surveys (CSUR)*, 45(4):1–35, 2013.

[28] Bertil Chapuis, Olamide Omolola, Mauro Cherubini, Mathias Humbert, and Kévin Huguenin. An empirical study of the use of integrity verification mechanisms for web subresources. In *Proceedings of The Web Conference 2020*, pages 34–45, 2020.

[29] Joseph C. Chen. Magecart card skimmers injected into online shops. https://www.trendmicro.com/en_us/research/19/j/

fin6-compromised-e-commerce-platform-via-magecart-to-inject-credit-card-skimmers-into
html, 2019.

[30] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, et al. Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4):1–4, 2015.

[31] Kelsey Clapp. Commodity skimming & magecart trends in first quarter of 2022. https://community.riskiq.com/article/017cf2e6, May 2022.

[32] Docker. Home — docker.com. https://docker.com, 2023.

[33] Huw Fryer, Sophie Stalla-Bourdillon, and Tim Chown. Malicious web pages: What if hosting providers could actually do somethingâĂę. *Computer Law & Security Review*, 31(4):490–505, 2015.

[34] Sean Gallagher. British airways site had credit card skimming code injected. https://arstechnica.com/information-technology/2018/09/british-airways-site-had-credit-card-skimming-code-injected/, Sep 2018.

[35] Misganaw Tadesse Gebre, Kyung-Suk Lhee, and ManPyo Hong. A robust defense against content-sniffing xss attacks. In *6th International Conference on Digital Content, Multimedia Technology and its Applications*, pages 315–320, 2010.

[36] github/pallets. GitHub - pallets/flask: The Python micro framework for building web applications. — github.com. https://github.com/pallets/flask/, 2010.

[37] github/peepw. GitHub - peewpw/Invoke-PSImage: Encodes a PowerShell script in the pixels of a PNG file and generates a oneliner to execute — github.com. https://github.com/peewpw/Invoke-PSImage, 2017.

[38] github/tlsfuzzer. GitHub - tlsfuzzer/python-ecdsa: pure-python ECDSA signature/verification and ECDH key agreement — github.com. https://github.com/tlsfuzzer/python-ecdsa, 2010.

[39] Maria J Grant and Andrew Booth. A typology of reviews: an analysis of 14 review types and associated methodologies. *Health information & libraries journal*, 26(2):91–108, 2009.

[40] Georgina Grant-Muller. 2020 magecart timeline. https://www.rapidspike.com/blog/2020-magecart-timeline/, May 2022.

[41] Massimo Guarascio, Marco Zuppelli, Nunziato Cassavia, Luca Caviglione, and Giuseppe Manco. Revealing magecart-like threats in favicons via artificial intelligence. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery. URL: https://doi.org/10.1145/3538969.3544437.

[42] Shashank Gupta and Brij Bhooshan Gupta. Js-san: defense mechanism for html5-based web applications against javascript code injection vulnerabilities. *Security and Communication Networks*, 9(11):1477–1495, 2016.

[43] Shashank Gupta and Brij Bhooshan Gupta. Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8:512–530, 2017.

[44] Md Arif Hassan, Zarina Shukur, and Mohammad Kamrul Hasan. An efficient secure electronic payment system for e-commerce. *computers*, 9(3):66, 2020.

[45] Lara Heidelberg. *Steganography in the Financial Sector*. PhD thesis, Utica College, 2016.

[46] Panchakshari N Hiremath. *A novel approach for analyzing and classifying malicious web pages*. PhD thesis, University of Dayton, 2021.

[47] Cecilia Hu. The year in web threats: Web skimmers take advantage of cloud hosting and more. https://unit42.paloaltonetworks.com/web-threats-trends-web-skimmers/#Web-Skimmer-Detection-Analysis, Mar 2022.

[48] Kai Hwang and Deyi Li. Trusted cloud computing with secure resources and data coloring. *IEEE Internet Computing*, 14(5):14–22, 2010.

[49] Nav Jagpal, Eric Dingle, Jean-Philippe Gravel, Panayiotis Mavrommatis, Niels Provos, Moheeb Abu Rajab, and Kurt Thomas. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 579–593, 2015.

[50] Tariq Jamil. Steganography: the art of hiding information in plain sight. *IEEE potentials*, 18(1):10–12, 1999.

[51] Segura JÃľrÃťme. Web skimmer hides within exif metadata, exfiltrates credit cards via image files. https://www.malwarebytes.com/blog/news/2020/06/web-skimmer-hides-within-exif-metadata-exfiltrates-credit-cards-via-image-files, June 2020.

[52] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 895–913. IEEE, 2016.

[53] Jonathan Katz. *Digital signatures*, volume 1. Springer, 2010.

[54] Will Keeling. GitHub - wkeeling/selenium-wire: Extends Selenium's Python bindings to give you the ability to inspect requests made by the browser. — github.com. https://github.com/wkeeling/selenium-wire, 2018.

[55] Tammo Krueger and Konrad Rieck. Intelligent defense against malicious javascript code. *PIK-Praxis der Informationsverarbeitung und Kommunikation*, 35(1):54, 2012.

[56] John Leyden. Credit card industry standard revised to repel card-skimmer attacks. https://portswigger.net/daily-swig/credit-card-industry-standard-revised-to-repel-card-skimmer-attacks, Apr 2022.

[57] Bo Li, Wei Li, Yuan-Yuan Chen, Dong-Dong Jiang, and Ying-Zhi Cui. Html integrity authentication based on fragile digital watermarking. In *2009 IEEE International Conference on Granular Computing*, pages 322–325. IEEE, 2009.

[58] Zhichun Li, Yi Tang, Yinzhi Cao, Vaibhav Rastogi, Yan Chen, Bin Liu, and Clint Sbisa. Webshield: Enabling various web defense techniques without client side modifications. In *NDSS*, 2011.

[59] Zhao Xiang Lim, Xiu Qi Ho, Daniel Zhonghao Tan, and Weihan Goh. Ensuring web integrity through content delivery networks. In *2022 IEEE World AI IoT Congress (AIIoT)*, pages 494–500. IEEE, 2022.

[60] Shukai Liu, Xuexiong Yan, Qingxian Wang, Xu Zhao, Chuansen Chai, and Yajing Sun. A protection mechanism against malicious html and javascript code in vulnerable web applications. *Mathematical Problems in Engineering*, 2016, 2016.

[61] LogoAI. AI Logo Maker - Generate your free logo online in minutes! — logoai.com. https://logoai.com/logo-maker, 2020.

[62] Gustav Lundsgård and Victor Nedström. Bypassing modern sandbox technologies. 2016.

[63] MalwareBazaar. Malwarebazaar magecart sample. https://bazaar.abuse.ch/sample/f9274347590156c3e86e\
b7015b6dbd3587de034c51cb52e5161cee671c1107e4/, 2022.

[64] Dimitris Mitropoulos, Panos Louridas, Michalis Polychronakis, and Angelos Dennis Keromytis. Defending against web application attacks: Approaches, challenges and implications. *IEEE Transactions on Dependable and Secure Computing*, 16(2):188–203, 2019. doi:10.1109/TDSC.2017.2665620.

[65] Dimitris Mitropoulos and Diomidis Spinellis. Fatal injection: a survey of modern code injection attack countermeasures. *PeerJ Computer Science*, 3:e136, 2017.

[66] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D Gribble, and Henry M Levy. Spyproxy: Execution-based detection of malicious web content. In *USENIX Security Symposium*, pages 1–16, 2007.

[67] Muralidharan, Trivikram, Aviad, Cohen, Assaf, Nissim, and Nir. The infinite race between steganography and steganalysis in images. *Signal Processing*, page 108711, 2022.

[68] Sangeeta Nagpure and Sonal Kurkure. Vulnerability assessment and penetration testing of web application. In *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, pages 1–6. IEEE, 2017.

[69] Kousha Nakhaei, Fateme Ansari, and Ebrahim Ansari. Jssignature: eliminating third-party-hosted javascript infection threats using digital signatures. *SN Applied Sciences*, 2(1):1–11, 2020.

[70] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[71] Timothy G Nye. Method and apparatus for providing geographically authenticated electronic documents, June 19 2007. US Patent 7,233,942.

[72] OpenAI. OpenAI API DaVinci — platform.openai.com. https://platform.openai.com/docs/models/davinci, 2021. [Accessed 29-Mar-2023].

[73] Henrich C Pöhls. Authenticity and revocation of web content using signed microformats and pki. 2007.

[74] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, 2014.

[75] Python. Welcome to Python.org — python.org. https://python.org, 2023.

[76] Malik Qasaimeh, Nancy Abu Halemah, Rana Rawashdeh, Raad S Al-Qassas, and Abdallah Qusef. Systematic review of e-commerce security issues and customer satisfaction impact. In *2022 International Conference on Engineering & MIS (ICEMIS)*, pages 1–8. IEEE, 2022.

[77] Matthias Quasthoff, Harald Sack, and Christoph Meinel. Why https is not enough–a signature-based architecture for trusted content on the social web. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)*, pages 820–824. IEEE, 2007.

[78] Joaquim Radua, Valentina Ramella-Cravaro, John PA Ioannidis, Abraham Reichenberg, Nacharin Phiphopthatsanee, Taha Amir, Hyi Yenn Thoo, Dominic Oliver, Cathy Davies, Craig Morgan, et al. What causes psychosis? an umbrella review of risk and protective factors. *World psychiatry*, 17(1):49–66, 2018.

[79] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. Security certification in payment card industry: Testbeds, measurements, and recommendations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 481–498, 2019.

[80] Shailendra Rathore, Pradip Kumar Sharma, and Jong Hyuk Park. Xssclassifier: an efficient xss attack detection approach based on machine learning classifier on snss. *Journal of Information Processing Systems*, 13(4):1014–1028, 2017.

[81] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 12 angry developers-a qualitative study on developers' struggles with csp. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3085–3103, 2021.

[82] Phoebe Rouge, Christina Yeung, Daniel Salsburg, and Joseph A Calandrino. Checkout checkup: Misuse of payment data from web skimming, 2020.

[83] Catalin Rus. GitHub - ruscatalin/NAISS: Research Project for NAISS: Network Authentication of Images to Stop e-Skimmers — github.com. https://github.com/ruscatalin/NAISS, 2023.

[84] Sansec. Lockdown: Stores closed, online stores hacked. https://sansec.io/research/magecart-corona-lockdown, June 2020.

[85] Selenium. WebDriver — selenium.dev. https://selenium.dev/documentation/webdriver/, 2021.

[86] Mandar Prashant Shah. *Comparative analysis of the automated penetration testing tools*. PhD thesis, Dublin, National College of Ireland, 2020.

[87] Similarweb. Top Desktop Browsers Market Share for February 2023. https://www.similarweb.com/browsers/worldwide/desktop/, 2023.

[88] Xinh Studio. Payment method icons by Xinh Studio — iconfinder.com. https://iconfinder.com/iconsets/payment-method, 2013.

[89] Jin Chen Taojie Wang and Tao Yan. A new web skimmer campaign targets real estate websites through attacking cloud video distribution supply chain. https://unit42.paloaltonetworks.com/web-skimmer-video-distribution/, January 2022.

[90] P Thiyagarajan, Gnanasekaran Aghila, and V Prasanna Venkatesan. Pixastic: steganography based anti-phihsing browser plug-in. *arXiv preprint arXiv:1206.2445*, 2012.

[91] Nees Jan Van Eck and Ludo Waltman. Vos: A new method for visualizing similarities between objects. In *Advances in Data Analysis: Proceedings of the 30 th Annual Conference of the Gesellschaft für Klassifikation eV, Freie Universität Berlin, March 8–10, 2006*, pages 299–306. Springer, 2007.

[92] W3C. Subresource integrity. https://www.w3.org/TR/SRI/, 2016.

[93] Cong Wang, Qian Wang, Kui Ren, Ning Cao, and Wenjing Lou. Toward secure and dependable storage services in cloud computing. *IEEE transactions on Services Computing*, 5(2):220–232, 2011.

[94] Joel Weinberger, Adam Barth, and Dawn Song. Towards client-side {HTML} security policies. In *6th USENIX Workshop on Hot Topics in Security (HotSec 11)*, 2011.

[95] Simon Wiseman. Content security through transformation. *Computer Fraud & Security*, 2017(9):5–10, 2017.

[96] Simon Wiseman. Stegware–using steganography for malicious purposes. 2017.

[97] Soojin Yoon, Hyun-lock Choo, Hanchul Bae, and Hwankuk Kim. Unified detection and response technology for malicious script-based attack. *International Journal of Research Studies in Computer Science and Engineering (IJRSCSE)*, 3, 2016.

[98] EV Zenkina. About current trends in global e-commerce. *BENEFICIUM Đ¡ĐřÑČÑĞĐ¡Đ¿Đţ Đ£ĐţÑĂĐÿĐ¿ĐťĐÿÑĞĐţÑĄĐžĐ¿Đţ ÑĄĐţÑĆĐţĐšĐ¿Đţ ĐÿĐůĐťĐřĐ¡ĐÿĐţ*, (1):68–73, 2022.

[99] Bing Zhang, Jingyue Li, Jiadong Ren, and Guoyan Huang. Efficiency and effectiveness of web application vulnerability detection approaches: A review. *ACM Computing Surveys (CSUR)*, 54(9):1–35, 2021.

[100] Kevin Alex Zhang, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Steganogan: High capacity image steganography with gans. *arXiv preprint arXiv:1901.03892*, 2019. URL: https://arxiv.org/abs/1901.03892.

[101] Qin Zheng, Shundong Li, Yi Han, Jinchun Dong, Lixiang Yan, and Jun Qin. Security technologies in e-commerce. In *Introduction to E-commerce*, pages 135–168. Springer, 2009.

[102] Marco Zuppelli, Giuseppe Manco, Luca Caviglione, and Massimo Guarascio. Sanitization of images containing stegomalware via machine learning approaches. In *ITASEC*, pages 374–386, 2021.