A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web

Stefano Calzavara
Università Ca' Foscari

Sebastian Roth
CISPA Helmholtz Center for Information Security
Saarbrücken Graduate School of Computer Science

Alvise Rabitti
Università Ca' Foscari

Michael Backes
CISPA Helmholtz Center for Information Security

Ben Stock
CISPA Helmholtz Center for Information Security

Abstract

Click-jacking protection on the modern Web is commonly enforced via client-side security mechanisms for framing control, like the X-Frame-Options header (XFO) and Content Security Policy (CSP). Though these client-side security mechanisms are certainly useful and successful, delegating protection to web browsers opens room for inconsistencies in the security guarantees offered to users of different browsers. In particular, inconsistencies might arise due to the lack of support for CSP and the different implementations of the underspecified XFO header. In this paper, we formally study the problem of inconsistencies in framing control policies across different browsers and we implement an automated policy analyzer based on our theory, which we use to assess the state of click-jacking protection on the Web. Our analysis shows that 10% of the (distinct) framing control policies in the wild are inconsistent and most often do not provide any level of protection to at least one browser. We thus propose recommendations for web developers and browser vendors to mitigate this issue. Finally, we design and implement a server-side proxy to retrofit security in web applications.

1 Introduction

The Web is the largest distributed system in the world, and it boasts an incredible variety and complexity. Unfortunately, complexity is where attackers lurk. To assist developers in securing their applications, the Web platform has evolved to support more and more server-sent, yet *client-enforced* security mechanisms. This approach is appealing because it offers uniform and well-thought defenses to as many Web developers as possible. Examples of popular client-side security mechanisms include Content Security Policy (CSP) [25], cookie security attributes [3], and HSTS [11].

Although client-side security mechanisms are undoubtedly useful and successful [23], delegating protection to Web browsers might introduce *inconsistencies* in the security guarantees offered to users of different browsers. The most obvious case is when legacy browsers access Web applications, but problems might also arise when the same defense is implemented differently across modern browsers [21]. In this paper, we are concerned about inconsistencies in *framing control*, a cornerstone of Web application security, which pioneered the adoption of client-side security mechanisms.

Framing control constrains the inclusion of Web content inside *iframes* (sub-documents) opened by malicious pages and it is particularly useful to prevent *click-jacking* attacks [7]. The original defense against click-jacking back in the days was the use of JavaScript-based *frame busters*. These scripts, placed in pages for which framing should be forbidden, merely checked conditions like `self == top` to assess whether they were loaded in the top-most frame. If not, they would navigate the top frame away. Unfortunately, researchers showed that this solution was often ineffective [20]. In 2009, Internet Explorer introduced the `X-Frame-Options` header (XFO) as a simple, browser-based mechanism to control framing without relying on JavaScript. This header gained extensive popularity and was quickly adopted by all the other major browsers. Unfortunately, since XFO was not standardized a priori, different browser vendors provided different implementations, leading to differing support of its directives and attacks like *double framing* in some browsers [20]. In 2014, the second iteration of the CSP specification introduced the `frame-ancestors` directive to control framing, with the goal of obsoleting XFO and to offer a comprehensive, uniform protection mechanism for all CSP-compliant browsers.

The way in which the Web platform evolved hints at the fact that the state of click-jacking protection on the Web is brittle. Most browsers provide two different defenses in the form of XFO and CSP, possibly with different implementations, and developers may choose to use any of these two mechanisms, or a combination thereof, to protect their Web applications. Given such complexity and the diverse levels of support for framing control, this potentially gives rise to inconsistencies. In this paper, we conduct a comprehensive study of the differing behavior of major browsers and introduce and apply a simple formal framework to study the problem in the wild.

Contributions. We make the following contributions:

1. we introduce a formal framework designed to rigorously study the problem of inconsistencies in framing control, based on existing work on the CSP semantics [4]. We use this framework to formalize the notion of policy consistency and to observe that not every inconsistency is equally dangerous. We thus propose more relaxed definitions which admit limited types of inconsistencies and might be justified by how the Web platform has been evolving (Section 3);

2. we develop a policy analyzer (dubbed FRAMECHECK) based on the proposed theory, which enables an automated security assessment of the state of click-jacking protection on a given Web page. Our implementation leverages a comprehensive set of test cases designed to understand how existing browsers implement the loosely specified XFO header. The test cases are of independent interest since they highlight potentially dangerous practices in major browsers (Section 4);

3. we run FRAMECHECK on policies collected from 10,000 popular websites from the Tranco list [18] and we assess their effectiveness. Our experiments show that 10% of the (de-duplicated) policies are inconsistent. Hence we carry out a systematic analysis of the main causes of inconsistency and their practical import. We also discuss the impact of the selected browsers on the results of our study, reasoning on the road forward for click-jacking protection (Section 5);

4. we present recommendations for developers and browser vendors to mitigate the dangers of the framing control inconsistencies that are currently affecting the Web. We also design and implement a server-side proxy to retrofit security in existing Web applications, which we release as open-source software (Section 6).

Artifact Availability. In the interest of open science, we make both our server-side proxy and the FRAMECHECK core available online.[1,2]

2 Background

In this section, we review framing-based attacks and the most popular client-side defense mechanisms against them.

2.1 Framing-based Attacks

The nature of HTML and CSS allows the developers of a Web site fine-grained control over how elements are placed and shown in the browser. This feature, which is one of the core pillars of the Web's success, can, however, be abused by attackers to their advantage. In particular, an attacker can trick their victims into clicking elements in another Web application. One popular example is the so-called *like-jacking* attack on social networks. Here, an attacker creates a page with an element a user is likely to click, e.g., a button promising some premium content. Then, the attacker adds an iframe pointing to a page with a *Like* button (e.g., from Facebook) at the same coordinates, and use CSS to make the iframe fully transparent. When the user tries to click the button for the premium content, she unknowingly clicks into the frame, inadvertently invoking the like functionality. In general, we refer to such attacks where the adversary lures the victim into unknowingly clicking a link on a different page as *click-jacking*.

2.2 X-Frame-Options

Starting from 2009, browser vendors picked up on the increasing danger of click-jacking and similar attacks, and Internet Explorer was the first browser to implement the so-called `X-Frame-Options` (XFO) header [9]. This header allows a site to control which other origins may frame it. At that time, Firefox and Internet Explorer supported three different directives for the XFO header: *SAMEORIGIN* to allow framing only from pages with the same origin (i.e., protocol, host, and port), *ALLOW-FROM origin* to selectively allow framing from a single origin or *DENY* to block framing completely.

Importantly, although an XFO specification exists in the form of RFC 7034 [9], that specification was written after various browsers had implemented XFO and notes that "not all browsers implement X-Frame-Options in exactly the same way, which can lead to unintended results". In particular, the ALLOW-FROM directive is not universally supported by all browsers: most importantly, all Chromium derivates do not understand this directive. Additionally, browsers might implement SAMEORIGIN (and ALLOW-FROM) differently because the origin check for framing can be performed in different ways. According to the specification, the check can be based "on the origin of the framed page and the top-level browsing context", on "the framed page and the framing page", or on "the whole chain of nested frames in between". When the XFO specification was originally written, the first practice was the most common, yet such implementation is potentially insecure because it opens the way to *double framing* attacks, where the attacker relies on multiple nested frames to circumvent existing defense mechanisms [20].

Overall, we find that XFO is indeed inconsistently implemented across browsers. We dive deeper into the actual inconsistencies and their impact in Section 4.2.

2.3 Content Security Policy

Given the problems of the underspecified XFO header, the Web security community proposed to incorporate framing

[1] https://github.com/cispa/framing-control-proxy
[2] https://github.com/cispa/framing-control-analytics

control into Content Security Policy (CSP). While initially meant as a means of mitigating injection attacks, CSP nowadays offers support for framing control and TLS enforcement as well. As a recent study has shown, CSP is equally widely used for these use cases as it is for its original purpose [19].

In particular, framing control in CSP can be enforced through the `frame-ancestors` directive. This solution has a clear advantage over XFO due to its standardized support and additional expressiveness. First, as the name suggests, the `frame-ancestors` directive performs the origin check for framing based on *the whole chain* of nested frames (ancestors) between the top-level browsing context and the framed page, which offers the strongest security guarantees by ruling out double framing. Moreover, CSP is strictly more expressive than XFO, since it can take advantage of the full CSP syntax, which allows one to whitelist an arbitrary (possibly empty) list of origins. For example, the DENY directive of XFO can be simulated by setting the `frame-ancestors` directive to `'none'`, while the SAMEORIGIN directive can be simulated by setting it to `'self'`. Even better, CSP can be easily used to whitelist all subdomains of given domains, e.g., `frame-ancestors *.foo.com *.bar.com`, which cannot be expressed through XFO. Hence, administrators have an easier job at maintaining a whitelist of sites through CSP; achieving the same through XFO is only possible by checking the Referer header of incoming HTTP requests. This header is sent by browsers and indicates the document which initiated the loading of a specific resource (in this case, an iframe). Hence, this can be combined with server-side logic to check the transmitted header against a whitelist, and respond with a corresponding ALLOW-FROM header. We refer to this mechanism as *Referer sniffing*.

In this paper, we refer to browsers supporting framing control via CSP as *modern* browsers; we deem all the other browsers as *legacy*. According to the CSP specification, modern browsers must ignore the XFO headers in the presence of a CSP, which includes a `frame-ancestors` directive. At the same time, however, XFO is still the only way for a site to control framing in legacy browsers. Given the difference in expressiveness between the two types of security mechanisms, this can cause inconsistencies when visiting the same page with different browsers.

3 Formal Framework

In this section, we lay the theoretical grounds for our research by formalizing the notion of policy consistency. We then observe that not every inconsistency is equally dangerous and propose more relaxed definitions which admit limited types of inconsistencies. We also argue why these definitions are of practical interest by taking into account the current state of the Web platform and its evolution.

Schemes	s	::=	`http` | `https`
Host Expressions	h	::=	`*` | `*.string` | `string`
Source Expressions	e	::=	`'self'` | s | h | (s,h)
Directive Values	v	::=	$\{e_1,\ldots,e_n\}$

Table 1: Syntax of CoreCSP

3.1 Policy Semantics

Since CSP is more expressive than XFO, it is straightforward to translate every XFO policy into an equivalent CSP policy. Hence, we can define the semantics of every framing control policy on top of the CoreCSP framework, which provides a simple denotational semantics for the content restriction fragment of CSP [4]. In particular, one can interpret the set of origins from which framing is allowed using *source expressions*, i.e., a sort of regular expressions representing a set of origins. The semantics of a framing control policy is then given by a *directive value*, i.e., a set of source expressions defining the origins where framing is allowed. The productions in Table 1 define the main syntactic categories of CoreCSP used in the present section. Note that, though relatively close to the original CSP syntax, CoreCSP abstracts from several details, which can still be easily modeled. For example, the `'none'` source expression of CSP is represented by the directive value $\emptyset$ (framing is not allowed anywhere).

To understand how the CoreCSP denotational semantics is defined, assume that `http://www.foo.com` deploys the following CSP:

```
frame-ancestors *.foo.com https://*
```

Since the protected page is served over HTTP, the semantics of the policy is formalized by the directive value $\{(\texttt{http},\texttt{*.foo.com}),(\texttt{https},*)\}$. However, note that this assumes the use of a modern browser since any legacy browser which does not support CSP will ignore the policy and enforce no framing restriction. This can be modeled by giving the semantics of the policy in terms of the more liberal directive value $\{(\texttt{http},*),(\texttt{https},*)\}$.

More generally, since the same policy might be enforced differently by different browsers and the same Web page may also send different policies to different user agents, we let $[\![w]\!]_b$ stand for the directive value representing the framing restrictions enforced on the page w by the browser b. We postpone to Section 4 the definition of $[\![\cdot]\!]_{\cdot}$ for the browsers of interest and develop a general theory in the present section.

3.2 Formal Definitions

We build on CoreCSP because directive values can be ordered by a relation $\sqsubseteq$ such that $v_1 \sqsubseteq v_2$ if and only if the set of origins represented by v_1 is contained in the set of origins

represented by v_2 [4]. CoreCSP allows us to readily formalize the intuition of a consistent policy, i.e., a policy that enforces the same restrictions across all browsers.

Definition 1 (Consistent Policy)**.** The policy of the Web page w is *consistent* for the set of browsers B if and only if, for all $b_1, b_2 \in B$, we have $[\![w]\!]_{b_1} \sqsubseteq [\![w]\!]_{b_2}$ and $[\![w]\!]_{b_2} \sqsubseteq [\![w]\!]_{b_1}$.

Example 1. Consider a Web site which only relies on XFO for framing control, specifying the policy:

```
ALLOW-FROM https://www.example.com
```

This policy is inconsistent, because it restricts framing in Edge, but leaves Chrome users completely unprotected.[3] To improve protection, the Web site might then additionally specify a CSP of the following form:

```
frame-ancestors https://www.example.com
```

The revised framing control policy is consistent for Edge and Chrome since CSP takes precedence over XFO. Hence, the users of these two browsers are equally protected.

Though consistency is undoubtedly a desirable property of policies, there might be practical reasons why real-world framing control policies are inconsistent. In particular, the limited expressiveness of XFO complicates the deployment of useful policies, which instead are trivial to specify using CSP, e.g., enabling framing from multiple origins or arbitrary subdomains of a trusted domain. Operators can work around this limitation by shipping different ALLOW-FROM directives to different pages through Referer sniffing, yet this requires the implementation of additional logic. We thus see pragmatic reasons why XFO and CSP headers might contain mismatches leading to inconsistencies, but (luckily) we also notice that not all the inconsistencies are equally dangerous. We provide an example below.

Example 2. Assume that `https://www.example.com` only relies on CSP for framing control, specifying the policy:

```
frame-ancestors https://*.example.com
```

This policy is inconsistent, because it restricts framing in Chrome, but does not protect the users of legacy browsers without CSP support. To improve protection, the Web site might then additionally specify an XFO policy of the form:

```
SAMEORIGIN
```

The revised policy is still inconsistent, yet it provides tighter security than the original one and is straightforward to deploy, so it might be more appealing for Web developers. Note that since the XFO policy is less permissive than the CSP policy, this might lead to compatibility issues in legacy browsers, e.g., if framing is required from `https://mail.example.com`, yet users of such browsers are protected against click-jacking.

[3] For details on the exact support for XFO and CSP in major browsers, see Section 4.2.

By elaborating on the previous example, we identify a new class of policies that has a useful property: legacy browsers are all in agreement on how the policy should be enforced, all modern browsers also share the same policy interpretation, but legacy browsers might be more conservative than modern browsers. This ensures that users of legacy browsers are protected and that no inconsistency arises among users of modern browsers, yet users of legacy browsers might be affected by compatibility issues. Formally, this is formulated by the following definition.

Definition 2 (Security-Oriented Policy)**.** The policy of the Web page w is *security-oriented* for the set of browsers B if and only if it is possible to partition B in two sets B_l, B_m such that all these properties hold true:

- B_l only includes legacy browsers and B_m only includes modern browsers;
- the policy of w is consistent for both B_l and B_m;
- for all $b_1 \in B_l$ and $b_2 \in B_m$ we have $[\![w]\!]_{b_1} \sqsubseteq [\![w]\!]_{b_2}$.

The last class of policies we consider still arises from the expressiveness gap between XFO and CSP yet makes the opposite choice of security-oriented policies: while it is still true that legacy browsers all give the same semantics to the policy, as well as modern browsers, the policy interpretation given by legacy browsers might be more liberal than one of the modern browsers. This ensures that users of legacy browsers can access the Web application without compatibility issues and that no inconsistency arises among users of modern browsers. Nevertheless, users of legacy browsers might be left unprotected.

Definition 3 (Compatibility-Oriented Policy)**.** The policy of the Web page w is *compatibility-oriented* for the set of browsers B if and only if it is possible to partition B in two sets B_l, B_m such that all these properties hold true:

- B_l only includes legacy browsers and B_m only includes modern browsers;
- the policy of w is consistent for both B_l and B_m;
- for all $b_1 \in B_l$ and $b_2 \in B_m$ we have $[\![w]\!]_{b_2} \sqsubseteq [\![w]\!]_{b_1}$.

Example 3. The original policy of Example 2 is inconsistent, yet compatibility-oriented. It is an insecure policy, but it might be a plausible choice for Web developers who are particularly concerned about compatibility with legacy browsers not supporting CSP, where no restriction is actually enforced. Instead, the original policy of Example 1 is not even compatibility-oriented, since two modern browsers like Chrome and Edge give different interpretations to the policy, due to Chrome's lack of support for ALLOW-FROM.

To summarize, we argue that consistency is the most desirable property for framing control policies since it implies the same policy interpretation in all browsers. Security-oriented policies can offer a proper level of protection on legacy browsers but might introduce compatibility issues with them. Compatibility-oriented policies might sacrifice protection on legacy browsers, but are backward compatible with them and thus potentially appealing to Web developers. Observe that a policy is consistent if and only if it is both security-oriented and compatibility-oriented.

Inconsistent policies which are neither security-oriented nor compatibility-oriented are generally hard to justify as correct because they fall in one of the following cases:

- two legacy browsers interpret the policy differently;
- two modern browsers interpret the policy differently;
- none of the above is true, yet legacy browsers and modern browsers give two incomparable interpretations of the same policy.

We refer to such policies as *unduly inconsistent*.

4 Policy Analyzer

We designed and implemented FRAMECHECK, an automated analyzer of framing control policies based on our theory. Given a URL to analyze, FRAMECHECK produces a security report on its state of click-jacking protection. We explain the details of the analyzer in the rest of this section.

4.1 FRAMECHECK Description

Our tool is parametric with respect to a set of browsers B. Each browser $b \in B$ is characterized by two ingredients:

1. its user-agent string UA_b, defining how the browser presents itself to Web applications;
2. the semantics $[\![\cdot]\!]_b$, expressed as a function translating a list of HTTP headers into a directive value of CoreCSP.

The user-agent string UA_b can be easily found by inspecting the HTTP requests sent by the browser, e.g., using the developers' tools. At the same time, the semantics $[\![\cdot]\!]_b$ can be identified either by manual source code inspection (in the case of open-source browsers) or by reverse-engineering.

Our implementation supports the 10 most popular browsers according to data from *Can I Use*.[4] For each browser, we downloaded the latest available version with at least 1% of market share[5] and we reverse-engineered its semantics through an exhaustive set of test cases (see Section 4.2). The set of browsers under study is shown in Table 2: only two browsers do not support framing control via CSP, i.e., Internet Explorer and Opera Mini, which we deem as legacy. Note that, according to *Can I Use*, Opera Mini does not support any mechanism for framing control. However, we installed the latest available version from the Google Play Store, and, according to our tests, Opera Mini, in fact, supports XFO.

Browser Name	Type	Version	Market
Chrome	Desktop	76	$\sim$ 23%
Chrome for Android	Mobile	76	$\sim$ 35%
Edge	Desktop	18	$\sim$ 2%
Firefox	Desktop	69	$\sim$ 4%
Internet Explorer	Desktop	11	$\sim$ 2%
Opera Mini	Mobile	44.1	$\sim$ 1%
Safari	Desktop	12.3	$\sim$ 2%
Safari for iOS	Mobile	12.3	$\sim$ 10%
Samsung Internet	Mobile	10.1	$\sim$ 3%
UC Browser	Mobile	12.12	$\sim$ 3%

Table 2: Browsers considered in the present study

Given a Web page w to analyse, FRAMECHECK first accesses w once for each $b \in B$, sending the corresponding user-agent string UA_b. Since w may redirect requests from different browsers to different landing pages, e.g., to provide a mobile-friendly variant of the page, this process eventually identifies a set of pairs of the form (B_i, w_i), where $B_i \subseteq B$ and w_i is the landing page of w for each $b_j \in B_i$. For each identified pair (B_i, w_i), FRAMECHECK computes $[\![w_i]\!]_{b_j}$ for each $b_j \in B_i$ and produces a security report on policy consistency based on the definitions in Section 3.

4.2 Test Cases

In total, we developed more than 40 test cases to reconstruct the semantics of the underspecified XFO header in our set of browsers. We designed the test cases through a careful analysis of the XFO specification [9] and a preliminary inspection of a large set of framing control policies collected in the wild by a simple crawler. Hence, the test cases are not esoteric examples of problems that might possibly arise in theory, but rather represent classes of potentially ambiguous policies that we observed in practice. We report below on the most interesting findings.

4.2.1 Support for ALLOW-FROM

Though it is widely known that Chrome does not support ALLOW-FROM, it turns out that only 3 out of 10 browsers actually support this XFO directive: Edge, Firefox[6] and Internet Explorer. This means that every Web page which adopts

[4] https://caniuse.com

[5] Note that Chrome derivates like Brave also show their UA as Chrome, leading to a slight over-approximation of Chrome usage.

[6] During our project, Firefox dropped support for ALLOW-FROM in version 70. We discuss the impact of this recent change in Section 5.4.

the ALLOW-FROM directive, but does not deploy a corresponding CSP, implements inconsistent protection against click-jacking and leaves (at least) 7 browsers unprotected.

We also tested what happens when the ALLOW-FROM directive is not followed by a valid serialized origin (e.g., `https://example.com`), as mandated by the XFO specification. In all the cases we tested, the browser implementations were conservative and denied framing, thus behaving as in the case of the DENY directive. There is one exception to this rule, though: Edge also supports the use of ALLOW-FROM with a hostname like `example.com` (without scheme). The corresponding interpretation is the following: if the policy is applied to an HTTP page, framing is allowed from `example.com` over both HTTP and HTTPS; if instead, the policy is applied to an HTTPS page, framing is only allowed from `https://example.com`. This interpretation is sensible from a security perspective because it mimics the behavior of source expressions in the CSP specification. However, it is worth noting that this introduces room for inconsistencies with other browsers, where framing is denied if the provided value is not a proper origin.

4.2.2 Support for Multiple Headers

When the same Web page sends multiple XFO headers, most of the tested browsers simultaneously enforce all of them: this is the case for 7 out of 10 browsers. Unfortunately, we observed that Edge, Internet Explorer and Opera Mini only enforce the first header and discard the other ones, which might lead to inconsistencies. For example, consider the following two headers:

```
X-Frame-Options: SAMEORIGIN
X-Frame-Options: DENY
```

This policy prevents framing in most browsers, since two directives are simultaneously enforced, and one of them denies framing. However, this policy allows same-origin framing in Edge, Internet Explorer and Opera Mini. Observe that this policy would not have been inconsistent if the two headers had been swapped.

4.2.3 Parsing of Header Values

The HTTP protocol specification in RFC 7230 mandates that it must be possible to replace multiple headers with the same name with a single header that includes a comma-separated list of the header values [8]. Therefore, the standard implies that browsers must be able to handle headers of the following form correctly:

```
X-Frame-Options: SAMEORIGIN, DENY
```

This policy prevents framing in most browsers since it is interpreted as two headers, one of which denies framing (see above). However, we discovered unexpected behaviors in 3 browsers: Edge, Internet Explorer and Opera Mini. In particular, we observed that these browsers do not split the header value on commas and rather parse the list as a single value, which is interpreted as a non-existing directive, i.e., not enforcing any framing restriction. This also happens when the *same* directive is repeated multiple times, such as in the case of `DENY, DENY`. This behavior has a particularly subtle implication on the interpretation of policies like:

```
X-Frame-Options: ALLOW-FROM <orig1>, <orig2>
```

Firefox parses this policy as two separate headers, one allowing framing from the first origin and the other one containing an incorrect value, which does not enforce any framing restriction: as a result, framing is only allowed from the first origin. Internet Explorer, instead, blocks every form of framing, since ALLOW-FROM is not set to a serialized origin. Remarkably, none of these two interpretations matches what the Web developer likely had in mind, i.e., whitelisting two different origins.

4.2.4 Double Framing Protection

Finally, we observed that most browsers implement XFO in a way that is robust against double framing attacks. This shows that current implementation practices had improved since the original XFO specification when all browsers used to perform origin checks for framing based on the top-level browsing context alone [9]. However, there are still 3 browsers that are susceptible to double framing attacks: Edge, Internet Explorer, and UC Browser.

In the rest of the paper, we do not consider inconsistencies arising from double framing, because otherwise even trivial XFO policies like `SAMEORIGIN` would be considered inconsistent and bias our study. This also implies that we do not need to take the full browsing context into account when defining the semantics of framing control policies in our framework, which is useful to keep the presentation simple.

4.2.5 Summary

The summary of our analysis is shown in Table 3. Based on our extensive set of test cases, we identified 6 different semantics across the 10 browsers we considered, without counting the unexpected support for hostnames in ALLOW-FROM implemented in Edge: this means that the room for inconsistent click-jacking protection is significant. Out of the 10 tested browsers, Firefox 69 is the only one that faithfully implements the specifications we checked, while Opera Mini offers little to no protection against click-jacking, because it does not implement CSP, it does not support ALLOW-FROM, and even basic XFO directives like SAMEORIGIN and DENY can be incorrectly enforced due to other quirks in the treatment of HTTP headers.

Browser	CSP	ALLOW-FROM	Multiple Headers	Header Parsing	Double Framing
Chrome	✓	✗	✓	✓	✓
Chrome for Android	✓	✗	✓	✓	✓
Edge	✓	✓	✗	✗	✗
Firefox	✓	✓	✓	✓	✓
Internet Explorer	✗	✓	✗	✗	✗
Opera Mini	✗	✗	✗	✗	✓
Safari	✓	✗	✓	✓	✓
Safari for iOS	✓	✗	✓	✓	✓
Samsung Internet	✓	✗	✓	✓	✓
UC Browser	✓	✗	✓	✓	✗

Table 3: Framing control semantics of popular browsers

5 Analysis in the Wild

In this section, we report on a large-scale analysis performed in the wild with our policy analyzer. Our analysis shows that many popular Web sites implement inconsistent protection against click-jacking and sheds light on the root causes of this potential security problem.

5.1 Data Collection

To assess inconsistencies at scale, we decided to analyze the top 10,000 sites from the Tranco list of October 29, 2019. As we did not only want to check the start pages in a static manner, we instead used a Chrome-based crawler to visit the start pages, collect all links on them, and follow those links up to at most 500 items per site. (Here, "site" refers to the registrable domain name or eTLD+1.) In doing so, we did not only collect the headers delivered with the pages we visited, but also those of all iframes on the visited pages. This way, we were able to (partially) account for sites where only specific pages are protected against framing-based attacks. We then retrieved the XFO and CSP headers of the collected URLs, sending each request to a URL once for each of the different user-agent strings considered in our study.

For this step, we primarily relied on Python's Requests library to collect data. However, Requests folds multiple response headers with the same name into a comma-separated list, as specified in RFC 7230 [8]. As discussed in Section 4.2, browsers do not necessarily follow this specification, but might rather consume each header separately, meaning that Requests' approach to parsing headers would not properly account for that. Therefore, in case we detect a comma in either the XFO or CSP header, we fall back to curl, which outputs the headers line-by-line. To further improve resiliency against possible crawling errors, we filtered out from the dataset all the pages where we observed that at least one user agent was not receiving the XFO or CSP headers, while other user agents were. Though this might lose some inconsistencies, e.g., when CSP headers are not actually sent to legacy browsers, we preferred to be conservative and work on more reliable data rather than risking to unduly exacerbate the number of inconsistencies in the wild. In particular, we found that several pages did not consistently deliver the same XFO and/or CSP headers, even when visited multiple times with the same User-Agent string. Finally, we performed a de-duplication of the collected framing control policies by removing all the duplicate combinations of XFO and CSP policies collected within the same origin, to avoid biasing the dataset construction towards origins with hundreds of pages all using the same policy.

At the end of the data collection process, we visited 989,875 URLs overall. Of those, 369,606 URLs (37%) across 5,835 sites carried either an XFO or CSP header aimed at framing control. After the dataset cleaning and the de-duplication process explained above, we were left with 17,613 framing control policies. Table 4 shows the adoption of the different security mechanisms in the different policies. We observe that XFO is still the most widespread defense mechanism against click-jacking in the wild by far, yet around 12% of the collected policies make use of CSP.

5.2 Inconsistent Policies

Overall, we identified 1,800 policies from 1,779 origins implementing inconsistent protection against click-jacking, i.e., where the enforced level of protection is dependent on the browser. This is 10% of the analyzed policies, which is already a significant percentage. But this result becomes even more concerning when we take a look at which click-jacking protection mechanisms are used by such policies.

Defense	Number of Policies	Percentage
Just XFO	15,415	88%
Just CSP	714	4%
XFO + CSP	1,484	8%

Table 4: Defenses used in the collected policies

Defense	Inconsistencies	Percentage
Just XFO	290	16%
Just CSP	705	39%
XFO + CSP	805	45%

Table 5: Defenses used in the inconsistent policies

Table 5 provides the breakdown: the relative majority of the inconsistencies (45%) occur when XFO and CSP are used together, which suggests that having two different mechanisms for the same purpose is potentially dangerous. Moreover, note that 805 out of the 1,484 pages (54%) which make use of both XFO and CSP together implement inconsistent protection against click-jacking, i.e., it is more likely to get the combination of the two defenses wrong than right.

Another interesting insight from our analysis is that 84% of the inconsistent policies make use of CSP. Intuitively, this seems related to the fact that the set of browsers we consider includes some legacy browsers without CSP support: in particular, Opera Mini provides very limited tools to protect against click-jacking. Hence, one might think that inconsistencies are motivated by its presence alone, yet this is not the case: if we removed Opera Mini from the set of browsers, the number of inconsistent policies would drop from 1,800 to 1,749, which is roughly a 3% reduction. One might then try to also remove Internet Explorer from the picture, since it also lacks support for CSP. However, this is a different story than Opera Mini, since Internet Explorer supports the ALLOW-FROM directive. Hence, inconsistencies could be fixed by simulating the behavior of CSP through different values of ALLOW-FROM based on the Referer header (see Section 2).

To understand the prevalence of such practice in the wild, we set up the following experiment: for each page in our dataset, we identify the hosts which are allowed framing according to CSP, and we send an HTTP request to the page with the Referer header set to one of such hosts. In the presence of wildcards in CSP, e.g., `*.example.com`, we generate a synthetic candidate Referer matching them, e.g., `https://test.example.com`. If we observe that the value of the Referer is reflected back in the XFO header of the response, it means that we might have false positives in our set of inconsistencies, because the originally collected XFO headers only provided a partial picture of the deployed policy. We managed to perform this test on the 2,198 pages with CSP and observed extremely low adoption of Referer sniffing: in particular, only 11 pages relied on such practice. This gives us confidence in the correctness of the conclusions we draw.

In the next section, we provide an in-depth analysis of the inconsistent policies we collected. We do this while considering the full set of browsers in Table 2, because those browsers are actively used, and we want to assess the state of click-jacking protection on the Web as of now. We elaborate on the impact of the chosen browsers on our study in Section 5.4.

5.3 Analysis of Inconsistent Policies

To have a more in-depth look into the set of inconsistent policies, we performed a further classification step: in particular, we identified 590 security-oriented policies (33%) and 795 compatibility-oriented policies (44%), while the other 415 inconsistent policies (23%) do not belong to any of these two classes, hence are unduly inconsistent. In the rest of this section, we perform an in-depth analysis of the collected inconsistent policies and identify dangerous practices therein.

5.3.1 Security-Oriented Policies

The existence of security-oriented policies is justified by the fact that XFO is less expressive than CSP, hence Web developers might be led into shipping XFO headers that are more restrictive than the corresponding CSP headers. For example, the Web site `https://www.icloud.com` deploys an XFO header set to SAMEORIGIN and a CSP whitelisting every subdomain of `icloud.com` and `apple.com`. A similar situation happens on `https://academia.stackexchange.com`, which sets XFO to SAMEORIGIN and uses CSP to whitelist both itself and `https://stackexchange.com`. These policies offer a good level of protection to legacy browsers, but might introduce compatibility issues therein.

We further categorized the 590 security-oriented policies in two classes. The first class includes ineffective policies, where CSP is overly liberal compared to XFO: these policies allow framing from any host on CSP-enabled browsers, possibly just restricting its scheme, hence modern browsers are left unprotected. We noticed this problem just in 13 cases (2%), and we conjecture it might come from the wrong assumption that, when both XFO and CSP are enabled, they are both enforced, while CSP actually overrides XFO and voids protection. However, it is positive to see that this class of policies is highly under-represented. The other policies all take advantage of the additional expressive power of CSP over XFO for fine-grained whitelisting: specifically, we observed 99 cases (17%) where CSP was used to whitelist all the subdomains of the host whitelisted via XFO, while in all other cases CSP whitelisted at least two source expressions.

To the best of our knowledge, these look like legitimate use cases, where policy inconsistency is not necessarily dangerous for security. However, this discrepancy raises concerns, because it implies that either legacy browsers suffer from compatibility issues due to overly harsh security enforcement, or modern browsers are excessively liberal in their treatment of framing, i.e., the policies violate principle of least privilege.

5.3.2 Compatibility-Oriented Policies

Compatibility-oriented policies might be justified by the need to make Web applications accessible by legacy browsers, at the cost of (partially) sacrificing security in that case. For example, the Web site `https://www.spotify.com` deploys

Inconsistency Reason	Number of Policies	Fraction
Use of the ALLOW-FROM directive	323	78%
Comma-separated directives in XFO header	94	23%
Incomparable policies in XFO and CSP	53	13%
Use of multiple XFO headers	16	4%
Different policies sent to different browsers	5	1%

Table 6: Practices in unduly inconsistent policies (classes might overlap)

a CSP whitelisting every subdomain of `spotify.com` and `spotify.net`, but does not ship any XFO header, likely because XFO does not support such expressive whitelists. Another similar example is `https://www.sony.com`, which does not deploy XFO, but uses CSP to allow framing from itself and all the subdomains of three other trusted sites.

Recall that our dataset contains 795 compatibility-oriented policies. The first analysis we perform aims at understanding how much security legacy browsers sacrifice for such policies. For the very large majority of compatibility-oriented policies, we observed that XFO does not provide any protection at all, i.e., framing is allowed from any origin: this happened in 758 cases (95%). In particular, we found 705 pages where an XFO header is entirely absent (89%) and 99 pages where the XFO headers contain an incorrect directive or are misinterpreted by some legacy browser (11%). This shows that most Web developers are not actually concerned about offering security to users of legacy browsers, or are just entirely unaware of the existence of this problem.

To get a better understanding of the reasons underlying the existence of compatibility-oriented policies, we analyze the combination of XFO and CSP for the following scenario: if CSP is used to whitelist at most one origin, it is straightforward to write an XFO header which enforces exactly the same restrictions, hence the adoption of a compatibility-oriented policy is unjustified. We observe that this was the case for 105 policies (13%), where protection could be improved with minimal effort and expertise by the Web developers, i.e., without resorting to Referer sniffing. This shows that the bleak picture given above could be easily improved to some extent, yet this is not happening in practice.

5.3.3 Unduly Inconsistent Policies

Finally, we focus on the 415 inconsistent policies that are neither security-oriented nor compatibility-oriented. These policies are hard to justify as secure, or even as intended, as explained in Section 3. In particular, we observe the following distribution of (possibly overlapping) classes:

- 315 policies are interpreted differently by at least two legacy browsers (76%);
- 289 policies are interpreted differently by at least two modern browsers (69%);
- 29 policies are given the same interpretation by all legacy browsers and all modern browsers, yet these two interpretations are incomparable (7%).

What is worse is that 380 of these policies (92%) do not enforce any form of framing restriction on at least one of the browsers considered in our study, which confirms that this class of inconsistencies is particularly dangerous for security. For example, the Web site `https://es.sprint.com` sets an XFO header to ALLOW-FROM `https://www.sprint.com`, but does not ship a companion CSP: this leaves browsers without support for ALLOW-FROM unprotected. As another example, `https://whois.web.com` sends two XFO headers, one set to SAMEORIGIN and one set to DENY, which allows same-origin framing in some browsers but not others.

It is instructive to have a look at why these undue inconsistencies arise. Table 6 provides the breakdown of the main practices leading to policy inconsistency (classes partially overlap). We observe that the ALLOW-FROM directive is present in most of the unduly inconsistent policies, which shows that XFO is not properly coupled with CSP in those cases. Indeed, 322 out of 465 policies that use ALLOW-FROM do not come with any CSP (69%) and do not offer any protection on most modern browsers. It is also interesting that we found 53 policies where both XFO and CSP are syntactically correct, yet express incomparable policies. For example, we noticed that `https://gfp.sd.gov` deploys an XFO header set to SAMEORIGIN, while its CSP allows framing from every subdomain of `arcgis.com`, `soundcloud.com` and `flipsnack.com`. We do not have definite explanations for this kind of policies, but a plausible reason could be that XFO was deployed for a legacy version of the Web site and never updated later.

5.3.4 Perspective

We summarize here the security impact of our findings by computing the number of policies that do not offer any level of protection to at least one browser. We also present the same perspective for modern browsers alone. The presence and distribution of vulnerable policies for these two cases are shown in Table 7. These numbers confirm our claim that not all inconsistencies are necessarily dangerous, yet their majority actually is (64%). In particular, almost every inconsistent policy that is not security-oriented is completely ineffective on

Inconsistency Class	Vulnerabilities (Any Browser)	Vulnerabilities (Modern Browser)
Security-Oriented	13 (2%)	13 (2%)
Compatibility-Oriented	758 (95%)	3 (<1%)
Unduly Inconsistent	380 (92%)	278 (67%)
Aggregate	1,151 (64%)	294 (16%)

Table 7: Presence and distribution of vulnerable policies

at least one browser. Luckily, our experiments also show that users of modern browsers enjoy a significantly higher level of protection than users of legacy browsers since only 16% of the inconsistencies actually void any form of security enforcement in a modern browser, where undue inconsistencies are essentially the only threat.

5.4 The Role of Browsers

Since we assess inconsistencies over a set of popular browsers, one might wonder to which extent the chosen browsers bias the results of our study. To understand this point, we decided to run a second analysis by removing Internet Explorer and Opera Mini from the set of browsers under test. The rationale of this choice is that these browsers do not support CSP, and thus, we might get a picture of how much the current policy deployment would be inconsistent in a world without legacy browsers. It turns out that the total number of inconsistent policies would drop from 1,800 to 289, which is a major improvement. However, observe that all such policies fall in the class of unduly inconsistent policies (since we removed legacy browsers), and we computed that for 278 of them (96%) there is at least one modern browser which does not enforce any form of restriction. This confirms that the adoption of modern browsers strongly mitigates the problem of inconsistencies, yet not entirely solved. The main reasons for inconsistency would still be the use of ALLOW-FROM and the adoption of a comma-separated list of directives in XFO.

It is also particularly interesting that two of the browsers that we tested have been undergoing major changes at the time of writing. The first significant change was implemented in Firefox, which dropped support for the ALLOW-FROM directive in version 70.[7] Moreover, Microsoft announced that Edge will move to the Chromium architecture in 2020, which likely means that it will drop support for ALLOW-FROM and fix the problems with XFO headers. These changes go in the direction of reducing the risk of inconsistencies in modern browsers, which will eventually be uniformed to Chromium derivates. Unfortunately, we also showed that 322 out of 465 policies that use ALLOW-FROM do not come with any CSP (69%), which implies that these changes are weakening the state of click-jacking protection on the Web.

At the end of the day, we believe that the problem of inconsistencies in click-jacking protection is far from solved. Though legacy browsers not supporting CSP are likely going to disappear in a few years, it is hard to predict a precise temporal horizon for this: for example, Internet Explorer 11 was launched in 2013, and it still has $\sim$ 2% of the market share based on publicly available data, while Opera Mini is still under active development and extremely popular with around 15% market share in Africa, where mobile traffic is still expensive.[8] Also, it should be noted that the versions of Edge and Firefox considered in the present study might still be around for a while, i.e., the Web platform will still be accessed by browsers supporting ALLOW-FROM at least in the near future. Though a full transition from XFO to CSP for click-jacking protection is the way to go to solve the issue of inconsistencies, the setting is complex and requires actions at different levels. We discuss recommendations and countermeasures in the next section.

5.5 Limitations

Though we strived to quantify the security impact of the detected policy inconsistencies, we cannot show that even policies that do not provide any form of framing control in some browsers lead to exploitable vulnerabilities in practice. To overcome this limitation, we would need to identify pages that are susceptible to framing-based attacks. However, identifying these in an automated fashion at a large scale requires accounts of all tested sites as well as an in-depth understanding of the application's semantics. However, we argue that it is fair to assume that site operators are deploying framing control for a reason. In our opinion, the widespread adoption of framing control policies (33% of all crawled URLs, spread across 58% of the sites we looked at) motivates that click-jacking is perceived as an important security threat. Our analysis acts as a cautionary tale aimed at raising awareness of the potential issues that arise from policy inconsistencies.

In addition to this, we also remark that our study specifically focuses on the 10,000 most popular sites at the time of writing the paper. Given the diversity of the Web in general, this does not necessarily enable us to generalize about framing control inconsistencies on the entire Web. As prior

[7] https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/70#HTTP

[8] https://blogs.opera.com/mobile/2019/08/opera-is-leading-the-digital-revolution-in-africa/

work has shown [24], though, the popularity of domains often represents a proxy for security measures, meaning that our results most likely are a lower bound of the actual problems discoverable in the wild.

6 Recommendations and Countermeasures

Based on the data gathered in our analysis of both browser implementations and real-world deployment of framing control, we discuss lessons learned to improve the situation. In particular, we first present recommendations for both Web developers and browser vendors, highlighting some room for improvement which we found. We then discuss our implementation of a server-side proxy capable of retrofitting framing control policies in existing Web applications for the diverse set of browsers we considered in our analysis.

6.1 Recommendations for Web Developers

The first important recommendation we make is that both XFO and CSP must be used for effective framing control on the current Web. XFO alone is insufficient for security because sites might be prone to double framing attacks (also in modern browsers like UC Browser) or even not protected at all (most notably, in the presence of the largely unsupported ALLOW-FROM directive). On the other hand, just using CSP results in leaving users of legacy browsers completely unprotected. Unfortunately, we found that only 8% of the collected policies use both XFO and CSP. Worse, the combination of the two mechanisms proved hard to get right for Web developers, as 54% of such policies are inconsistent.

The other crucial recommendations are about the use of XFO. Web developers should ensure that at most one XFO header is sent with every Web page because existing browsers have inconsistent interpretations in the presence of multiple XFO headers. What is worth noting here is that there is no good practical reason to deploy more than one XFO header. In the presence of multiple XFO headers, existing browsers either enforce the first one (thus voiding the others) or simultaneously enforce all of them. However, even this is useless, because any pair of XFO directives always contains either redundant or contradictory information, which can be expressed with a single XFO directive (see Table 8). For the same reasons we just discussed, Web developers should avoid the use of comma-separated values in XFO headers. These headers are parsed as multiple XFO headers in most browsers, while in other browsers, they are interpreted as non-existing directives that do not enforce any form of framing control. This latter observation shows that even the apparently innocuous practice of repeating the same directive multiple times is actually insecure because it voids protection on some browsers.

6.2 Recommendations for Browser Vendors

Though the `frame-ancestors` directive obsoleted XFO back in 2014, XFO is still very popular in the wild: 88% of the policies we collected are still based on XFO alone. This means that this is not the right time to drop support for XFO, and one might wonder if this will ever be possible without leaving a significant fraction of the Web unprotected. An important point we would like to stress is the need for more informational messages for Web developers, e.g., in the JavaScript console. A prime example of this issue comes from the recent removal of support for ALLOW-FROM in Firefox. When visiting a page that sends an XFO header containing such a directive, Firefox merely notes an invalid header and points the developer to the generic Mozilla Developer Network page on XFO. This page does note that ALLOW-FROM is now obsolete and should not be used, but does not provide an immediately visible and explicit warning that sites using ALLOW-FROM have suddenly become unprotected. As to Chrome, the JavaScript console only shows a warning about an unrecognized directive and nothing more.

We argue that browsers should explicitly warn Web developers about the possibility of using CSP to achieve the same effect of XFO, which is straightforward considered that CSP is more expressive than XFO. In particular, XFO policies which do not contain glaring mistakes can be readily transformed into corresponding CSPs. We designed one such solution as part of our server-side proxy (see Section 6.3), which might be inspiring also for browser vendors since the same approach could be applied at the client. We understand that major browser vendors might consider such transformations dangerous for backward compatibility, yet even simple transformations might significantly increase security in the wild and are worth testing in our opinion. At the very least, a candidate value for `frame-ancestors` combined with a clear warning about the unprotected state of the site should be reported in the JavaScript console.

On more general terms, we think that our paper shows the importance of implementing only client-side security mechanisms that come with a clear and precise specification. The XFO specification was put together only after major browsers already implemented support for the XFO header, which led to many different implementations. Though the auto-update feature of modern browsers certainly helps in mitigating the problem of inconsistencies, real-world market share data show that legacy browsers are hard to eradicate. Once a client-side security mechanism has been inconsistently implemented across browsers, it might be challenging to understand its long-lasting impact in the wild. For example, without moving away from CSP, the `strict-dynamic` source expression has first been implemented in Chrome due to an independent effort from Google's engineers and then pushed into the CSP standard. This kind of practice is dangerous because other browser vendors might be unwilling to pick up: for example,

Directive 1	Directive 2	Conjunction of Directives
SAMEORIGIN	SAMEORIGIN	SAMEORIGIN
SAMEORIGIN	ALLOW-FROM o'	DENY if $o \neq o'$, SAMEORIGIN otherwise
SAMEORIGIN	DENY	DENY
ALLOW-FROM o'	ALLOW-FROM o''	DENY if $o' \neq o''$, ALLOW-FROM o' otherwise
ALLOW-FROM o'	DENY	DENY
DENY	DENY	DENY

Table 8: Simplification of multiple XFO directives into a single one (adoption at origin o)

Safari still lacks support for `strict-dynamic`. This decision, however, may well be a good one, given that recent work has shown the dangers of `strict-dynamic` through script gadgets, and even Google engineers now advocate to instead rely on explicit passing of nonces [13]. Nevertheless, this feature is inconsistently implemented across browsers already and unlikely to be removed in the near future.

6.3 Retrofitting Security

As Web developers might not be aware of the intricacies of the two mechanisms available to control the framing of their sites, we developed a server-side proxy designed to enforce consistency in framing control policies, i.e., to ensure all browsers enforce the same level of protection. The proxy is a Python script ($\sim$ 800 LoC), which can be run at the server. It inspects the HTTP traffic to automatically fix the framing control headers so as to ensure policy consistency. To enable researchers to build on our work and website administrators to benefit from the tool, we have made the proxy available at https://github.com/cispa/framing-control-proxy.

In particular, for any request r, let $\overline{r}$ stand for the corresponding HTTP response. If $\overline{r}$ contains XFO headers, but no CSP header with a `frame-ancestors` directive, the proxy behaves as follows:

1. if multiple XFO headers are present in $\overline{r}$, they are first folded into one XFO header set to a comma-separated list of the specified directives;

2. after step 1, $\overline{r}$ is guaranteed to contain exactly one XFO header. If the header contains a comma-separated list of directives, it is replaced by a single directive enforcing the same security restrictions of the conjunction of the directives. This is always possible, thanks to the simplification rules in Table 8;

3. the proxy finally attaches to $\overline{r}$ a new CSP header enforcing the same framing control restrictions of the sanitized XFO header. This is straightforward, since CSP is more expressive than XFO, and does not conflict with other CSP headers possibly present in $\overline{r}$, since, when multiple CSP headers are sent, their conjunction is enforced and no other `frame-ancestors` directive is present.

If $\overline{r}$ contains CSP headers with a `frame-ancestors` directive, the proxy instead behaves as follows:

1. all the XFO headers of $\overline{r}$ are stripped away;

2. the proxy computes the union of the source expressions whitelisted in all the `frame-ancestors` directives contained in the CSP headers of $\overline{r}$;

3. if CSP denies framing, $\overline{r}$ is extended with an XFO header containing the DENY directive. If instead CSP only allows same-origin framing, $\overline{r}$ is extended with an XFO header containing the SAMEORIGIN directive. Otherwise, the proxy checks if the Referer header of r contains a URL whitelisted by any of the source expressions identified at step 2: if this is the case, $\overline{r}$ is extended with an XFO header containing an ALLOW-FROM directive set to the origin of the Referer header; otherwise, the XFO header is set to DENY. If r lacks the Referer header, the proxy conservatively sets the XFO header to DENY.

Eventually, the proxy ensures the consistency of framing control policies with respect to the set of tested browsers, by equating the security guarantees of XFO and CSP (up to double framing). Observe that, although Opera Mini supports neither CSP nor ALLOW-FROM, the proxy still manages to rectify its limitations. In particular, if the Referer of the request is set to a whitelisted URL, the proxy sets XFO to the corresponding ALLOW-FROM directive, which is just ignored by Opera Mini and framing is allowed. Otherwise, the proxy sets XFO to DENY, and the page cannot be framed.

In our design, we prioritize CSP headers over XFO headers when both are present since CSP is the preferred method to enforce framing control in modern browsers. This means that it is occasionally possible for the proxy to relax security restrictions beyond least privilege: for example, if a page sets XFO to DENY and CSP allows same-origin framing, then XFO will be relaxed to SAMEORIGIN. However, this is sensible from a security perspective, because modern browsers already allow same-origin framing, so we assume this was

intended by the site administrators, as modern browsers are the primary target in the market and are also easier to test. This is also backed up by our dataset, where we observed only 13 policies where XFO was tighter than CSP and CSP was configured in an obviously insecure manner (see Table 7).

As a final point, we note that the Referer header may be stripped when controlled through the Referrer-Policy [16], which would disable the possibility of performing Referer sniffing in the proxy. However, Referrer-Policy is only supported in browsers that also support the `frame-ancestors` directive of CSP. Since the proxy only relies on Referer sniffing in the presence of `frame-ancestors`, the DENY directive placed in the absence of the Referer header would be overridden by CSP in all cases. After implementing our proxy, we tested it out against the full set of test cases of Section 4.2. By doing so, we confirmed that the proxy behaves as expected and enforces the same security restrictions in the entire pool of browsers.

7 Related Work

In this section, we present related work, and for the work closest to ours, we explain the main differences.

CSP and XFO for Framing Control In their 2019 paper, Luo et al. [14] studied the evolution of mobile browsers and their support for client-side security mechanisms over time. In doing so, they also documented the interplay between CSP and XFO, reporting in particular that some mobile browsers did not prioritize CSP over XFO in the past. Their paper generically hints that inconsistencies between CSP and XFO could occur based on the collected headers, yet the paper does not go much in detail about this. The increased importance of CSP for framing control was also documented by Roth et al. [19], who analyzed the evolution of CSP from 2012 to 2018, indicating that CSP has become more and more popular as a protection mechanism against click-jacking. They also evaluated the dangers coming from the inconsistent support for ALLOW-FROM and CSP in different browsers, most notably by leveraging the well-known observation that the ALLOW-FROM directive is not supported in Chrome.

Though both these studies have been inspiring starting points for our work, we extend the mere analysis of the potential problems by building a comprehensive framework to reason about inconsistencies. In particular: (i) we formally define the problem of inconsistencies in framing control policies to provide a full account of this security problem, highlighting different classes of inconsistencies with different security implications; (ii) we focus on both desktop browsers and mobile browsers, exposing many new and unreported dangerous implementations of the underspecified XFO header; (iii) we perform an in-depth analysis of several root causes of inconsistencies in the wild, their security import, and some possible countermeasures, discussing the potential role of browser vendors on the way forward; and (iv) we implement and release a server-side proxy designed to retrofit security in existing Web applications by enforcing consistency for the set of browsers that we analyzed.

Click-Jacking Protection and Attacks In 2010, Rydstedt et al. [20] studied the usage of frame busting scripts in the Alexa Top 500 sites, showing that the deployed mechanisms through JavaScript were trivial to bypass. In the same year, Balduzzi et al. [2] built a system capable of detecting click-jacking, primarily based on the assumption that elements should not be overlapping when clicked. In 2012, Lekies et al. [12] highlighted additional techniques for bypassing existing defenses and showed the shortcomings of XFO for fine-grained framing control. In the same year, Huang et al. [7] conducted an in-depth analysis of the underlying issues and proposed InContext, in which applications could mark specific elements as sensitive (e.g., Like buttons), which would, through various defensive techniques, be protected from forced clicks at the browser. In 2014, Akhawe et al. [1] generalized click-jacking to perceptual UI attacks and showed how easily users could be tricked into clicking unwanted elements while seemingly playing a benign game.

Inconsistencies in Web Security Inconsistencies in the implementation of client-side security mechanisms have been first studied by Singh et al. [22]. Their seminal work focused on access control policies and, in particular, on parts of the Same Origin Policy (SOP), which proved to be inconsistently implemented in existing Web browsers at the time. A similar study was later performed on modern browsers by Schwenk et al., and also exposed dangerous inconsistencies [21]. Automated testing has been proposed as an effective technique to catch bugs in the implementation of client-side security mechanisms by Hothersall-Thomas et al. [6]. None of these studies focused on inconsistencies in framing control policies.

Naturally, the client is not the only software where inconsistencies may occur. In particular, prior work has investigated the handling of multiple Host headers in CDNs and origin servers, showing that due to differences in handling multiple headers, these two components end up with a different understanding of the requested host [5]. In a recent paper, Nguyen et al. [17] showed that inconsistencies in allowed header lengths or control characters could allow an attacker to force origin servers to yield error pages. This, in combination with CDNs that cache such error pages, can lead to a cache-poisoned Denial of Service attack. In non-academic research, Kettle [10] showed that using multiple Content-Length headers as well as conflicting Transfer-Encoding allows for HTTP Desync attacks. Albeit only indirectly related to our paper, these works clearly document the dangers of inconsistent implementations on the Web.

Finally, Mendoza et al. [15] studied the inconsistent adoption of security mechanisms in the mobile and the desktop version of the same Web site. They even showed attacks where the insecurity of a mobile site could be exploited to target the desktop site, which sits at a higher security level.

8 Conclusion

In this paper, we presented the first comprehensive analysis of inconsistencies in framing control policies. We based our investigation on a formal framework, which constituted the basis for the implementation of a real-world policy analyzer dubbed FRAMECHECK. Our analysis of 10,000 Web sites from the Tranco list showed that the problem of inconsistencies is widespread on the Web, since around 10% of the (distinct) framing control policies in the wild are inconsistent and most often do not provide any form of protection to at least one browser. Given the insights of the dangers caused through inconsistencies, we proposed different countermeasures in terms of recommendations for Web developers and browser vendors, as well as the implementation of a server-side proxy designed to retrofit security to existing Web applications. We are currently in the process of responsibly disclosing the security issues found throughout our comprehensive analysis to the affected browser vendors and site operators.

We foresee a few avenues for future work. First, we would like to extend our current analysis to uncover inconsistencies between the desktop version and the mobile version of the same Web site, following the approach proposed by Mendoza et al. [15]. Then, we plan to generalize our formal framework to other client-side security mechanisms besides XFO and the framing control fragment of CSP. Finally, we would like to carry out a systematic analysis of the compatibility impact of some of our proposed countermeasures, which we only evaluated in terms of security so far. This might require close collaboration with browser vendors to understand their impact on a large scale.

Acknowledgements

We would like to thank the reviewers for their advices on how to improve the presentation of our paper. In particular, we thank Adam Doupé for his guidance in the shepherding process. Furthermore, we want to thank Alexander Fink for the helpful discussions regarding implementation details of the proxy's network traffic interception.

References

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking revisited: A perceptual view of UI security. In *USENIX WOOT*, 2014.

[2] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *AsiaCCS*, 2010.

[3] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Wilayat Khan. Cookiext: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4), 2015.

[4] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Semantics-based analysis of content security policy deployment. *TWEB*, 12(2), 2018.

[5] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of troubles: Multiple host ambiguities in http implementations. In *CCS*. ACM, 2016.

[6] Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic. Browseraudit: automated testing of browser security features. In *ISSTA*, 2015.

[7] Lin-Shung Huang, Alexander Moshchuk, Helen J. Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security*, 2012.

[8] Internet Engineering Task Force. Hypertext transfer protocol (http/1.1): Message syntax and routing, . URL https://tools.ietf.org/html/rfc7230.

[9] Internet Engineering Task Force. Http header field x-frame-options, . URL https://tools.ietf.org/html/rfc7034.

[10] James Kettle. HTTP Desync Attacks: Request Smuggling Reborn. Online https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn.

[11] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS*, 2015.

[12] Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz, and Martin Johns. On the fragility and limitations of current browser-provided clickjacking protection schemes. In *USENIX WOOT*, 2012.

[13] Lukas Weichselbaum and Michele Spagnuolo. CSP - A Successful Mess Between Hardening and Mitigation. Online https://static.sched.com/hosted_files/locomocosec2019/db/CSP%20-%20A%20Successful%20Mess%20Between%20Hardening%20and%20Mitigation%20%281%29.pdf.

[14] Meng Luo, Pierre Laperdrix, Nima Honarmand, and Nick Nikiforakis. Time does not heal all wounds: A

longitudinal analysis of security-mechanism support in mobile browsers. In *NDSS*, 2019.

[15] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. Uncovering HTTP header inconsistencies and the impact on desktop/mobile websites. In *WWW*, 2018.

[16] Mozilla Developer Network. Referrer-Policy. Online https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy.

[17] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your cache has fallen: Cache-poisoned denial-of-service attack. In *CCS*, 2019.

[18] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *NDSS*, 2019.

[19] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex Security Policy? – A Longitudinal Analysis of Deployed Content Security Policies. In *NDSS*, 2020.

[20] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities on popular sites. In *W2SP*, 2010.

[21] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-origin policy: Evaluation in modern browsers. In *USENIX Security*, 2017.

[22] Kapil Singh, Alexander Moshchuk, Helen J. Wang, and Wenke Lee. On the incoherencies in web browser access control policies. In *IEEE S&P*, 2010.

[23] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in)security. In *USENIX Security*, 2017.

[24] Tom Van Goethem, Ping Chen, Nick Nikiforakis, Lieven Desmet, and Wouter Joosen. Large-scale security analysis of the web: Challenges and findings. In *TRUST*, 2014.

[25] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *CCS*, 2016.