Saarland University
Department of Computer Science

Three modest proposals for building trust in social media discourse, in the software that powers it and in the browsers that run the software

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Ilkan Esiyok

Saarbrücken, 2023

Tag des Kolloquiums:	3 September 2024
Dekan:	Prof. Dr. Roland Speicher

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Markus Bläser
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Dr. Stefano Calzavara
Akademischer Mitarbeiter:	Dr. Robert Künnemann

Zusammenfassung

Da das Web immer komplexer wird, ist die Schaffung von Vertrauen (definiert als die Gewährleistung eines sicheren und zuverlässigen Systembetriebs) wichtiger denn je. Diese Arbeit befasst sich mit der notwendigen Vertrauensbildung in drei wesentlichen Komponenten des digitalen Ökosystems: dem Diskurs in sozialen Medien, der zugrunde liegenden Software und den Webbrowsern, die für die Ausführung dieser Software verantwortlich sind. In dieser Arbeit stellen wir drei innovative Vorschläge vor, um das Vertrauen in jedem dieser Bereiche zu verbessern.

Unsere erste Initiative namens „Trollthrottle“ zielt auf den Diskurs in den sozialen Medien ab. Dieses Protokoll begrenzt die Anzahl der Kommentare, die ein Nutzer auf teilnehmenden Websites veröffentlichen kann, und macht diese Websites gleichzeitig für etwaige Inhaltszensur verantwortlich. Da das Protokoll als externe Browserfunktion implementiert ist, hängt die Wirksamkeit des Protokolls von der genauen Übertragung und Ausführung seines Codes durch den Browser ab. Da ein bösartiger Webserver unentdeckt unterschiedlichen Code an verschiedene Nutzer senden kann, wird Vertrauen zu einem Problem.

Um dieses Problem anzugehen, präsentieren wir unseren zweiten Vorschlag mit dem Namen „Accountable JS“, der das Vertrauen in den zugrunde liegenden Webanwendungscode sicherstellen soll. Mit diesem Protokoll können Nutzer bestätigen, dass der aktive Inhalt im Code einer Webanwendung, der von einer Website geliefert wird, für alle Besucher konsistent ist und den Richtlinien des Protokolls entspricht. Diese Maßnahme setzt jedoch auch Vertrauen in den Webbrowser selbst voraus, um eine zuverlässige Ausführung zu gewährleisten.

Unser dritter Vorschlag mit dem Titel „Formales Browsermodell für die Sicherheitsanalyse“ zielt direkt auf dieses Problem ab. Dieses umfassende Framework nutzt ein formales Modell von Webbrowsern, das auf RFC-Standards basiert, um potenzielle Sicherheits- und Datenschutzschwachstellen zu identifizieren. Sollte bei der Modellprüfung ein Gegenbeispiel gefunden werden, generiert das Framework ein entsprechendes Testszenario zur Ausführung im Browser. Dies ermöglicht es Browserentwicklern und -testern, ihre Produkte eng an den RFC-Richtlinien auszurichten, Schwachstellen zu identifizieren und schließlich zu beheben. Eine solche gründliche Überprüfung erhöht das Vertrauen der Nutzer, da sie sicherstellt, dass der Browser strengen Tests und Validierungen anhand idealer Sicherheitsmaßstäbe unterzogen wurde.

In dieser Arbeit gehen wir näher auf diese Protokolle ein, befassen uns intensiv mit der Software, die ihnen zugrunde liegt, und wenden sie auf verschiedene Fallstudien an, die jeweils in separaten Kapiteln besprochen werden.

Abstract

As the web has become increasingly complex, establishing trust (defined as the assurance of secure and reliable system operations) is more vital than ever. This thesis addresses the trust imperative across three critical components of the digital ecosystem: social media discourse, underlying software and the web browsers that are responsible for executing this software. In this thesis, we offer three modest proposals to enhance the trust in each of these domains.

Our first initiative called "Trollthrottle" targets social media discourse. This protocol limits the number of comments a user can post on participating websites while also holding these sites accountable for any content censorship. As the protocol implemented as an external browser feature, the protocol's effectiveness relies on the accurate transmission and execution of its code by the browser. Given that a malicious web server can deliver different codes to different users without detection, trust becomes a concern.

To address this, we present our second proposal called "Accountable JS" designed to ensure trust in the underlying web application code. This protocol empowers users to confirm that the active content in the web application code delivered by a website is consistent for all its visitors and compliant with the protocol's guidelines. However, this measure also necessitates trust in the web browser itself for faithful execution.

Our third proposal called "Formal Browser Model for Security Analysis" aims directly at this issue. This comprehensive framework utilises a formal model of web browsers based on RFC standards to identify potential security and privacy vulnerabilities. When a counterexample is found during model checking, the framework generates a corresponding test case scenario for execution in the browser in order to check whether the same behaviour can be observed in the real-world browser. This enables browser developers and testers to align their products closely with RFC guidelines, thereby identifying and mitigating vulnerabilities. Such thorough verification increases user confidence, as it ensures that the browser has undergone rigorous testing and validation against ideal security benchmarks.

Throughout this thesis, we elaborate on these protocols, delve into the software that underpins them and apply them to various case studies, each discussed in separate chapters.

Background of this Dissertation

In the following, we discuss the papers and the open-source artefacts that form the foundational basis of this thesis. While the author of this thesis led all of these research projects, we will emphasise the contributions made by other researchers. With the exception of one paper, which we plan to submit to The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) conference in 2024 by submission deadline (Round 1) in December 2023, all the remaining papers have been accepted and presented at prestigious peer-reviewed conferences in the field of information security.

Chapter 3, in which we describe an internet protocol [P1] that addresses the problem of astroturfing on social media websites and provides accountability for censorship, was presented in Applied Cryptography and Network Security (ACNS) 2020 conference. Cryptographic proofs and an extension to the protocol in this paper were conducted by Lucjan Hanzlik. For completeness they are included in Chapter 7 in Trollthrottle Appendix.

Chapter 4 is based on our paper presented in the Network and Distributed System Security Symposium (NDSS) 2023, in which we present an opt-in accountability protocol [P2] for websites that want to convince their users that they are trustworthy in an economical way. Using this protocol the users can verify that the active content on the web page, they are visiting is the same for all visitors of the website. Moreover, in the client side the code adheres to the standards set by the protocol. This project received a research award for privacy-preserving technologies from Meta research, specifically for the ”Transparency.js, transparency for active content” initiative. In the beginning of this project, we were in close contact with Meta's WhatsApp team to gather industrial expectations for accountability and security, and we shared the first draft of this paper with the WhatsApp team. Then, Meta took the lead to implement Code Verify tool [76] that also implements accountability for active content for only Meta websites for now and it has been made available for end users in 2022.

In Chapter 5, we present our work [P3] that is conducted with Hamed Nemati and Robert Künnemann. We intend to submit this work to the ISSTA 2024 conference (deadline in December 2023). In this work, we contribute to web browser security by systematically identifying potential vulnerabilities in web browsers. We utilise formal modelling of web browsers and generate test cases for the counterexamples from model checking, enabling further analysis on the potential vulnerabilities.

[P1] Esiyok I., Hanzlik L., Künnemann R., Budde L. M., and Backes M., “TrollThrottle —Raising the Cost of Astroturfing”. In Applied Cryptography and Network Security, 2020.

[P2] Esiyok I., Berrang P., Gordon K-C., Künnemann R., "Accountable Javascript Code Delivery". In Network and Distributed System Security Symposium, 2023.

[P3] Esiyok I., Nemati H., Künnemann R., "Formal Browser Model for Security Analysis". Under submission, 2023.

The Open-source Artefacts

We developed several open-source implementations and formal models for this thesis.

- For [P1] in Chapter 3, we implemented a prototype of the protocol to evaluate in terms of how easy it is to deploy [164] and how much performance overhead it incurs [165]. We also forked an implementation of a cryptographic operations library MIRACL [118] for adapting it to support the requirements of the prototype in the web browser. In addition, we formally modelled and verified correctness of the protocol using Tamarin Prover in Appendix Section 7.4. Furthermore, we generated big query scripts to conduct empirical analysis on big social media data [163].
- For [P2] in Chapter 4, we implemented a web browser extension prototype to evaluate performance and applicability of the approach on the client side [58]. We also formally verified the correctness of the protocol using Tamarin Prover.
- For [P3] in Chapter 5, we generated a formal model in Alloy language that captures web browser behaviours, and we provided an implementation that parses the model checking tool's output and extracts test cases out of them. Then, we implemented a simulator tool that simulates these attack scenarios on the browser to check if the attack can be realised on the browser.

Further Contributions of the Author

The author of this thesis has contributed to two more papers ([P4], [P5]) which are not part of this thesis.

[P4] received a Distinguished Paper Award from Computer Security Foundations Symposium (CSF) 2019 conference. In this work, we defined a mechanised framework for the verification of accountability in security protocols in which the framework at the time was built on top of SAPIC [92] tool. The author took the responsibility for formally modelling and verifying the case studies of this project using SAPIC and Tamarin Prover. The author also took part in improving the SAPIC implementation in Tamarin Prover repository.

In [P5], we applied formal analysis and compared the security of the two widely used exposure notification systems : the ROBust and privacy-presERving proximity Tracing protocol (ROBERT) and the Google Apple Exposure Notification (GAEN) framework. In this project, the author was responsible to reverse engineer the ROBERT and GAEN implementations that contributed to the comparative analysis.

[P4] Künnemann R., Esiyok I., and Backes M., "Automated Verification of Accountability in Security Protocols". In IEEE 32nd Computer Security Foundations Symposium (CSF), 2019.

[P5] Morio K., Esiyok I., Jackson D., Künnemann R., "Automated Security Analysis of Exposure Notification Systems". In 32st USENIX Security Symposium, 2023.

Acknowledgments

I am profoundly grateful to my advisor, Prof. Michael Backes, for providing me the incredible opportunity to pursue my Ph.D. in his research group. I consider myself very fortunate to have been a part of such a dynamic and enthusiastic team. Special thanks go to Dr. Robert Künnemann, who has been an invaluable mentor for me from the beginning of my research journey to its conclusion. I have always admired his calm, patient and rational approach to problem-solving. It was a great pleasure to work with Dr. Künnemann who always has lots of diverse and insightful ideas. I also wish to acknowledge my research collaborators: Lucjan Hanzlik, Lena Marie Budde, Pascal Berrang, Katriel-Cohn Gordon, Kevin Morio, Dennis Jackson and Hamed Nemati, for their invaluable contributions.

Additional thanks go to Patrick Speicher and Hamed Rasifard, who have been excellent roommates. I am also grateful to my colleagues in the research group : Faezeh Nasrabadi, Matthis Kruse, Marco Patrignani, Xaver Fabian, Guido Patrick Battiston and Tiziano Marinaro, for their awesome support. Further, I would like to thank all the employees at CISPA for contributing to a collaborative and enjoyable work environment. Furthermore, I wish to express my deepest appreciation to my family for their unconditional support throughout my journey, without which this work would not have been possible. Lastly, a special mention to my dear friend Paulina Jaszewska, whose friendship and encouragement during challenging times have been invaluable.

I am further indebted to the generous support provided by the Meta Research Award, the ERC Synergy Grant and the Turkish Ministry of Education Scholarship, providing the financial support for my pursuit.

Contents

List of Figures

List of Tables

List of Code Listings

1

Glossary

1.1 Glossary

CA	Certificate Authority
CDN	Content Delivery Network
CSP	Content Security Policy
CT	Certificate Transparency
DOM	Document Object Model
JS	JavaScript
OCSP	Online Certificate Status Protocol
PKI	Public Key Infrastucture
SPA	Single Page Applications
SRI	Subresource Integrity
SXG	Signed HTTP Exchanges
TLS	Transport Layer Security
XSS	Cross-Site Scripting
RFC	Request for Comments
CEGAR	Counterexample Guided Abstraction Refinement
SAB	SharedArrayBuffer

2
Introduction

This thesis addresses three key areas in the web where building trust is important: social media discourse, the underlying software and the browsers that are responsible for executing the software. We present three proposals to enhance the trust in each of these domains.

This chapter is structured into three subsections, each representing one of the three proposals. The first subsection introduces a trustworthy social media platform. The second subsection provides a protocol designed to address trust issues in the underlying software of the social media platform and other web applications. Lastly, the third subsection presents a formal model and a test case validation platform aimed at enhancing the trust in the web browser used for executing the software.

Trustworthy Social Media Platform

The internet is a decentralised platform and it is a global collection of distributed computer networks spanning across countries, organisations and individuals. There is no single entity that has the authority to control the internet. However, in the context of the web, that is the primary platform to access the online content, it has increasingly become centralised and dominated by a few large technology operators such as Facebook, YouTube, Twitter and Reddit, that have lots of power and influence on controlling the web.

This centralisation of the web raises concerns about trust, freedom of expression, censorship, political activism and manipulative content selection particularly on social media platforms. Platforms may selectively censor certain opinions that they think unfavourable or inconsistent with their policies. Currently, there is a lack of effective mechanisms to detect and hold platforms accountable for censorship, manipulative content selection or other concerns. A recent analysis [75] on the Twitter algorithm, that has recently been posted online by Twitter [150], showed that the platform performs more aggressive filtering on the Ukraine-Russia crisis content which means fewer tweets and more censorship in this context. Another example is manipulative content selection on social media feeds and on trending topics. A feed algorithm may prioritise posts and trending topics that align with the user's interests saved on the platform to keep users engaged and addicted to the platform. This can lead to less exposure to diverse viewpoints and increase polarisation. In fact, a recent study presents polarisation of opinions on the vaccination debate on Facebook [141] and discusses that social media campaigns which aim to provide accurate information struggle to reach beyond specific sub-groups because of the echo chambers created by social media platforms.

Few platforms focus on building trust with their users and aim to provide more reliable and transparent social media experience. Those platforms are open about their algorithms for the operations such as content filtering, content selection and moderation. In contrast to the large operators, users of these platforms have greater confidence in the contents they encounter. For instance, they can be assured that 'all' content related to a specific topic is presented to the users without censorship or the content they observe is presented without manipulation by paid agents trying to advertise some opinion or product for profit or political gain.

In the first part of this thesis in Chapter 3, we propose such a trustworthy platform that mainly tackles the problem of astroturfing and censorship in social media. Astroturfing can be defined as the fabrication of public discourse by private or state-controlled sponsors via the creation of fake online accounts. Classic astroturfing involves paid agents fabricating false public opinion about a particular product or topic. These agents may act as ordinary users for creating an illusion to show widespread support on a product or opinion to manipulate public perception. With the rise of social media and various functionalities in the web to remain anonymous, astroturfing has become more prevalent and cost-effective. In fact, advancements in the technology has also allowed to mechanise the astroturfing with automated internet bots.

Following the 'one person one voice' principle, we introduce Trollthrottle platform that limits the number of comments a person can post on participating websites to a certain threshold. The goal is raising the cost of astroturfing: if the threshold is τ, the cost of posting n comments is the cost of acquiring $\lceil \frac{n}{\tau} \rceil$ identities, be it by employing personnel, by bribery or by identity theft. An important ingredient of this platform is accountability for censorship: if a user believes her comment ought to appear on the website, she can provide evidence that can be evaluated by the public to confirm misbehaviour on the part of the website. By employing a public ledger that stores users' comments, the platform provides assurance that the comments that are addressed to a website should actually appear on the website. If a website does not publish a user's comment, it must have sufficient grounds for censorship that must be evaluated by the public.

The astroturfing problem has become more prevalent with the emergence of large language models such as ChatGPT from Open AI. Previously the private or state-controlled sponsors had to employ personnel and allocate funds to generate comments on the websites. However with the large language models, the process has become easier and cost-effective for these sponsors. Rather than relying on human personnel, they can simply make requests to the large language models, such as asking for a 'two thousand ways to downplay the significance of global warming'. This change has enabled easier and cheaper astroturfing. This makes Trollthrottle more relevant to address this problem as it limits the number of comments that can be posted on websites, regardless of whether they are carried out by human personnel or internet bots.

The Trollthrottle platform is web-based and operates by delivering necessary libraries and content directly to the user on the client side. To ensure the effective functioning and to fulfil the desired properties of the Trollthrottle system which we define extensively in the Chapter 3, it is essential for the website visited by the user to deliver the correct libraries that will be executed on the client side of the application. Any manipulation or tampering with these libraries by the website can undermine the integrity of the system. Therefore, Trollthrottle system requires the trust in the executable (or active) content delivered from the websites.

For addressing this trust requirement on the active content, we developed our second work Accountable JavaScript which is defined in the Chapter 4.

Active content such as JavaScript that is delivered to the user for each web request can be tampered with by the website to microtarget and compromise the security and privacy of the user. Currently, there are no effective transparency or audit mechanisms in place for the web, leaving users vulnerable to attacks. A client visiting a website has no guarantee that the code it receives today is the same as yesterday, or the same as other visitors receive.

Despite advances in web security, the ephemeral nature of web applications brings a significant challenge to audit the web application code on the client side. Unlike curated software repositories that undergo security analysis before deployment, web applications are rendered dynamically in the browser environment. Since browsers focus primarily on performance to render the content immediately, applying comprehensive security analysis on the code after delivery is very challenging before an attack actually takes place in the browser. To address this issue, it is essential to perform the security analysis on active content before the delivery to the browser like in the curated software repositories. However, in the current design of the web, even if a security analysis takes place before the delivery, there is no mechanism to verify that the audited code is actually delivered to the client. The server can decide to deliver different codes in runtime without any notice to the client. Hence, the code auditing becomes ineffective to protect users without a reliable verifiability mechanism.

This lack of verifiability in active content can lead to security breaches, as malicious servers can use this vulnerability to microtarget the user by for example injecting malicious code to the client side environment without any indication to the user. In our second work, we introduce Accountable JavaScript protocol that aims to mitigate this issue of verifiability. Using this protocol, users can verify that the active content they receive from the server is consistent across all users of the website. The users can ensure that they received a specific version of the code from the server and they can prove it. While auditing the code is beyond the scope of this work, it enables effective code auditing in the web.

Since Accountable JavaScript is a client-side protocol, it cannot prevent the server to deliver unaudited code to the user. However, it can hold the developer of the code accountable when it is the case. The Accountable JavaScript protocol generates an undeniable record of the code delivery, by using digital signatures that is the proof of the version of the code delivered to the user. If any inconsistencies occur between the intended and delivered code, the verifiable record can be used to hold the developer of the code responsible. As a result, accountability will create a deterrence mechanism for developers. The developers will be more careful about the code they deliver and adhere to the best practices knowing that malicious activities can be traced back to them.

More concretely, Accountable JavaScript employs cryptographic operations to enable users to verify that the active content they download is consistent across all users of a website and that the code adheres to the standards set by protocol on the client side. To implement this approach, we propose that the web application developers, who choose

to opt in to the Accountable JavaScript protocol, provide a signed manifest enumerating all the active content in their web applications. The signed manifest files are stored in publicly readable transparency logs. When a browser requests a URL and downloads the resulting HTML document from the web server, the web server also provides the corresponding signed manifest for this URL. Then, the browser performs several checks to ensure accountability: it verifies that the active content provided by the server matches the manifest entry, the manifest is correctly signed and it is consistent with the entry in the transparency logs. Moreover, the practical implementation of Accountable JavaScript continuously monitors the web application on the client side to check whether the code behaves exactly as declared by the developer. This monitoring enables to detect any unexpected changes to the active content. For instance, undeclared third party active content dynamically imported to the web application will make Accountable JavaScript report an issue to the user.

With Accountable JavaScript, users are able to verify the integrity and authenticity of the active content on the website they visit. Thus, they are able to ensure that the active content runs exactly as intended by the web application developer. Consequently, integration of Accountable JavaScript enhances the trust in the active content, using accountability. The system applies to the Trollthrottle protocol but also any other web-based platforms in the web ecosystem.

However, while Accountable JavaScript addresses the trust in active content, it raises another trust issue related to the browser itself. Users have to trust that the browser implements operations correctly and executes the active content faithfully. We try to address this concern in Chapter 5, Formal Browser Model for Security Analysis work.

Enhancing the Trust in the Browser

Web browsers serve as key gateways to the internet, relying heavily on protocols and guidelines set by Request for Comments (RFC) documents. While RFCs aim to standardise browser behaviour for a secure and consistent user experience, real-world browser implementations sometimes vary depending on optimisations throughout the code base with respect to security, privacy, user experience and others. This results in potential inconsistencies and vulnerabilities in different browser implementations. The challenge is increased by the ever-changing nature of the web, which requires constant updates to both RFCs and browsers. Furthermore, browsers are complex systems, with multiple interdependent functionalities like URL parsing, HTML interpretation, security protocols and others. A minor change in one component can have cascading effects on others, leading to unforeseen vulnerabilities.

A browser with security vulnerabilities can be used by malicious actors for cyber attacks against users. They can gain unauthorised access to the user's device, steal sensitive information, display misleading content or even exploit larger web-based systems. In Chapter 5, we focus on systematically identifying potential security vulnerabilities in web browsers, relative to a set of RFCs that we will just call "the RFCs" in the follow-up.

We formally model the browser features, with respect to RFCs, as a state transition system in the Alloy language and apply model checking to find out whether the system

satisfies given security properties. If the property is not satisfied, then the model checking provides a 'counterexample' that is the sequence of transitions that leads to a state in which this property is not satisfied. Then, to further analyse this vulnerability in practice, we use the sequence of transitions that leads to the error state in the model to automatically generate test case scenarios that can be simulated in the browser to check whether the error state could also be reached in the browser. This allows us to examine the real-world implications of the vulnerability. The whole process, except the modelling, is fully automated. The user only needs to execute a desktop Java application.

Our framework is built with adaptability in mind, easily accommodating changes or additions to browser functionalities, security policies and APIs. Due to its modular architecture, the integration of new features is a simple task. Furthermore, it can facilitate exhaustive testing of security and privacy implications in the browser. Each module in the formal model can communicate with others, enabling a comprehensive validation of vulnerabilities arising from interactions between different browser components. This design not only streamlines the incorporation of updates but also contributes to ongoing improvements in browser security and privacy.

With our framework, browser developers and testers can ensure that their browser's implementation aligns as closely as possible with the ideal security specifications set by the RFCs. They can identify potential vulnerabilities and then even check whether their improvements to these vulnerabilities also lead to any attacks using the framework. Therefore, our framework raises the trust on browsers, as users can be more confident that their browser is secure and reliable knowing that the browser has been tested and validated against ideal security standards.

2.1 Outline

In this chapter, Chapter 2, we delved into the problem of trust in the web that forms the motivation of this thesis, we stated the existing challenges in the field and we introduced our contributions that approach to the problem. Next, we present our first work Trollthrottle in Chapter 3 that mainly addresses the astroturfing problem in social media and solves censorship problems with accountability. In the following chapter, Chapter 4, we describe our second work Accountable JavaScript that solves the trust issue on the active content that Trollthrottle and alike web-based platforms encounter. Subsequently, in Chapter 5, we present our final work Browser model for Security Analysis which is a comprehensive approach to tackle the trust problem related to the browser itself such as ensuring that it implements operations correctly so that web-based platforms like Accountable JavaScript can trust. Then, in Chapter 6, we discuss our conclusions and possible future works of this thesis. After the conclusion, we have two more chapters in the Appendix. In Chapter 7, we put the Trollthrottle Appendix forward that includes cryptographic proofs of the protocol, an extension to the protocol and the formal model of the protocol in Tamarin Prover. Then, in Chapter 8, we present formal verification of the Accountable JavaScript protocol and evaluation details on various case studies introduced in the main part.

3

Trollthrottle, Raising the Cost of Astroturfing

Astroturfing, i.e. the fabrication of public discourse by private or state-controlled sponsors via the creation of fake online accounts to put narratives forward, has become incredibly widespread in recent years. The anonymity of the cyberspace makes astroturfing much cheaper and it can even be mechanised with social bots. Prior efforts without deanonymising the participants have not yet proven effective.

In this chapter, we present an online protocol that limits the number of comments a single person can post on participating websites in total to raise the cost of astroturfing. Using direct anonymous attestation scheme and a public ledger, the user is free to choose any nickname, but the number of comments is aggregated over all posts on all websites, no matter which nickname was used. Thus, if the threshold is τ, the cost of posting n comments for an astroturfer is the cost of acquiring $\lceil \frac{n}{\tau} \rceil$ identities, be it by employing personnel, by bribery or by identity theft.

To present Trollthrottle, we firstly discuss how we approach the problem; thereafter we define the protocol in terms of a set of cryptographic algorithms which we also analyse for security. We then consider caveats of implementing this system that is not covered in the cryptographic model: the incentive structure for the participants and how to perform verification. What's more, we also present our prototype implementation for the protocol, we demonstrate its deployability by retrofitting it to a popular news aggregator website Reddit and we evaluate the cost of deployment of such a system for the scenario of:

- a national newspaper receiving around 100K messages per day,
- an international newspaper receiving around 168K messages per day,
- and the Reddit itself receiving around 3.5M messages per day.

3.1 Problem Statement

Astroturfing describes the practice of masking the sponsor of a message in order to give it the credibility of a message that originates from 'grassroots' participants (hence the name). Classic astroturfing involves paid agents fabricating false public opinion surroundings, e.g. some product. The anonymity of the cyberspace makes astroturfing very inexpensive; now, it can even be mechanised [63]. This form of astroturfing, also called 'cyberturfing', is a Sybil attack that exploits a useful, but sometimes fallible heuristic strategy in human cognition: roughly speaking, the more people claim something, the improved judgement of credibility [99, 96]. In the wake of the 2016 US elections, Twitter identified, '3,814 [..] accounts' that could be linked to the Internet Research Agency (IRA), a purported Russian 'troll factory'. These accounts 'posted 175,993 Tweets, approximately 8.4% of which were election-related' [166], which is likely only a fraction of the overall activity. This influence comes at a modest price, as the IRA had a $1.25M budget in the run-up to the 2016 presidential election [48] and only 90 members of staff producing comments [35].

The everyday political discourse has also suffered. Many newspapers have succumbed under the weight of moderation, e.g. the New York Times [61]. Some newspapers decided to move discussion to social media [195], where they only moderate a couple of

stories each day and leave out sensitive topics such as migration altogether [133]. Kumar et. al. show that many popular news pages have hundreds of active sock puppets, i.e. accounts controlled by individuals with at least one other account [95]. The New York Times, one of the largest newspapers worldwide, has put serious effort and technological skills into moderating discussion, but ultimately, they had to give up. In mid-2017, they reported how they employ modern text analysis techniques to cluster similar comments and moderate them in one go. At that point in time, they had 12 members of staff dedicated to moderation, handling a daily average of 12,000 comments [102]. Despite the effort and expertise put into this, they had to give up three months later, deactivating the commenting function on controversial topics [61].

Even if troll detection could be automated, e.g. via machine learning, as soon as the detection algorithm becomes available to attackers, numerous techniques permit the creation of adversarial examples [100] to evade classifiers. Fundamentally, astroturfing does not even rely on automated content generation and can be conducted by paid authors in countries with low labour cost: e.g. the so-called 50-cent party, a group of propagandists sponsored by China, was named after the remuneration they receive per comment [115].

3.2 Overview

In this work, we propose a cryptographic protocol that permits throttling the number of comments that a single user can post on all participating websites *in total*. The goal is raising the cost of astroturfing: if the threshold is τ, the cost of posting n comments is the cost of acquiring $\lceil \frac{n}{\tau} \rceil$ identities, be it by employing personnel, by bribery or by identity theft. Our proposal retains the anonymity of users and provides accountability for censorship, i.e. if a user believes her comment ought to appear on the website, she can provide evidence that can be evaluated by the public to confirm misbehaviour on the part of the website.

Our approach is orthogonal to detection by content. If we can limit the number of messages to a certain threshold τ that each physical person can send per day, bots become largely useless, and troll farms need to pay, bribe or steal the identity from sufficiently many actual people to send messages in their name. Besides raising the cost, this also raises the probability of larger operations being detected.

This approach comes at a cost for honest users, as it imposes a bound on power users, too. We will discuss this issue in-depth in Section 3.8.

We want to establish a system that serves a set of websites W_1 to W_n and that, for each user U, provides the following guarantees:

I. If the number of messages a user posts to *any* of the websites exceed τ, all subsequent messages should be discarded.

II. A user is free to choose a virtual identity (nickname) of his choice for any comment. His comments are unlinkable, even if one or more websites conspire against him.

III. A website should be accountable for censorship: should it choose not to display a comment, the user is able to provide a piece of verifiable evidence for the public that this comment was withheld, without revealing his identity.

IV. The trust placed in the organisation running the system should be limited.

V. For each party in the system, there should be a clear incentive to participate.

Our approach is based on direct anonymous attestation scheme (DAA) of Brickell and Li [27]. In DAA, there is an issuing party that distributes membership credentials to signers (in our case users) that it considers legitimate. Each signer can prove membership by signing data: a valid DAA signature guarantees that a valid signer signed this data, but does not reveal the signer's identity. DAA schemes can also be seen as group signature schemes that do not allow the issuing party to identify the signer of the message, a feature known as the *opening*.

To avoid a single point of trust, the identification of the user is not only a matter of the issuer, who is likely to be the provider of this service. Instead, an agreed upon set of verifiers establishes the legitimacy of users, i.e. they are real people and that they have not received a DAA key before. We will discuss how the issuer and the verifiers keep each other honest in Section 3.4.7. To provide accountability for censorship, a public ledger keeps record about the comments that websites ought to publish.

Thus, the following parties (shown in the Figure 3.1) cooperate in TrollThrottle: an issuer I, who issues DAA keys, a set of verifiers V, who verify the user's identities, a set of users U, who create DAA signatures from their comments, a public append-only ledger L, who records these signatures, and a set of websites, who verify these signatures and are bound to publish comments whose signatures appear on the ledger.

In DAA, a signature can be created and verified with respect to a so-called *basename*. Signatures created by the same user with the same basename can be linked. This is the key feature to achieve throttling. Within a commenting period t, e.g. a day, only signatures with a basename of the form (t, seq) are accepted, where seq is a sequence number between 1 and the desired threshold τ. If a user signs two messages with the same basename, they can be linked and discarded by the website. Hence a user can create at most τ signatures that are unlinkable to each other. A valid DAA signature assures the website that a valid user signed this comment, but neither the website, nor the issuer or the verifier learns who created the comment, or which other comments they created.

By storing the signatures on the ledger L, the websites can :

(a) enforce a global bound, and

(b) provide accountability for censorship by promising to represent all comments to this website that appear in the ledger. If a website does not publish a user's comment, it must have sufficient grounds for censorship.

We build on Brickell and Li's DAA scheme [27] for its efficiency, but extend it with various features to make TrollThrottle more efficient (TrollThrottle Appendix Section 7.1), more secure (Section 3.3.2), more practical (Section 3.4.4), and more resistant against compromise (Section 3.4.5)

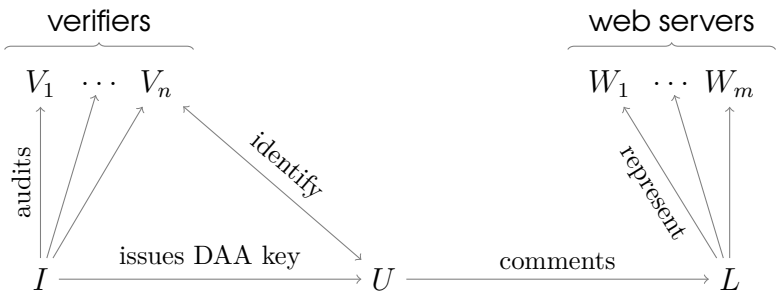

Figure 3.1: TrollThrottle parties

We took care to explicitly devise a clear system of incentives for all participating parties. Part of this system is a pseudo-random audit process to ensure honest behaviour, which we have formally verified.

We show that this protocol, TrollThrottle, can be retrofitted to existing websites. We set up a forum[1] on Reddit that demonstrates our proposal. We also compute the additional cost of operation incurred by our protocol by simulating user interaction for three real-life scenarios: an international newspaper, a nationwide publication and all comments posted on Reddit in one day. In the newspaper case, the computational overhead incurs a cost of about $1.20; for the whole of Reddit, $3.60 is sufficient.

As a by-product and second contribution, we extend the notion of direct anonymous attestation (DAA) by proposing two features with applications outside our protocol. Both are already supported by an existing DAA scheme by Brickell and Li [27]. First, updatability, which means that the issuer can non-interactively update the users' credentials. This allows for easy key rollover in the mobile setting and for implicit revocation of credentials by not updating them (old credentials invalidated). Second, instant linkability, which means that each signature contains a message-independent pseudonym that determines whether two signatures can be linked. This allows to efficiently determine whether a signature can be linked to any existing signature within a given set.

3.2.1 Accountable Commenting

We assume the issuer I is known to all users and websites; the verifiers V are known to the issuer and all websites; and the public ledger L is known to all participants in the protocol. The ledger can be implemented using a consensus mechanism between the websites and some trusted representatives of civil society (e.g. via Tendermint [78] or Practical Byzantine Fault Tolerance (PBFT) [36]), or open consensus mechanisms like blockchains. We will formalise our approach in terms of probabilistic polynomial time algorithms (PPT) and an interactive protocol.

Definition 1 (Accountable commenting scheme). *An accountable commenting scheme consists of a tuple of algorithms* (Setup, KeyGen, Comment, Verify, Claim,

[1] `https://old.reddit.com/r/trollthrottle/`

VerifyClaim) *and an interactive protocol* (Join − Issue)*. The algorithms and the protocol are specified as follows.*

Setup(1^λ) models the generation of a setup parameter ρ used by all participants from the security parameter 1^λ. This parameter is an implicit argument for the other algorithms, but we omit it for brevity. The issuer I invokes KeyGen(ρ) to generate its secret key sk_I and public key pk_I from this parameter.

The issuing procedure $\langle \mathsf{Join}(pk_\mathrm{I}, U) \leftrightarrow \mathsf{Issue}(sk_\mathrm{I}, \mathsf{ver}, U)\rangle$ is an interactive protocol between I and a new user (identified with U) that has not registered so far. At the end of the protocol, the user receives a credential $cred_U$ and a secret key sk_U. We abstract away from the verifiers by giving the issuer access to a read-only database ver such that $\mathsf{ver}[V, U] \in \{0, 1\}$ is 1 iff the verifier V confirms the identity of a user. In Section 3.4.3, we present and verify a protocol to implement and audit this verification step.

The commenting procedure is split into four PPT algorithms, Comment for U to generate comments that she sends to the ledger, Verify for W to verify that a comment on the ledger should be displayed, Claim for U to generate a claim that a valid comment on the ledger ought to be published, and VerifyClaim for the public to verify that said claim is valid.

Comment($pk_\mathrm{I}, sk_U, cred_U, dom, m$) is executed by U, who knows the issuer's public key pk_I, its own secret key sk_U and credentials $cred_U$. U chooses a basename $dom \in \{0,1\}^*$ and a message $m \in \{0,1\}^*$ and obtains a signed comment γ and a pseudonym nym, both of which are stored on the ledger. The basename determines a given user's nym, so that anyone can check whether two comments were submitted with the same basename by checking their respective nyms for equality. This is a key feature: in TrollThrottle, all basenames have to be of the form $\langle t, i\rangle$ for a commenting period t and an integer $i \in \{1, \ldots, \tau\}$. Hence there are at most τ unique basenames within t, and thus at most τ nyms per sk_U and t.

Verify($pk_\mathrm{I}, nym, dom, m, \gamma$) can be computed by any website that has access to the issuer's public key pk_I, the comment on the ledger γ, pseudonym nym, domain dom (which can be determined by trial and error) and a message $m \in \{0,1\}^*$ received from the user. If the output is 1, the comment γ is valid w.r.t. m, and m needs to be displayed by the website. Should W fail to display m, the user computes Claim($pk_\mathrm{I}, sk_U, cred_U, dom, m, \gamma$) on the same data as before. The output *evidence* can be publicly verified using the algorithm VerifyClaim($pk_\mathrm{I}, dom, m, \gamma, evidence$), which outputs 1 iff *evidence* and the ledger entry γ prove that m ought to be displayed in the commenting period indicated by dom.

3.3 Protocol Definition

Before we present TrollThrottle as an instance of an accountable commenting scheme, we introduce the necessary cryptographic notions.

3.3.1 Direct Anonymous Attestation Scheme

We follow the DAA definition proposed in [27]. We slightly simplify their model, as our protocol's computations are performed by a single host and not split between a TPM and an untrusted device.

A DAA scheme consists of four PPT algorithms :

$$(\mathsf{Setup}_{\mathsf{DAA}}, \mathsf{Sign}_{\mathsf{DAA}}, \mathsf{Verify}_{\mathsf{DAA}}, \mathsf{Link}_{\mathsf{DAA}})$$

and an interactive protocol:

$$(\mathsf{Join} - \mathsf{Issue}_{\mathsf{DAA}})$$

between parties: an issuer I, a verifier V and a signer S. In our case, the websites take the role of the verifiers, and the users the role of the signer.

$\mathsf{Setup}_{\mathsf{DAA}}(1^\lambda)$ is run by I; based on the security parameter 1^λ, it computes the issuer's secret key sk_I and public key pk_I, including global public parameters. $\mathsf{Join} - \mathsf{Issue}_{\mathsf{DAA}}$ is an interactive protocol between I and S to provide credentials issued by I to S. It consists of sub-algorithms $\mathsf{Join}_{\mathsf{DAA}}$ and $\mathsf{Issue}_{\mathsf{DAA}}$. S executes $\mathsf{Join}_{\mathsf{DAA}}(pk_I, sk_S)$ on input pk_I and sk_S to obtain the commitment com. We slightly alter the original definition and assume that instead of sampling this key inside the algorithm, S provides the key as an input. I executes $\mathsf{Issue}_{\mathsf{DAA}}(sk_I, \mathsf{com})$ to create a credential $cred_S$ that is associated with sk_S and sent to S. Note the key of S remains hidden from I.

$\mathsf{Sign}_{\mathsf{DAA}}(sk_S, cred_S, dom, m)$ is executed by S to create a signature σ for a message m w.r.t. a basename dom, which is optionally provided by V. If $dom \neq \perp$, signatures created by the same signer can be linked. In the original definition, $\mathsf{Sign}_{\mathsf{DAA}}$ also takes a nonce n_V as input, which the signature verifier provides to prove freshness of the message. Brickell and Li make this nonce explicit but it can be part of the signed message. What's more, we do not make use of such a nonce in our system and assume that if freshness of a signature is required then this nonce will be part of the signed message.

$\mathsf{Verify}_{\mathsf{DAA}}(pk_I, m, dom, \sigma, RL)$ is a deterministic algorithm run by V on a message m, a basename dom, a signature σ, and a revocation list RL to determine if a signature σ is valid. In the original definition, I stores revoked secret keys in the revocation list RL; signatures created with a revoked secret key are not valid.

$\mathsf{Link}_{\mathsf{DAA}}(\sigma_0, \sigma_1)$ is a deterministic algorithm that determines with overwhelming probability whether signatures σ_0 and σ_1 were created by the same signer with the same basename $dom \neq \perp$. It outputs 1 if the signatures are linked, 0 for unlinked and $\perp$ for invalid ones.

DAA features Brickell and Li's DAA scheme [27] has the following security properties:

- *Correctness:* if an honest signer's secret key is not in the revocation list RL, then, with overwhelming probability, signatures created by the signer are accepted and correctly linked by an honest verifier.

- *User-controlled-anonymity:* a PPT adversary has a negligible advantage over guessing in a game where she has to distinguish whether two given signatures associated with different basenames were created by the same signer or two different signers.
- *User-controlled-traceability:* no PPT adversary can forge a non-traceable yet valid signature with $dom \neq \perp$ without knowing the secret key that was used to create the signature, or if her key is in the revocation list RL. The basenames in TrollThrottle are always different from $\perp$ (see Section 3.3.3).
- *Instant-linkability:* we add this feature to DAA. There is a deterministic poly-time algorithm NymGen s.t. $\mathsf{NymGen}(sk_S, dom)$ generates a nym that is otherwise contained in the signature, and two nyms are equal if and only if the corresponding signatures are linkable.

3.3.2 Zero Knowledge

The user creates non-interactive proofs of knowledge to show that her key was honestly generated. Let $\mathcal{R}$ be an efficiently computable binary relation. For $(x, w) \in \mathcal{R}$, we call x a *statement* and w a *witness*. Moreover, $L_{\mathcal{R}}$ denotes the language consisting of statements in $\mathcal{R}$, i.e. $L_{\mathcal{R}} = \{x \mid \exists w : (x, w) \in \mathcal{R}\}$.

Definition 2. *A non-interactive proof of knowledge system* Π *consists of the following three algorithms* $(\mathsf{Setup}, \mathsf{CreateProof}, \mathsf{VerifyProof})$.

$\mathsf{Setup}(1^\lambda)$**:** *on input security parameter* 1^λ*, this algorithm outputs a common reference string* ρ*.*

$\mathsf{CreateProof}(\rho, x, w)$**:** *on input common reference string* ρ*, statement* x *and witness* w*; this algorithm outputs a proof* π*.*

$\mathsf{VerifyProof}(\rho, x, \pi)$**:** *on input common reference string* ρ*, statement* x *and proof* π*; this algorithm outputs either* 1 *or* 0*.*

3.3.3 Trollthrottle Protocol

We define TrollThrottle protocol based on the accountable commenting scheme (Section 3.2.1). Besides an instantly linkable DAA scheme (Section 3.3.1), we assume a collision-resistant hash function h and a non-interactive proof of knowledge system (Section 3.3.2) for the relation:

$$((\mathsf{com}, pk_{I,\mathsf{DAA}}), (sk_{S,\mathsf{DAA}})) \in \mathcal{R}_{\mathsf{Join}} \iff \mathsf{com} \xleftarrow{\$} \mathsf{Join}_{\mathsf{DAA}}(pk_{I,\mathsf{DAA}}, sk_{S,\mathsf{DAA}})$$

We assume that the witness for the statement $(\mathsf{com}, pk_{I,\mathsf{DAA}})$ contains the random coins used in $\mathsf{Join}_{\mathsf{DAA}}$.

Definition 3. *TrollThrottle Protocol*

The algorithms Setup *and* KeyGen *generate the issuer's DAA keys and parameters for the non-interactive zero-knowledge proof of knowledge for the relation* $\mathcal{R}_{\mathsf{Join}}$*.*

$\mathsf{Setup}(1^\lambda)$ - *compute the parameters for the zero-knowledge proof of knowledge:*

$$\rho_{\mathsf{Join}} \xleftarrow{\$} \mathsf{Setup}_{\mathsf{ZK}}(1^\lambda)$$

and output:

$$\rho = (1^\lambda, \rho_{\mathsf{Join}})$$

$\mathsf{KeyGen}(\rho)$ - *execute:*

$$(pk_{I,\mathsf{DAA}}, sk_{I,\mathsf{DAA}}) \xleftarrow{\$} \mathsf{Setup}_{\mathsf{DAA}}(1^\lambda)$$

set and return:

$$pk_{\mathrm{I}} = pk_{I,\mathsf{DAA}}$$
$$sk_{\mathrm{I}} = (pk_{I,\mathsf{DAA}}, sk_{I,\mathsf{DAA}})$$

The $\mathsf{Join}-\mathsf{Issue}$ *protocol closely resembles the* $\mathsf{Join}-\mathsf{Issue}_{\mathsf{DAA}}$ *protocol of the DAA scheme with two main differences. Firstly, the user provides her secret key as input to the* Join *algorithm. This is for practical reasons: in Section 3.4.6, we explain how this key can be recomputed from a pair of login and password using a key derivation function when a user switches machines. The second difference is the* Π_{Join} *proof created by the user to ensure honestly generated secret keys and allow the security reduction to extract secret keys generated by the adversary. We remark that during the* $\mathsf{Join}-\mathsf{Issue}$ *protocol, the user communicates with a publicly known verifier who validates her identity and confirms it to* I*. In Section 3.4.3, we present a protocol for obtaining this confirmation and running a pseudo-probabilistic audit of* V *by* I*.*

$\mathsf{Join}(pk_{\mathrm{I}}, sk_U, U)$ - *let :*

$$pk_{\mathrm{I}} = pk_{I,\mathsf{DAA}} \quad \textit{and} \quad sk_U = sk_{S,\mathsf{DAA}}$$

run:

$$\mathsf{com} \xleftarrow{\$} \mathsf{Join}_{\mathsf{DAA}}(pk_{I,\mathsf{DAA}}, sk_{S,\mathsf{DAA}})$$

compute proof:

$$\Pi_{\mathsf{Join}} = \mathsf{CreateProof}(\rho_{\mathsf{Join}}, (\mathsf{com}, pk_{I,\mathsf{DAA}}), sk_{S,\mathsf{DAA}})$$

send $(\mathsf{com}, \Pi_{\mathsf{Join}})$ *to the issuer and receive* $cred_U$*, then return:* $(cred_U, sk_U)$*.*

$\mathsf{Issue}(sk_{\mathrm{I}}, \mathsf{ver}, U)$ - *parse :*

$$sk_{\mathrm{I}} = (pk_{I,\mathsf{DAA}}, sk_{I,\mathsf{DAA}})$$

receive $(\mathsf{com}, \Pi_{\mathsf{Join}})$ *from the user.*

Abort if the proof is invalid, i.e.

$$\mathsf{VerifyProof}(\rho_{\mathsf{Join}}, (\mathsf{com}, pk_{I,\mathsf{DAA}}), \Pi_{\mathsf{Join}}) = 0$$

Otherwise, execute the $\mathsf{Issue}_{\mathsf{DAA}}$ *protocol with input* $(\mathsf{com}, sk_{I,\mathsf{DAA}})$*, receiving credentials* $cred_U$*. Send* $cred_U$ *to the user.*

Comment *creates the information that U stores on the ledger, consisting of the signed comment γ and pseudonym nym. To provide accountability for censorship, U sends the signature to the ledger, which notifies the website W. At this point, W must publish the comment $\gamma = (\sigma, nym, dom, m)$ as long as the signature σ, message and dom are deemed valid, and nym appears exactly once on the ledger.*

With the validity requirement on the basename dom and the ability to detect repeated basenames in the ledger, we can easily implement the desired throttling mechanism. Let τ be a threshold for some time frame (e.g. a day) and let t mark the current period. Then, a valid dom is of the form (t, seq) with $seq \in \{1, \ldots, t\}$. The sequence number seq in dom is allowed to arrive out-of-order, but it cannot be larger than τ. The throttling is ensured because there exist only τ valid basenames per commenting period and thus only τ valid nym per (sk_U, dom).

$\mathsf{Comment}(pk_{\mathrm{I}}, sk_U, cred_U, dom, m)$ *- set and return:*

$$\gamma = (\sigma, nym, dom, h(m))$$

where :

$$\sigma \xleftarrow{\$} \mathsf{Sign}_{\mathsf{DAA}}(sk_U, cred_U, dom, h(m))$$

$$nym \xleftarrow{\$} \mathsf{NymGen}(sk_U, dom) = \mathsf{NymExtract}(\sigma)$$

$\mathsf{Verify}(pk_{\mathrm{I}}, nym, dom, m, \gamma)$ *- Parse*

$$\gamma = (\sigma, nym, dom, h^*)$$

and :

$$pk_{\mathrm{I}} = pk_{I,\mathsf{DAA}}$$

then output 1 *iff :*

$$\mathsf{Verify}_{\mathsf{DAA}}(pk_{I,\mathsf{DAA}}, h^*, dom, \sigma, RL_{\emptyset}) = 1$$

$$h(m) = h^*$$

$$\mathsf{NymExtract}(\sigma) = nym$$

$$\mathsf{VerifyBsn}(\sigma, dom) = 1$$

If W refuses to publish the comment, then U can use Claim *to claim censorship and provide the entry on the ledger γ and m as evidence to the public that m ought to be displayed. The public checks the same conditions that W should have applied. Part of this check is to interpret a common agreement for moderation, which we discuss in more detail in Section 3.4.6, but do not model explicitly. We show the security of this protocol in the cryptographic model, see TrollThrottle Appendix Section 7.2.*

$\mathsf{Claim}(pk_{\mathrm{I}}, sk_U, cred_U, dom, m, \gamma)$ - *return*

$$evidence = \gamma$$

$\mathsf{VerifyClaim}(pk_{\mathrm{I}}, dom, m, \gamma, evidence)$ - *parse*

$$\gamma = (\sigma, nym, dom, h)$$

and output 1 iff :

$$\mathsf{Verify}(pk_{\mathrm{I}}, nym, dom, m, \gamma) = 1$$

3.4 Practical Implementation

A deployable system needs more than just a cryptographic specification, but a system of incentives and checks. First, we discuss what methods for identity verification are available. We detail how to identity verification can be deferred to the verifiers and misbehaviour can be detected using pseudo-probabilistic audits. A realistic system also has to deal with revocation, which we solve by exploiting a novel property called *updatability*. Finally, we discuss questions related to the end user: how moderation is handled and where to store credentials. Table 3.1 summarises the protocol components and their security analysis.

Table 3.1: Overview: security analysis.

components	security analysis
base protocol	cryptographic proof (TrollThrottle Appendix Section 7.2)
encrypted ledger	strictly weakens the attacker
identity verification	formally verified (Section 3.4.3)
revocation	simple hybrid argument (TT. Appendix Section 7.5)
extended protocol	cryptographic proof (TT. Appendix Section 7.3)
storing credentials	trivial modification

3.4.1 Identity providers

The verifiers need to attest that only real people receive digital identities and each person obtains only one. We discuss multiple competing solutions to this problem, none perfect by itself. In combination, however, they cover a fair share of the users for our primary target, news websites.

Identity verification services (IVS): Banks, insurers and other online-only services already rely on so-called identity verification services, e.g. to comply with banking or anti-money laundering regulations. Usually, IVS providers verify the authenticity of claims using physical identity documents, authoritative sources, or by performing ID checks via video chat or post-ID.

Subscriber lists: Newspaper websites are the main targets of our proposal, because of their political and societal relevance and the moderation cost they are currently facing. It is in their interest to provide easy access to their subscribers. Insofar as bills are being paid, they do have some assurance of the identity of their subscribers, so they can use their existing relationship to bootstrap the system by giving access to their customers right away.

Biometric passports and identification documents: Biometric passports are traditional passports that have an embedded electronic microprocessor chip containing information for authenticating the identity of the passport holder. The chip was introduced to enable additional protection against counterfeiting and identity theft. This authentication process can be performed locally (as part of e.g. border control) or against a remote server. Biometric passports are standardised by the International Civil Aviation Organization (ICAO) [84] and issued by around 150 countries [67]. More importantly, even many electronic identification documents are supporting this standard, e.g. the German eID [29].

Our system can easily leverage this infrastructure to authenticate users. Of course, we need to assume that governments are issuing those documents honestly, however, large-scale fraud would have serious repercussions for the issuing government.

These methods can be combined: even if somebody is neither a subscriber of a newspaper nor the owner of a digital passport, they still have the option of identifying to the IVS. We note that a natural person could use two methods (e.g. IVS and subscriber list) to obtain two DAA credentials and thus effectively double her threshold. As we provide a method for revocation (see Section 3.4.3), the verifiers can run private set-intersection protocols (see [131] for an overview) and revoke parties in the intersection.

Table 3.2: Time periods used in the protocol.

name	symbol	purpose	typical duration
epoch	t_e	implicit revocation	one week
billing period	t_b	billing	one month
commenting period	t	throttling	one day

3.4.2 Encrypting comments on the ledger

We distinguish a billing period t_b that is distinct from the commenting period t (see Table 3.2). Assume a CCA-secure public key encryption scheme $(\mathsf{KG_{enc}}, enc, dec)$, a collision-resistant hash function h and a standard existentially unforgeable digital signature scheme $(\mathsf{KG_{sig}}, \mathsf{sig}, \mathsf{ver})$. We apply the accountable commenting scheme from Definition. 3. The output of $\mathsf{Comment}$ is encrypted with a public key $pk_{\vec{W},t_b}$ distributed to all websites participating in the current billing period t_b. Claims need to include the randomness used to encrypt. See Figure 3.2 for the complete message flow.

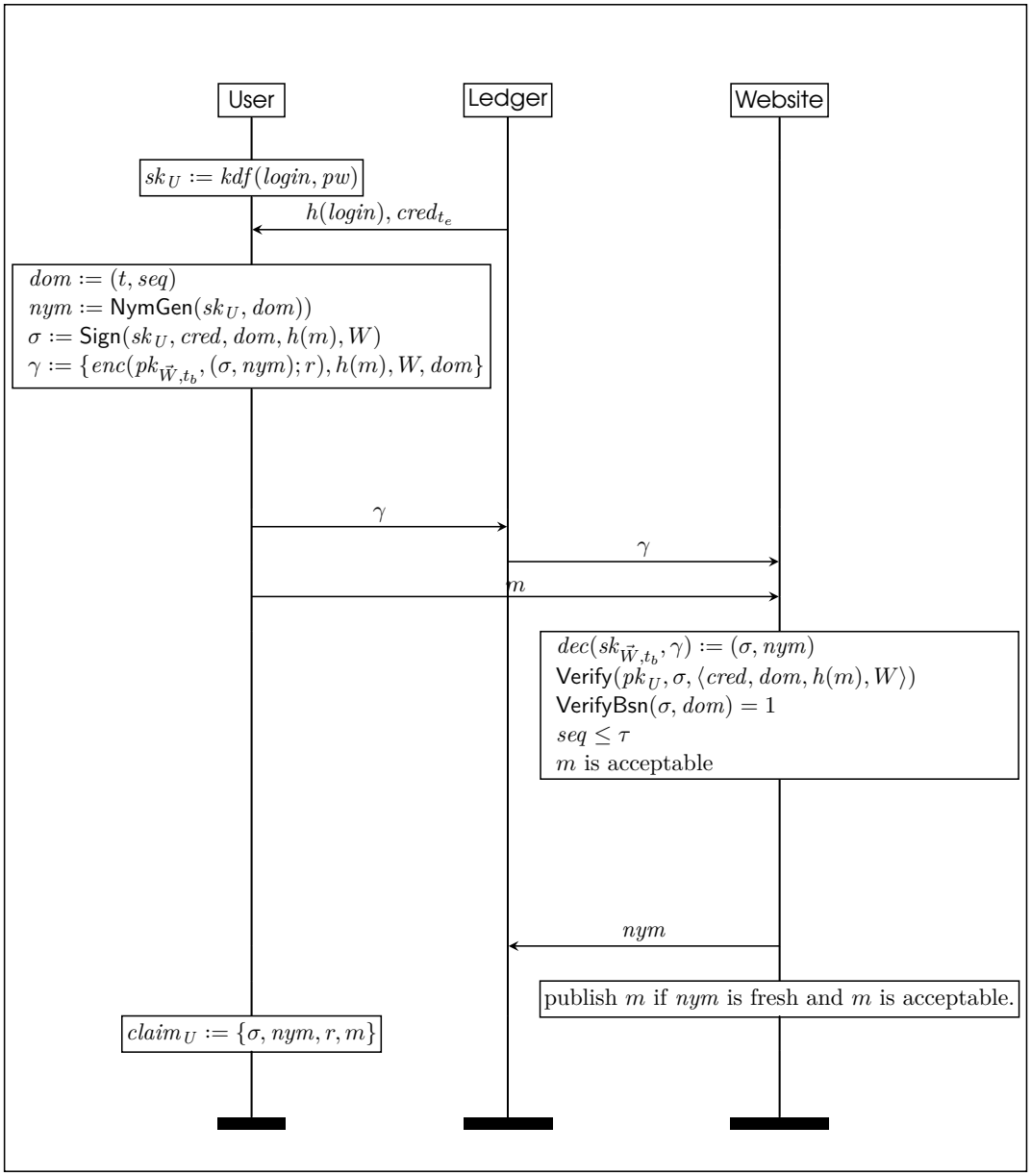

Figure 3.2: Message flow for commenting scheme. Note that in step 0, the user's secret DAA key is restored using a password, see Section 3.4.6, and that entries in the ledger are encrypted, see Section 3.4.7. Furthermore, to save space on the ledger, we identify the comment m by its hash.

.

3.4.3 Deferring identity verification with pseudo-probabilistic auditing

Our security model in TrollThrottle Appendix Section 7.2 abstracts away the communication between verifier and issuer. We propose a protocol to implement this step and formally verify it in the symbolic setting, which is better suited for reasoning

about complex interactions. The protocol (Figure. 3.3) improves privacy by hiding the identity verification process from the issuer and improves accountability by providing a pseudo-random audit.

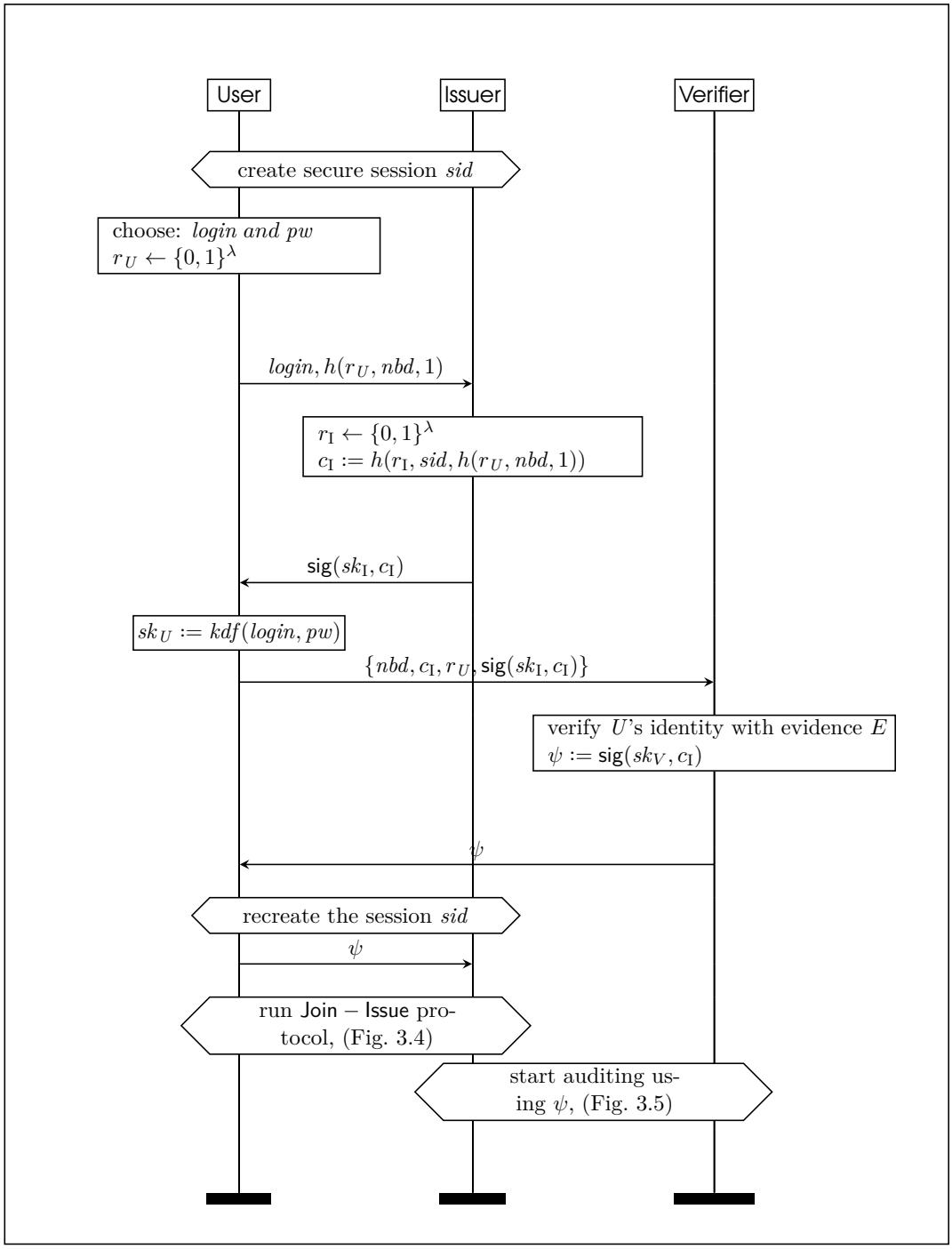

Figure 3.3: Identity verification protocol scheme

.

We assume a collision-resistant one-way hash function h to instantiate a binding commitment scheme. When a user wants to register, the website directs her to the issuer. They run an authentication protocol akin to the ASW protocol for fair exchange[9] where, in the end, U gets V's signature on a commitment c_{I} generated by I. Only with this signature, the issuer runs the $\mathsf{Join}-\mathsf{Issue}$ procedure from Definition. 3 (repeated in Fig. 3.4 for completeness). Note that the ledger distributes the issuer's public key and public parameters. In Section 3.4.4, we explain a revocation mechanism that is based on updating the issuer's public key every epoch and publishing the fresh key in the ledger. U also makes use of the ledger by storing its credentials in case it needs to recover its state (see Section 3.4.6).

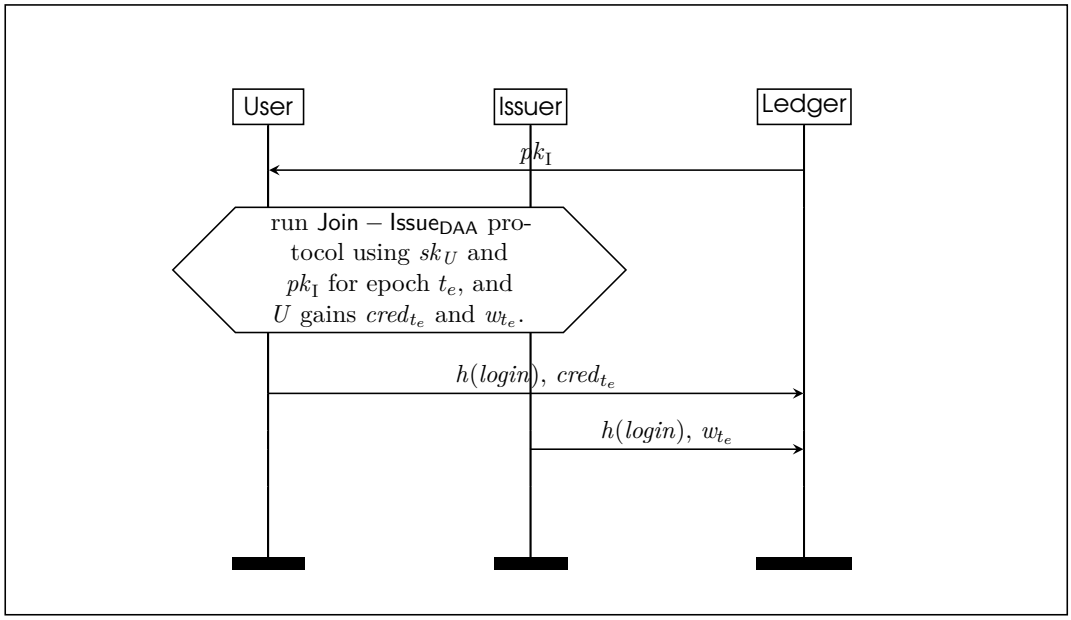

Figure 3.4: $\mathsf{Join}-\mathsf{Issue}$ protocol scheme

.

After verification, I may trigger a pseudo-random audit by sending the previously hidden values sid, r_I in the commitment c_I of the identity verification protocol to V (see Fig. 3.5). If the hash of these values matches the hash of V's signed commitments, an audit is triggered. If we consider a random oracle in place of the hash function, the probability of an audit is $\Pr[\text{audit}] = 2^{-L}$, where L is the number of bits both parties compare. L is agreed upon in advance, to define this probability. Since the nonce r_I has been revealed to V before, I cannot modify the second hash (s') to avoid audit. As the digital signature scheme is existentially unforgeable, I cannot fabricate a valid signature to raise the probability of an audit and to learn something about U. If the session is chosen for audit, V has to hand over the evidence $\{\mathrm{E}\}$ it collected for identification — this is a standard procedure for IVS. If V fails to comply, then I can publish a *claim* and the public can determine whether to audit V.

Presuming that I is honest, the probability that colluding U and V can create n usable fake identities is thus bound by $(1 - \Pr[\text{audit}])^n + \mathsf{negl}(\lambda)$ for some negligible function

$\mathsf{negl}(\lambda)$.

The auditing protocol is very simple cryptographically, but has many possible message interleaving. It is well known that pen-and-paper proofs for such protocols are not only tedious, but also prone to errors. We analyse the protocol in the symbolic model, using the SAPIC process calculus [93] and Tamarin protocol verifier [142]. We formally verify that:

1. Whenever I accepts to run the $\mathsf{Join-Issue}$ protocol with a user, V has validated her identity, unless I or V are dishonest.
2. When determining the need for an audit, neither a dishonest I, nor a dishonest V can predict the value of the other party, unless both are dishonest. Therefore, they cannot trigger or avoid the audit.
3. If the public accepts a claim, then V did indeed receive the values r_I and *sid* and send out ψ (unless V is dishonest and tricks itself into the obligation of an audit). As these values determine both hashes, the public can now decide if an audit was justified.

The verification takes about 10 sec on a 3.1 GHz Intel Core i7 and 16 GB RAM computer. See TrollThrottle Appendix Section 7.4 for the model code.

3.4.4 Revocation

In case U runs the identification protocol a second time with a different V, or simply forgets her password and needs to re-identify, her previous DAA key $sk_{U,\mathsf{DAA}}$ needs to be revoked. But how can U revoke her DAA key if she forgets her password? We circumvent this problem by *implicit revocation*: DAA keys are short-lived by default, but the system can issue new keys without interacting with the user. Keys that are not issued are thus implicitly revoked by the end of their lifetime, which we call the *epoch* (see Figure. 3.2).

At the start of each epoch t_e, I defines a new public key pk_{I,t'_e} which is chosen so that I can recompute all credentials $cred_{t'_e}$ for the new epoch by itself (see Fig. 3.6). At this point, only those DAA keys remain valid, for which such a *cred* is computed, all others are implicitly revoked. If a user forgets her password, she reports to the verifier, who confirms (by means of the commitment c_I) that her old key is invalidated. Starting from the next epoch, she can use her new key. To allow for such mechanism, the DAA scheme has to be structured in a way that I can update her public key and all users 'credentials without any interaction.

Definition 4 (Updateable DAA scheme). *A DAA scheme is* updateable *if:*

1. $\mathsf{Setup}_{\mathsf{DAA}}$ *can be divided into* Setup_1 *and* Setup_2*, s.t.:*
   - $\mathsf{Setup}_1(1^\lambda)$*, outputs a* persistent *group public key* gpk_1
   - $\mathsf{Setup}_2(1^\lambda, gpk_1)$ *outputs an* ephemeral *group public key* gpk_2 *and a secret key* sk_S*, where* $pk_I = (gpk_1, gpk_2)$.
2. *The* $\mathsf{Join-Issue}_{\mathsf{DAA}}$ *protocol consists of two steps* $\mathsf{User}_{\mathsf{DAA}}$ *and* $\mathsf{Issuer}_{\mathsf{DAA}}$*, s.t.:*

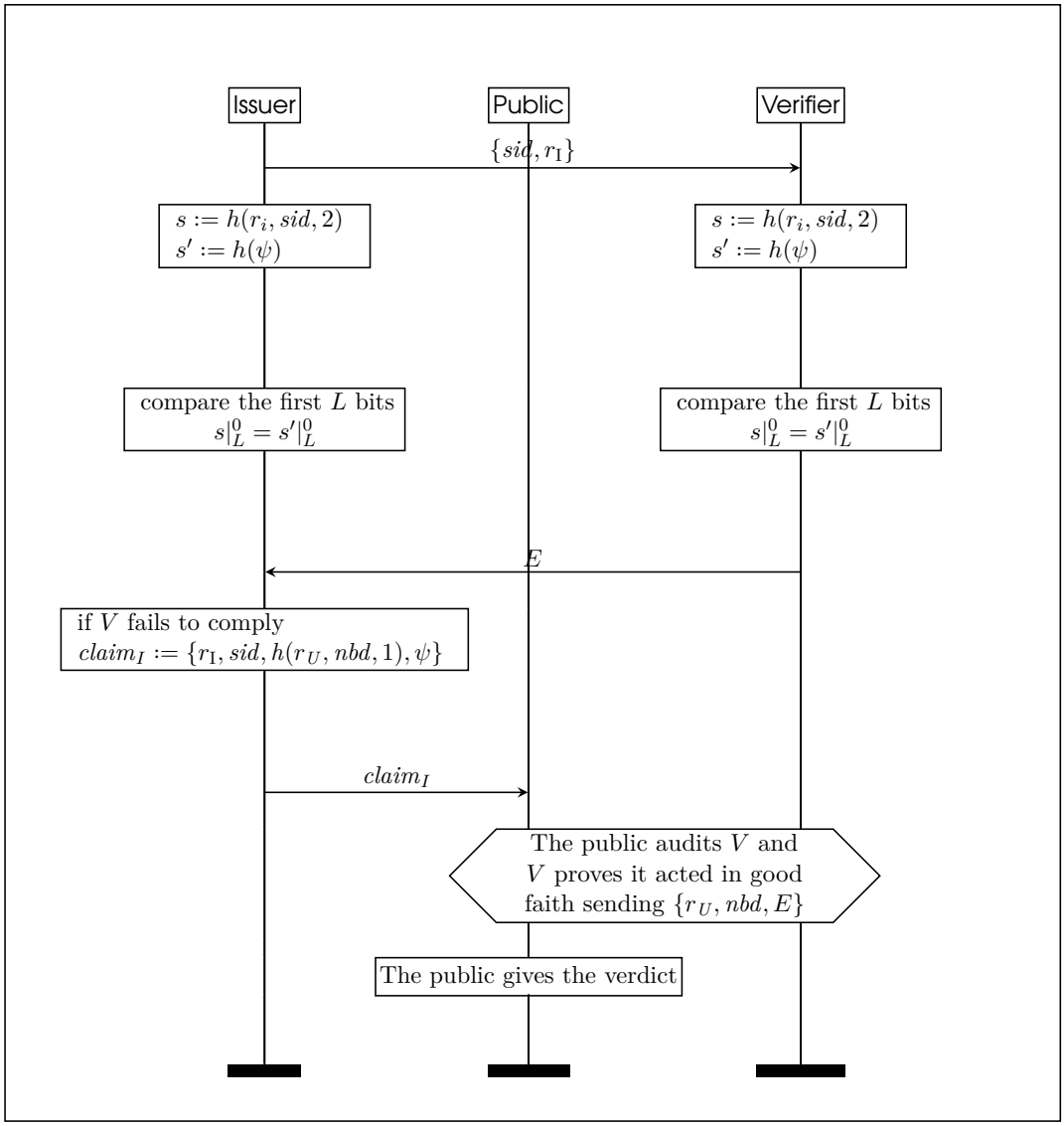

Figure 3.5: Auditing protocol scheme

.

- $\mathsf{User}_{\mathsf{DAA}}(gpk_1, sk_S)$ *outputs an* update u,
- $\mathsf{Issuer}_{\mathsf{DAA}}$ *takes* pk_I, u *and* sk_I *as inputs and outputs valid credentials* $cred_S$ *for secret key* sk_S *with respect to the new* pk_I.

3. Setup_1 *uses only public coins to generate* gpk_1, *i.e. there are no secrets required to generate* gpk_1 *and giving those coins to the adversary only negligibly increases its advantage in the user-controlled anonymity and traceability experiments.*

Brickell and Li's scheme with a minor modification possess these features (see TrollThrottle Appendix Section 7.5 for a formal proof).

Updatability is interesting on its own: it allows for regular, non-interactive key rollovers

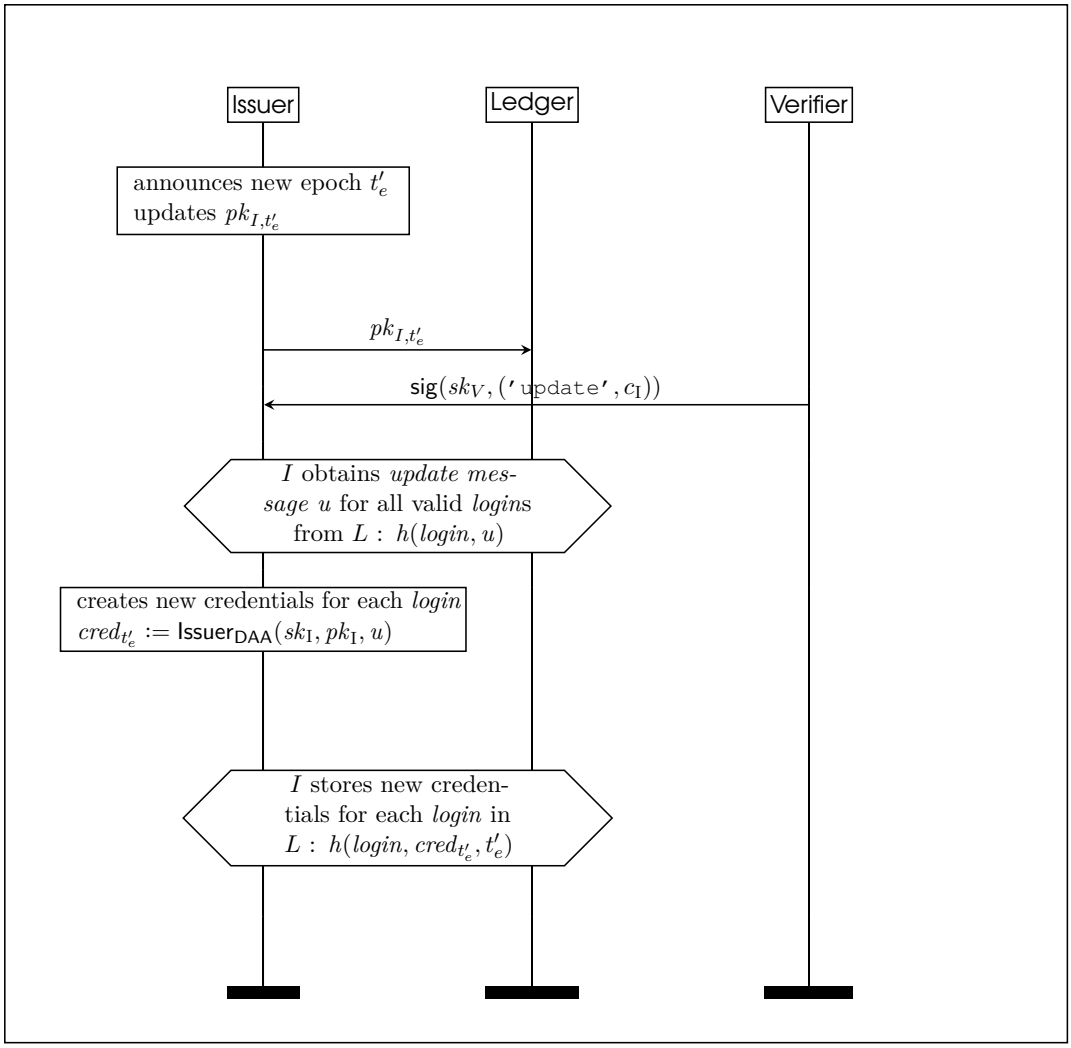

Figure 3.6: Certificate update protocol scheme

.

in DAA. I can create each user's credential offline, so the user can fetch this credential (in encrypted form) at a later point, even if I is offline.

3.4.5 Holding the issuer accountable

In TrollThrottle, a corrupt issuer and verifier can collude to introduce arbitrarily many valid credentials into the system. This form of Sybil attack is difficult to counter while retaining the user's privacy: Without trust in either the verifiers or the issuer, the only way of determining whether a user is legitimate is to have another entity (e.g. the websites, or the public) check this identity — otherwise, the adversary controls all parties involved. Even if done in a pseudo-random manner similar to the auditing procedure in Section 3.4.3, the loss of individual privacy would be considerable.

In TrollThrottle Appendix Section 7.3 we present the extended TrollThrottle protocol to mitigate this issue to the extent possible. Here, for every user that joins, a *genesis block* is added to the ledger. This block is signed by the verifier, which allows the public to tell how many credentials were validated by each verifier. Large-scale fraud could thus be detected through an unusual number of participants coming from a single verifier. This information is public and can be computed by any participant at any time.

During the commenting phase, U downloads a subset of genesis tuples and computes a zero-knowledge proof that her genesis tuple is part of this set. To achieve, e.g. anonymity among 100 users, about 49 KB of data is downloaded once per commenting period. She includes this proof along with the time point at which she queried the list in her DAA signature. In TrollThrottle Appendix Section 7.3.5, we show that for Brickell and Li's scheme [27], we can instantiate a non-interactive proof of knowledge system with proofs that are logarithmic in the number of genesis tuples in the ledger. We show that, in addition to the security properties in TrollThrottle Appendix Section 7.2, no adversary can create comments that cannot be attributed.

3.4.6 Other considerations

Moderation News websites need to moderate comments (see step 8 in Fig. 3.2). This decision is ultimately a human decision, but it should be based on a binding agreement between the participating websites, and in compliance with the laws that apply to them. When U claims censorship, the public has to judge based on the agreement and the content of m.

Storing credentials By default, a cookie or browser plugin may store the credentials, however, many users expect a system to work similarly to a third-party website, where they can log in from a computer of their choice. We, therefore, allow users to restore their identities, by making the users' secret keys e.g., sk_U derivable from their login and password chosen by themselves in the identification process. Hence, we assume there exists an efficiently computable key-derivation function *kdf* [90] that maps to the space of secret keys. Such a function exists for the scheme we use, where the secret key is just an element in $\mathbb{Z}_q^*$.

The secret key sk_U can be recomputed by applying the key-derivation function to login and password, while the DAA credentials *cred* can be recovered from the ledger by querying with the hash of her login. Note that the login should not identify the user on other platforms, otherwise an attacker can use it to check if the user is participating in TrollThrottle. The last value of *seq* can be recovered by using bisection to discover the largest *seq* s.t. $\mathsf{NymGen}(sk_U, (t, seq))$ is on the ledger.

3.4.7 Goals & Incentives

A system like TrollThrottle can only be deployed if all parties have incentives to run it and we build our design on the following incentives.

Websites: Websites have an incentive to get information about the trolls to lessen the burden on moderation and save on personnel. The system requires paying the issuer a

fee for running the infrastructure; hence these costs must be covered by the websites' fee to the issuer. As they benefit from the system, they have an incentive to pay, as long as it is not possible to piggyback on the system. The information necessary to determine whether a user is a troll must hence only be available to paying websites. This can be achieved by using public-key encryption, see below.

Issuer: The issuer runs a service and collects a fee. She relies on the trust of the websites to maintain her business. The short-term gain of accepting bribery for issuing non-validated DAA certificates could, however, outweigh the loss of this trust and the potential failing of her business. First, the protocol only allows forging identities in collusion with a verifier. Second, it is possible to keep track of the number of identities in the system that each verifier attested to by modifying the protocol (see Section 3.4.5). If a verifier confirms an unusually large number of identities or it is inconsistent with public information (e.g. subscriber lists; the circulation of newspapers is independently audited in most countries, as it is used to set advertising rates), this will raise doubts. A fake identity can thus be linked to the verifier that colluded in creating it. Hence, both the issuer and the verifier carry the risk of exposure, which grows with the number of fake identities they produce.

Verifiers: The verifier's incentive to participate is that it is either being paid (IVS), run by one of the participating websites (subscriber list) or is a readily available governmental service (smart passports/identity cards). For an IVS, as well as the government issuing smart identification documents, trustworthiness is existential. By adding a pseudo-probabilistic procedure to the protocol (see Section 3.4.3), large-scale fraud can be detected with high probability, which would terminate the IVS's business. For newspapers, the participating websites need to carefully decide, which subscriber list they accept — the newspaper should have a reputation to lose. As discussed before, a dishonest issuer has to collude with a dishonest verifier. In this case, they can quietly skip the audit; however, the public can still determine the number of identities verified per verifier. This allows at least some amount of public scrutiny, as, e.g. the approximate size of the subscriber list is publicly known. One could make the auditing publicly verifiable. This, however, comes the expense of some users' privacy.

3.5 Evaluation

We evaluate TrollThrottle in terms of how easy it is to deploy, and how much performance overhead it incurs. To demonstrate the former, we retrofit it to an existing website, without any modification to the server-side code — in fact, without the website being aware of this. To demonstrate that it incurs only modest costs, we simulate realistic traffic patterns using a recorded message stream and measure computational overhead and latency.

Deployability We demonstrate that the protocol can be deployed easily by retrofitting it, without any server-side changes, to Reddit.com, the most visited news website in the world [4] and an alleged target for large-scale astroturfing and propaganda efforts [159].

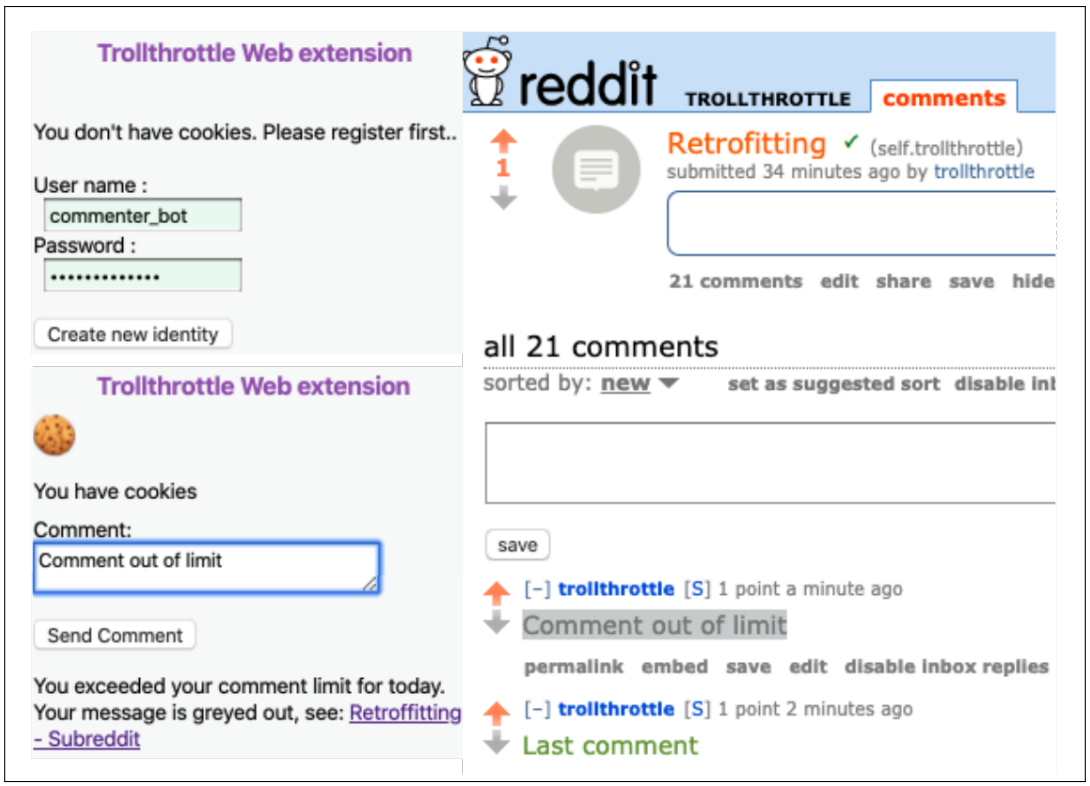

Figure 3.7: Screenshot of Reddit deployment, for identity creation and commenting scenarios, see Retrofitting subreddit

On Reddit, we created a forum as a testing ground. We implemented signature creation and verification in a JavaScript library and used a simple browser extension [164] to load this library when entering the forum. In an actual deployment, this library would be loaded via JavaScript inclusions. We point out, however, the known problem that there is no guarantee the website W is transmitting the correct script. This is a well-known issue for all web-based services that claim end-to-end security (e.g. ProtonMail [88]), and sometimes mitigated by offering optional plugins (e.g. `mega.co.nz`). Note, though, that there was an incident where the chrome extension itself was compromised via the Chrome web store, which highlights the need to trust both the software developer and the distribution mechanism [5].

We forked the Multiprecision Integer and Rational Arithmetic Cryptographic Library (MIRACL)[121] and modified it for portability. Then, using emscripten, we compiled this library to JavaScript. As this library is widely used to implement elliptic curve cryptography and large integers, this modified version could be of independent interest for front-end applications[118]. We use this library to implement Brickell and Li's pairing based DAA scheme using Barreto-Naehrig (BN) [12] elliptic curve at the 128-bit security level. Furthermore, we use Libsodium [101] for digital signatures used in the identity verification protocol (based on Ed25519 with 256 bit keys), hashing (BLAKE2B), authenticated public-key encryption (based on XSalsa20 stream cipher and X25519 key

exchange with 256 bit keys) and randomness generation. For key derivation, we used PBKDF2 in Crypto-js [47] with 100,100 iterations.[2] The simulation is available in [165].

Any comment posted in this subreddit is transmitted according to the protocol (see Fig. 3.2). As the server side is not validating the comments in this instance, this task is performed by the JS library as well. It communicates with a simple HTTP server implementing the public ledger. Comments that do not pass are greyed out by using a subreddit-specific stylesheet (see Fig. 3.7).

Performance To evaluate TrollThrottle's performance, we compiled three realistic datasets [21] to represent plausible scenarios. Our focus is on traditional news outlets that want to establish a close relation with their readership. We thus examine two scenarios in this domain, and a third, representing an extreme case: the entirety of Reddit, the largest website categorised as 'News' by Alexa [4].

Scenario I: nationwide news source Most news pages operate primarily on a national scale. Here, traffic patterns can have sharp peaks, e.g. is the vast majority of German speakers situated in the same time zone. Using Germany as an example, the most popular news page [3] reports a 'clear five-figure number' of comments per day [110], with other pages reporting between 12k and 80k [201]. As we have no access to comments *before* moderation, we simulate the traffic patterns. We do so by combining the traffic of the German-speaking `r/de` subreddit of sufficiently many days until we reach a volume of 168k comments.

Scenario II: international newspaper In mid-2017, just three months before they decided to stop accepting comments out of lack of resources, the New York Times reported 12K comments per day [102]. If we assume an exponential growth at a rate of about 140% per year[3], we estimate 398k incoming comments in January 2020. Again, we use Reddit data to retrieve realistic traffic patterns. In this case, we collected all comments on submitted links to `nytimes.com` from a 24h period. We aggregated the comments over a two-month period, from the beginning of May to the end of June 2019, to reach 268k comments.

Scenario III: Number of comments per day on Reddit From a 10-year dataset that includes all comments ever posted on Reddit, we pick the recent busiest day, which is 27 June 2019 with 4,913,934 comments. As for the other datasets, we did not filter out the comments that are marked as '[deleted]', i.e. were removed by moderators or their respective authors. They do not contain information about their authors, but still show the request patterns that the website needs to handle. Hence, we regarded them as one regular user.

[2] Default for server-side storage of passwords in LastPass [97].
[3] Extrapolated from data points in 2016 [200] and 2018 [201].

3.5.0.1 Performance measures

We focus on the performance requirements from the perspective of the news outlet that has to serve users within a given latency and compute the additional cost due to the new computations. To get a precise measure of the overhead incurred, our experiment only simulates the cryptographic operations and does not display the comments or use network communication. The computation is performed separately for the server and the client. We assume the issuer is trusted and thus disregard the extension in TrollThrottle Appendix Section 7.3.

As for the other datasets, we collected the comments annotated with their author's nickname and the time point they were posted. The dataset is thus a sequence of tuples (t, u, m) ordered by the time point t at which u posted comment m. We assume each nickname corresponds to a different actual person, thus over-approximating the effort for key generation.

(1) simulate the issuing protocol, if u comes out in the entire (10 years) dataset for the first time,

(2) simulate the commenting protocol to produce a signature for the comment, and finally

(3) simulate the server side signature verification.

Step (1) and (2) can be done in a pre-processing step, as they are computed by the user and issuer. We measure the time for commenting ($\delta^{\mathsf{Comment}}$) and issuing ($\delta_I^{\mathsf{Issue}}$ and $\delta_U^{\mathsf{Issue}}$, for the issuer and the user, respectively). For step (3), we simulate the load of the server side on a Ruby-on-Rails application with Nginx load balancer.

Firstly, we estimated the number of cores needed to satisfy a latency requirement of $l = 0.1$ seconds using a simple first-come-first-serve scheduler. To determine this value, we used the following algorithm. We sampled the server computation time δ_s by measuring the verification time for a random comment. We start with one core. We compute a first-come-first-serve scheduling until we reach a point where a comment posted at t is scheduled at t' such that $t' + t_s < t + l$. If we never reach such a point, we are done and output the number of cores. Otherwise, we add a new core.

Secondly, we simulated the load on the server. For each point (t, u, m) in the database, we simulate the arrival of the encrypted signature (γ, nym) resulting from pre-processing m, at time $t + \delta^{\mathsf{Comment}} + \delta_I^{\mathsf{Issue}} + \delta_U^{\mathsf{Issue}}$. We run Verify on the signature and measure the finishing time t_f, as well as the actual processing time δ^{Verify}. We report the results in seconds for the largest dataset in Table 3.3.

In Table 3.4, we report the number of cores needed and the cost incurred by the computations just described, i.e. the overhead compared to normal website operations. The number of cores to meet the latency requirement was estimated as described above and used in the simulation. To account for the cost, we employ the core hours metric, which is the product of the number of cores and the total running time on the server. We take Amazon on Demand EC2 pricing [8] as an example and assume \$0.05 per core hour. We also report on the maximal latency encountered in the simulation and the percentage of comments that met the target latency of $\leq 0.1s$. Finally, we report the

Table 3.3: Evaluation for Reddit use case (3 cores).

measure		mean	median	variance
issuing (on U)[1]	$\delta_U^{\mathsf{Issue}}$	0.038	0.036	0.069
issuing (on W)[1]	$\delta_I^{\mathsf{Issue}}$	0.010	0.009	0.0006
commenting[2]	$\delta^{\mathsf{Comment}}$	0.036	0.032	0.0003
verification	δ^{Verify}	0.021	0.018	0.0002
latency[3]	$t_f - t$	0.022	0.019	0.0002
commenting (on U)[4]	$\delta_U^{\mathsf{Comment}}$	0.058	0.057	0.01

(1) over all new users.(2) computation overhead w/ pre-computed signatures.(3) shows server-side total processing time.(4) on 1000 samples, single-threaded.

Table 3.4: Scenarios for performance evaluation, including the number of comments, source of the data stream, number of Intel E5 2.6 GHz cores, operating cost per day, maximum latency, percentage of queries answered within 0.1 secs, number of genesis tuples computed (i.e., number of distinct nicknames), and total ledger size.

scenario	#comments	#cores	daily cost	max. latency	latency < 0.1s	#genesis tuples	ledger size(mb)
Nationwide newspaper (r/de)	168k	1	$1.20	0.166s	99,99%	13,975	204
International news. (url:nytimes)	268k	1	$1.20	0.391s	99.99%	87,223	633
Reddit (r/all)	4.9M	3	$3.60	1.011s	99.99%	1,217,761	10628

number of genesis tuples created in the ledger, i.e. the number of nicknames in the dataset, and the total size of the ledger, representing an over-approximation of the storage requirements of a single day of operation.

Since comments are hashed before signing, the communication overhead is approximately 2.4 KB, independent of the comment size. To evaluate the storage requirements on a consensus-based public ledger, we chose Tendermint [78] as an example. Tendermint employs a modified AVL tree to store key-value pairs. Values are kept in leaf nodes and keys in non-leaf nodes. The overhead is about 100 bytes per non-leaf node [156]. For the largest dataset, each participant in Tendermint would thus require approximately 12 GB of space. Once the current commenting period is over, the signed comments and hence most of the data can be purged. To allow accountability for censorship over the last month, the data of the last thirty commenting periods can be stored on less than 0.5 TB.

In summary, the additional cost on the websites is modest compared to the moderation effort saved.

3.6 Limitations

Despite the auditing by the issuer and the limited accountability for colluding issuer and verifiers in the extended protocol, we have centralised trusted authorities. One way to remove these is to introduce protocols that can recognise Sybils. This could relieve the issuer from the responsibility of auditing the verifiers and potentially allow for a protocol with accountability features to deter misbehaviour. As this topic is orthogonal to our protocol, we leave it for future work, but remark that, theoretically, Sybil-detection is possible without user identification. A potential approach is to combine biometric methods [149, 10] with captchas. Uzun and Chung proposed such a protocol to show liveness. Here, the user's response to a captcha involves physical actions (smiling, blinking) that she captures in a selfie video [168] within a 5s time limit. Their approach is based on the fact that automated captcha-solving takes considerable time, and face reenactment (e.g. [158]) is difficult to do at scale. Building on the same assumptions, a Sybil-detection scheme could be built by pseudo-randomly defining sets of users that need to show liveness *at the same time*.

TrollThrottle aims to provide a similar user experience to website logins. Hence, all client-side secrets are derived from the login and password of the user and thus vulnerable to password-guessing attacks. This can be mitigated by incorporating a two-factor authentication into the protocol, or by setting up the key generation to require a password of sufficient length and entropy, as to enforce the use of password managers.

Finally, the client-side code is loaded by the website, which could potentially include a different script albeit this behaviour would leave traces. As previously discussed (see Section 3.5), this is a well-known problem for web-based apps, and usually mitigated by offering optional plugins.

3.7 Related Work

The detection of astroturfing has been tackled using reputation systems (e.g. [124]), crowdsourcing (e.g. [171]) and feature-based analysis (e.g. n-gram detection [127], sentiment analysis [144], or by analysing responses [116]). Fundamentally, the posting profile of a politically motivated high-effort user is not very different from a state-sponsored propagandist[91], hence we focus on prevention instead of detection. The detection and prevention approaches could be combined, but detection approaches either come at a loss of accountability, or they need to explain their decisions, although many of them rely on the fact that the bot is not adapting to the mechanism (e.g. via adversarial machine learning [83]).

Our approach is similar to anonymity protocols in which we specify a way of exchanging messages without revealing identities. In contrast to anonymity protocols, TrollThrottle provides anonymity with respect to the ledger, but presumes the communication channels to provide sufficient anonymity. By itself, TrollThrottle is not resistant against traffic analysis — here anonymity protocols come into play. One might ask whether anonymity protocols already do what TrollThrottle proposes to do. To the best of our knowledge, Dissent [46] is the only anonymity protocol that provides explicit account-

ability guarantees, but these pertain to the type of communication, not to sending more messages than allowed. Furthermore, unlinkability is not achieved within the group, but towards outsiders. In each protocol phase, parties generate new secondary keys, which they broadcast signed and encrypted to all members of the group. This requires setting up a group in advance. This is unsuitable for our setting, where the group comprises all registered users of a web. TrollThrottle preserves unlinkability even within this group.

Pseudonymity systems like Nym [79] or Nymble [87] provide anonymous, yet authenticated access to services, but some allow resource owners to block access at their discretion. By using a ledger and a common set of rules, TrollThrottle users can claim and prove censorship, but have to trust the ledger. This is in contrast to p2p-protocols, where censors may be sidestepped, but cannot be forced to publish the content themselves. Dingledine et al. advocate for the transaction of reputation/credit between pseudonyms [50]. By contrast, the credit in our scheme is essentially the number of nyms. This simplifies the system and ensures unlinkability, at the cost of inherent limitations: the 'credit' is the same for every participant (τ for each commenting period) and cannot be transferred.

One of the main cryptographic components of TrollThrottle is a specific DAA scheme with additional properties (instant-linkability and updatability). DAA was introduced as a way to address privacy issues of the remote attestation protocol proposed for TPMs [73]. There exists a number of schemes, e.g. based on the RSA assumption [24], on elliptic curve cryptography [25, 41], on the LRSW assumption [26, 18] and on the q-SDH assumption [39, 27, 38]. We focused on the scheme by Brickell and Li [27], because it supports these properties, produces short signatures and because a reference implementation was available.

There are building blocks besides DAA that are compatible with TrollThrottle. Anonymous Credentials (AC) allow users to prove (a set of) attributes about themselves to third parties, usually via an interactive protocol (but there are non-interactive schemes). An efficient scheme, by Baldimtsi and Lysyanskaya[11], supports only single-time use of a credential, which would require to store a fresh credential for each comment that the user would like to post in the future (the shown attribute would also need to include the date and some unique value). Multi-show credentials, for instance the one by Camenisch et al. [32], would decrease the number of fresh credentials required, but would still depend on the user's obtain an attribute for every possible day/comment number combinations. What's more concerning is that the attribute value would have to be set by the issuer to a unique value (to prevent double spending/commenting), which would decrease the privacy of this approach and allow the Issuer to link certain comments. Therefore, it seems more efficient to use a system that supports domain-specific pseudonyms with a secret-key based attribute than lightweight credential systems. It worths noting that DAA can be viewed as such a credential system for just a single and secret attribute (the secret key).

The most similar credential system to the DAA scheme, that we used, was proposed by Camenisch et al. [33]. In this system, an issuer creates and distributes so-called dispensers. Dispensers are used to create a predefined number of one-time credentials valid for a given date. This system can be immediately used in TrollThrottle. As an

implementation was not available, we perform a qualitative analysis. On the one hand, verification is faster in their scheme, they perform seven multi-exponentiations in a prime order group and one in an RSA group, while Brickell and Li's scheme perform one multi-exponentiation in each group i.e. $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, and one pairing computation. On the other hand, the signatures, which consists of a unique serial number (similar to a pseudonym) and a number of proofs of consistency are at least twice as much larger and their size depends on how the proofs are implemented. This produces considerable computation and communication overhead in the ledger. Moreover, the verification of comments is performed by the websites, making verification efficiency less important than the size of the data included in the ledger. Therefore, the DAA scheme represents a preferable tradeoff.

3.8 Impact on Society

We provide a solution for newspapers that want to interact with their readership, but cannot bear the cost of moderation. As of now, among the Top 10 websites in the Alexa 'News' section that belongs to a newspaper, three do not offer on-site commenting, two others disable commenting functionality for controversial topics and four require a Facebook signup with a real name. The last one apparently has this functionality, but did not display any comments or provide a link to leave one, presumably due to a glitch. Hence any technique making this interaction feasible again is an improvement to the political discourse. We shall nevertheless discuss some implications in case TrollThrottle, or a similar system, should be adopted in larger parts of the web.

Setting the threshold

From a technical view point, setting the threshold is a matter of balancing the number of regular users that post beyond this threshold with some target cost that an astroturfing operation should incur. From this perspective, there should be a clear demarcation between bots and regular users, that is characterised among other features by the number of messages these users send. This is, however, not the case, as the journalist Michael Kreil argues in response to a scientific study that used text mining and other learning techniques to recognise social bots [77]. He contacted these purported bots and found out that many of them were, in fact, real people who post well over 150 politically divisive messages per day [91]. In our evaluation database, we found that around 2 % of users are above the threshold of 20 messages per day. Upon inspection, some of those can be categorised as bots, but many are just very active users or cannot be clearly distinguished from those. One particular user posts a daily average of 96 messages on a cricket-related forum. He or she is just a big sports fan. If, in addition to the daily limit, we impose a limit of 100 comments a week and 300 comments per month, then only an additional 443 Reddit users will be affected, compared to $206,855$ out of $6,619,612$ users affected by the daily limit itself (in June 2019).

This shows that the system can be set up to avoid affecting intensive users, but, ultimately, *there is no threshold that distinguishes trolls from intensive users.* During the world cup match between India and Afghanistan in June, e.g. the aforementioned

cricket fan posted 928 comments. The method we propose is thus affecting the political discourse. It discourages communication patterns employed by power users. This is not necessarily a bad thing, as collective belief formation is driven both by learning from the (stated) beliefs of others and by some interest in maintaining social acceptance [96]. Due to the difficulty of mapping virtual identities to real-world identities, one may argue that the discourse stands to benefit from a limit on the messages, which favours thought-out contributions. In summary, we propose a method for moderation instead of a clear-cut filtering mechanism. It can enable discussion where, currently, there is none. It can be adjusted to accommodate for fluctuations in use by evaluating current patterns, it will impose restrictions on a minority of users.

Centralisation of discourse

The public ledger provides a centralised view of the discourse on participating websites, even if its implementation is decentralised. This offers several potential advantages: with a slight modification of the protocol, the user can optionally add a pseudonym $nym_{rep} = \mathsf{NymGen}(sk_U, x)$ for some arbitrary x (e.g. by signing the comment again under an additional basename x), to make a set of messages — across websites — cryptographically linkable. Users can thus build a reputation across websites. Similarly, we may add yet another pseudonym $nym_{thr} = \mathsf{NymGen}(sk_U, tid)$, with tid some global identifier for discussion threads, to ensure that authors have only one identity per thread and don't respond to themselves with a different account. Websites could, theoretically, stop providing their own infrastructure for user registration, and only permit signed posts to appear, providing essentially projections of the ledger's representation of the public discourse.

While these features seem appealing, the idea of a centralised political discourse beyond news websites has to be seen critically. Most importantly, it may undermine the incentives of the issuer described in Section 3.4.7, so she might take the risk and collaborate with a malicious verifier. In our view, this scenario is unlikely. According to a 2018 study, about two thirds of U.S. adults obtain their news on social media sites, about 43% from Facebook[147]. The business model of most of these social media sites is based on exclusive access to their users' information; hence they have little interest to share it. Our focus is therefore on traditional news pages, who benefit from a healthy discourse.

3.9 Conclusion

The prevalence of social bots and other forms of astroturfing in the web poses a danger to the political discourse. As many newspapers are closing down their commenting functionality despite the availability of sophisticated detection methods, we argue that they should be combined with a more preventive approach.

We presented TrollThrottle, a protocol that raises the cost of astroturfing by limiting the influence of users that emit a large amount of communication, even if using different pseudonyms. TrollThrottle preserves anonymity, provides accountability against

censorship, it is easy to deploy and comes at a modest cost. We have also discussed its social impact in Section 3.8.

By how much do we raise the cost of astroturfing? We shall regard the last week before the 2016 US election for a rough calculation. The computational propaganda project considered around 3.4M election-related tweets to be originating from bots who emit more than 50 messages per day [80]. If we assume a threshold of 20 messages/day and perfect coordination between the bots, 24,178 identities need to be stolen to reach the same target. A lab study [15] finds that users are willing to sell their Facebook accounts for $26 on average, which is only slightly above the black-market value for stolen verified Facebook accounts. Such operation would thus face a cost of $634,501 and a risk of detection.

A remaining challenge for future work is to provide Sybil detection without identifying the users. The verification of user identities inherently relies on the party asserting them, but, theoretically, this step is not necessary to determine whether, at a given time, two virtual identities are controlled by independent parties.

4

Accountable Javascript Code Delivery

The Internet is a major distribution platform for web applications, but there are no effective transparency and audit mechanisms in place for the web. Due to the ephemeral nature of web applications, a client visiting a website has no guarantee that the code it receives today is the same as yesterday, or the same as other visitors receive. Despite advances in web security, it is thus challenging to audit web applications before they are rendered in the browser. A compromised or malicious web server can easily target classes of users by sending different codes to different users. The web server might insert malware based on browser fingerprints, an email provider might disable encryption on a specific IP range, a cryptocurrency wallet might redirect payments made in some countries or the server might use the user's browser for cryptojacking.

In this chapter, we present an opt-in Accountable JS protocol that can be used by websites to convince their users that they are trustworthy in an economical way. Using the Accountable JS protocol the users can verify that the active content on the web page they are visiting is the same for everyone and in the client side it adheres to the standards set by the Accountable JS protocol.

To present Accountable JS protocol, we begin with outlining our threat model and underlying assumptions. Next, we describe several types of web application use cases (including WhatsApp Web, AdSense and Nimiq) that can benefit from our protocol. Thereafter, we provide an overview of our approach to the problem and how we use cryptographic algorithms. Then, we revisit the use cases to illustrate how Accountable JS could be applied to real-world applications and discuss how it enhances the security and accountability in those use cases.

4.1 Problem Statement

Over the years, the web has transformed from an information system into a decentralised software distribution platform. Websites are programs that are freshly fetched whenever accessed and the web browsers are runtime environments. This design implies that when a user opens a website, they have no reason to trust it will run the same program that it did yesterday or the same program that other users receive. Instead, the application loaded may vary over time, and different users may receive different codes.

The majority of web pages, and even web applications, have neither specified security goals nor the need to establish them. Nevertheless, for some websites, maintaining trust between developers and users is part of the business model:

- a private email provider might wish to reassure users that it will always encrypt their messages,
- a cryptocurrency wallet might wish to guarantee that it has no access to users' funds, or
- a tracking pixel might wish to prove that it only receives data that is explicitly sent to it.

Some academic proposals for secure protocols implemented for browsers include TrollThrottle [59] and JavaScript Zero [143];

industry proposals include payment platforms such as Stripe [154] and Square [152], chat protocols such as WhatsApp Web [175], Facebook Messenger [62] and Matrix's Hydrogen client [107], encrypted cloud storage such as MEGA [111] or SpiderOak [151].

A concrete example is Nimiq [123], an entirely web-based digital currency managing private keys in the browser. It is challenging for such websites to make verifiable guarantees to their users: a compromised or malicious web server can precisely target classes of users: the email provider might disable encryption on a specific IP range, the cryptocurrency wallet might redirect payments made in some countries, or the tracking pixel might exfiltrate data only for certain users.

Auditing A common risk mitigation strategy is *auditing*: a developer who wishes to build trust appoints external auditors to inspect the client code. This can include both vulnerability research (e.g. via bug bounties) or commissioned security audits. Audits work well where it is possible for a user to verify that the code they are running is the same code that was audited, for example when binaries are received via third party package repositories or app stores that control the distribution and targeting. App stores do not usually permit developers to deliver different codes to different users for the same app, except in a restricted set of circumstances such as for beta testing new features.

However, auditing does not work for web applications: a compromised or malicious web server can simply choose at load time to deliver unaudited code to a user. No matter how careful the audit or even verification of the web application, users cannot know that they are receiving the audited code. Large parts of modern web security thus depend on techniques like sandboxing or access control to critical resources like cameras, but fail to capture properties defined in the context of the application (e.g. authorisation of transactions in a payment system).

Accountability A second risk mitigation strategy is *accountability*, where developers can be held accountable for applications which they publish. In curated software repositories such as Debian GNU/Linux or the Apple App Store, developers' code is reviewed and malicious or compromised code is linked to their identities. Developers who repeatedly publish malicious code may face consequences such as loss of user trust or banning from the repositories. For example, a package mirror which publishes malicious code may be removed from future lists of mirrors, or a developer who takes over a browser extension and publishes a malicious version [42] may be blocked from publishing future code updates.

Again, web applications fail to have accountability. A malicious or compromised web server may publish malicious code to certain users, but there is no public record of the code which it serves, and thus no way for users to hold the server accountable.

Summarising, it is difficult to establish trust in the web as a software distribution mechanism because it lacks *auditability* (the means for anyone to inspect the code being distributed to others) and *accountability* (the means to hold a developer accountable for the code they publish).

4.2 Overview

In this chapter, we propose an opt-in transparency protocol that aims to establish more rigorous trust relations between browsers and web applications, and provide the foundation for a more secure web. Using our standard for accountable delivery of active content, efficient and easy-to-use code-signing technique, and public transparency logs; websites can convince the users that they are trustworthy in an economical way. At a high level, we propose that web application developers, who choose to opt in, provide a signed manifest enumerating all the active content in their applications. The manifest files in our proposal are stored in publicly readable transparency logs. When a browser requests a URL and downloads the resulting HTML document from the web server, the web server also provides the corresponding manifest for this URL. The browser checks that the active content provided by the server matches the manifest entry, that the manifest is correctly signed, and it is consistent with the transparency logs.

Moreover, our proposal aims to reinforce the communication between the browser and the web server by adding non-repudiation to the HTTP request-response procedure. By itself, Transport Layer Security (TLS) does not provide evidence that what was delivered actually originated from the web server. Using digital signatures, we show how HTTP requests can be extended to provide a proof of origin.

From the signed manifest, the transparency logs, and the non-repudiation mechanism, the protocol establishes that:

- The code a user executes is the same for the users of the plugin within a certain timeframe depending on the validity of the manifest and a new manifest is signed.
- On the client side, the code is bound to interact with third party code according to how the developer declared in the manifest. This includes the order of execution, the trust relation to third party code, and the use of sandboxing.
- If the code's execution is inconsistent with the manifest, the browser can provide a claim that can be verified by the public.

Our proposal can be implemented by changes in the server configuration only, without the need to modify the served web content (assuming that the web page already makes use of Subresource Integrity hashes) and without changes to the HTML standard.

To sum up, our contributions are as follows:

1. We propose Accountable JS, a protocol to enable auditability and accountability for web apps.
2. We formally model Accountable JS with the Tamarin Prover and prove desired properties in the presence of active adversary.
3. We implement Accountable JS in a browser extension that obtains the signed manifest, verifies its signature, and both statically and dynamically ensures that the active content on a web page agrees with the manifest. We also provide a code-signing mechanism for the developers.

4. We evaluate the deployment of this technology and the performance overhead for the client in six case studies, including real-world applications: Google AdSense, Nimiq and WhatsApp.
5. We model Meta's Code Verify protocol and compare its properties with Accountable JS.

Relationship to Meta's Code Verify protocol In [76], Meta (formerly Facebook) proposed Code Verify, likewise implementing a mechanism to enforce accountability via transparency for active content in the web. Our present proposal goes beyond Code Verify and provides a superset of its functionality, most notably the ability to delegate trust to third parties. On the other hand, our browser extension is an academic prototype and thus not ready for productive use. The protocol has the same message flow, but choses a different signature scheme and encodings. We elaborate on these differences in Section 4.11.1.

An initial draft of the present proposal was shared with Meta's WhatsApp team in 2022. The protocol, manifest file format and browser extension we present in this work are academic developments by the authors and not endorsed by Meta in any way.

4.3 Background

Web pages are delivered via HTTP or HTTPS. In the latter case, a secure and authenticated TLS channel tunnels the HTTP protocol. Typically, the initiator of the TLS connection, i.e. the web browser, is not authenticated[1], whereas the responder, i.e. the web server, is identified with their public key and a certificate linking the public key to the domain.

The authentication guarantees of TLS do not include non-repudiation of origin, i.e. a communication party cannot prove to a third party that they received a certain message. This property is an important building block for accountability and can be achieved, e.g. using digital signatures. After the shared keys are established in TLS, any messages exchanged could be produced by either party. Roughly speaking, the party providing the evidence has enough information to forge it. Ritzdorf et al. [136] proposed a TLS extension that provides non-repudiation, but it has not been deployed in the wild.

Browsers typically parse the HTML document describing the web page into a tree of HTML elements called Document Object Model (DOM) [37]. Some HTML elements have *active content*, which includes Flash or Silverlight, but we will focus on JavaScript (JS) in this work. Active content can be *inline*, i.e. hard-coded in `<script>`-tags or event handlers, *external*, i.e. referring to an external JS file by URL, or *via iframe*, i.e. the web page contains an iframe that refers to an HTML file which, again, contains active content. The browser may include multiple windows with multiple tabs, displaying websites in parallel. For our purposes, we can abstract the browser to a single top-level window that represents the client side in the HTTP protocol. App stores provide a unified distribution system for applications. Typically, they are curated by their owners:

[1] At the communication layer. Authentication may be implemented at the application layer.

developers submit their software, the app store owner inspects them for compliance with their guidelines (which can include quality control, but also censorship) and distributes them to their users. While it has been proposed to use app stores to deliver diversified versions of software [65] and both Google and Apple's App Store support A/B testing, users expect to not receive targeted applications.

Like in the case of app stores, we distinguish between the roles of the *website*, which is distributing the web application, and the *developer*, which is the author of the web application. This allows us to view the website as a distribution mechanism that is necessarily online and publicly visible, as opposed to the developer, who can be offline most of the time. We distinguish the following roles:

- The web application developer (short: *developer*) creates the active content and has a secure connection to the web server. It is not active all the time.
- The web server (short: *server*) delivers code provided by the developer to the *client*. The website and the developer are associated with a domain, but the client is anonymous.
- The web browser (short: *client*) requests a URL from the website.

A transparency log (short: *ledger*) provides a publicly accessible database. It typically has the property of being append-only (for consistency), auditable, verifiable, and it hinders equivocation. Hence, for the data in the logs, all parties are convinced that it is a public record and that everyone sees the same version of it. We are using the ledger to store manifest files for each URL. Having public records of the manifest files allows us to reason about accountability.

4.3.1 Threat Model

Dolev-Yao attacker We consider a Dolev-Yao style adversary, i.e. cryptography is assumed perfect (i.e. cryptographic operations do not leak any information unless their secret keys are exposed), but the attacker has full control over the network. This is formalised in our SAPiC [92] model in Section 8.1. Informally, we assume hash function to behave like random oracles, signature schemes to be unforgeable and TLS to implement an authentic and confidential communication channel. We also rely on an intact public-key infrastructure.

Corruption scenarios We assume honest parties follow the protocol specification and dishonest parties are controlled by the attacker. The parties which considered to be honest are determined by the property of interest:

- *Accountability and Authentication of Origin:* An honest client wants to be sure that code is executed only if it was made public and transparent i.e. inserted into logs by the developer; here developer and web server are assumed dishonest.
- *Non-repudiation of Reception* A dishonest client may want to present false evidence for having received some JS code. Here we assume the public to be trusted and run a specified procedure[2] to check the evidence, and the web server to behave honestly,

[2]Detailed in Accountable JavaScript Appendix Section 8.2.

i.e. not to help the client provide false claims of reception, which are against the web server's interest.

- *Accountability of Latest Version* An honest client that receives a version of the code and wants to ensure it is the most recent version. We assume an honest global clock that helps comparing the time of the code reception and the latest version at that time, and consider a dishonest developer and web server.

Target websites We target developers that *aim at establishing user trust* or pretend to do so. Hence we assume, for honest developers, that active content changes infrequently, e.g. multiple times per day, and that their code is designed to facilitate the audit. Dishonest developers may counteract, but, due to accountability and authentication of origin, there is public record of that.

Therefore, while our formal security arguments make no assumption on how often the code changes are or how obfuscated it is, we assume that, from accountability of authentication of origin, code obfuscation attacks or microtargeting are practically disincentivised.

Browser features & Transparency log We assume existing browser security features, in particular the `sandbox` attribute of the iframe tag, to be implemented correctly. Furthermore, the transparency log is trusted, efficient, available, append-only and provides non-equivocation (i.e. the same information is served to everyone).

There are many strategies to implement such a log. For example, Trillian [161] and CONIKS [114] use data structures that can be distributed over multiple parties and allow to efficiently prove append operations. Misbehaviour can thus be detected by trusted public auditors or by honest logs distributing such proofs (called *gossiping*). See [113] for a survey over different mechanisms.

4.4 Use Cases

We introduce several types of web applications that will benefit from our protocol. We will revisit these examples later and show how our approach is applicable to them.

4.4.1 Self-Contained Application

Perhaps the simplest possible web application is a one-page HTML document with active content that simply prints 'Hello World' into the developer console. Upon loading this website, a user can manually check that its sole behaviour was to print 'Hello World', but they have no guarantees about subsequent page loads: a server could easily decide to provide different behaviour to certain users, or to insert malware based on IP address or browser fingerprint. For this simple example, the consequences of a malicious or compromised server are relatively limited, although we remark that cryptojacking[3] is a growing trend [34]. We remark that, by default, every user who loads this web

[3]Malicious JS which secretly mines cryptocurrencies in unsuspecting users' browsers.

application receives the same source code. However, there is no easy way for users to verify this fact.

More complicated web applications may have login functionality, or asynchronous client-server communication, or other advanced features. In order to personalise users' experiences, applications may dynamically fetch data using technologies such as Relay [4] or Apollo [5]. However, it is often still the case that all users receive the same JS source code bundle.

WhatsApp Web is a large real-world self-contained web application: its source code is bundled using WebPack and served to all users; personalisation is implemented through local storage and dynamic data fetching. We will show how our protocol can be applied.

4.4.2 Trusted Third-Party Code

Many websites rely not just on their own content but on resources served by a third party. This may be a Content Delivery Network (CDN) serving common JS libraries, embedded content such as photos or videos, analytics and measurement libraries, tracking pixels, fraud detection libraries, or many other options. For example, the following code loads the jQuery JS library from a CDN, and uses it to display a 'Hello World' message.

```
<html>
  <head>
    <script src="https://googleapis../jquery-3.6.1.min.js"
    integrity="sha384-i6..."/>
  </head>
  <body>
    <script>$("body").html("Hello␣World")</script>
  </body>
</html>
```

Listing 4.1: Trusted third party code

As before, users are supposed to always receive the same code from the server. This time, there is an additional avenue for compromise, though: even if the first-party server is honest, it is possible for the CDN to perform targeted attacks. The developer, however, wants to pin the third party code to the precise version that they inspected or trust.

4.4.3 Delegate Trust to Third Parties

The application uses third party code that its developer cannot vouch for. This can be the case if the code is too complex to inspect or if the application developer wants to always use the latest version. The third-party developer, however, is willing to vouch for their code. An example of this is Nimiq's Wallet, a web application for easy payment with Nimiq's crypto currency. This application can be embedded by first-party applications that provide, e.g. a web shop, who are willing to trust Nimiq, but only given that they make themselves accountable for the code they deliver.

[4] https://relay.dev/
[5] https://www.apollographql.com/

```
<html>
  <body>
    <script type="text/javascript">
      function addTransaction () {
        window.postMessage({'id': '123', 'amount': '10n', '↷
        from':'abc'}, 'https://wallet.nimiq.com/');
      }
    </script>
    <iframe src="https://wallet.nimiq.com/" onload="↷
    addTransaction()">
    </iframe>
  </body>
</html>
```

Listing 4.2: Delegate trust to third party

4.4.4 Untrusted Third-Party Code

For web technologies, consecutive deployability is a must. Hence, in this use case, the application developer cannot audit the code, but the third party does not use Accountable JS. The application developer needs to blindly trust the third party, but using sandboxing techniques, it can restrict the access that the possibly malicious script provided by the third party can have.

A particularly important instance of this problem is ad bidding. The third party is an ad provider that decides online which ad is actually served. Because they cannot review the ads that they distribute, which may contain active content, they are not willing to vouch for the code they distribute. This is the case for Google AdSense, used by over 38.3 million websites. Cases where ads were misused to distribute malicious code are well documented [2].

4.4.5 Code Compartmentalisation

The application that the developer provides can be compartmentalised so that the most sensitive information is guarded by a component that is easy to review and changes rarely. The other components that are user-facing and changing more often are separated from this component using sandboxing. The developer wishes to reflect this structure and make themselves accountable for the whole code, but also separately commit on keeping the secure core component small and auditable. For example, Nimiq's Wallet components follow a similar structure.

4.5 Approach: Accountable JS

We propose a cryptographic protocol between the client, the server, the developer, and a distributed network of public transparency logs. The protocol's objective is to hold the developer accountable for the code executed by the browser. The protocol provides four main functionalities:

- The server provides a manifest declaring the active content and trust relationships of the web application, which the client compares with a published version on the transparency logs.
- The client measures and compares the active content received by collecting active elements, e.g. JS, in the HTML document delivered by the web server.
- Developers and clients submit manifests to a public append-only log to verify that everybody receives the same active content.
- The server signs a nonce as non-repudiable proof of origin for the JS that the client receives.

Website Manifests Website developers may provide a signed manifest for each publicly accessible URL in their websites (excluding the query string). The signed manifest comprises a manifest and a signature block over this manifest. A manifest describes the webpage, including, besides the active content, its URL and a version number. The active content is described in a custom format. We elaborate on the manifest directives in the supplementary material [58]. The developer's identity is distinct from the server's, but their certificates must share the same Common Name(CN) in order to restrain from unauthorised manifest deployments. The browser validates the authenticity of the developer's public key in the same way, using the existing Public Key Infrastucture (PKI) and its built-in root Certificate Authority (CA) certificates.

Accountable JS is an opt-in mechanism. The website declares the signed manifest using an experimental HTTP response header field called `x-acc-js-link`. Henceforth, the client, however, expects the website to always provide a valid manifest for this URL.

Client Measurement The client measures the active content inside the HTML document delivered in the response body, collecting information about each active element in the document and validating it with the corresponding manifest block in a manifest file. Elements that cannot be matched trigger an error and the user is warned about this error. The current extension is not preventive, but in the future with pervasive developer support, browsers may be designed to choose to halt the execution if delivered code is inconsistent with the boundaries drawn by manifest. The active content is measured with a so-called *mutation observer*, starting with the first request. The measurement procedure that we developed listens to the observer's collected mutations that regard active elements in a list. In Section 4.8, we explain the process in more detail.

Manifest Logs While a signed manifest may prove the integrity and authenticity of the manifest, it cannot prevent equivocation, i.e. it cannot prove the same signed manifest is delivered to every request by the web server. To this end, we propose to use transparency logs. Manifest files declare version numbers and there can be only one manifest file per version number. The developer publishes their signed manifest in a publicly accessible, auditable, append-only log very similar to the Certificate Transparency (CT) protocol [134], which provides logs for TLS certificates. Clients may

verify that a version they receive is the most recent online, or use a mechanism similar to Online Certificate Status Protocol (OCSP)-Stapling [128] to check that a version they receive was the most recent version a short time ago. Any client that encounters a signed manifest that is not yet in the log can submit it to the log. We discuss the transparency log considerations in more detail in Section 4.12.

Non-Repudiation of Origin We propose a non-repudiation mechanism for the client's web requests done by the client. In case a developer distributes damaging active content, an individual client cannot prove that they have received that content from a web server. While TLS provides integrity of communication via Message Authentication Codes and authenticity of the communication partner via its handshake, the client is nevertheless unable to prove that they received damaging content as both communication partners are able to forge the message transcripts after the key exchange.

We propose a simple mechanism whereby the web server signs a nonce chosen by the client, along with the signed manifest. The client transmits this nonce via a request header. We elaborate on the non-repudiable web request protocol in Section 4.10.

4.6 Manifest File

In the manifest, the developer declares the active elements a web application is bound to execute during its run time. The run time starts from the web request and ends with the window's close or a new web request. For Single Page Applications (SPA) (e.g. Nimiq), the run time for the web page ends when page is refreshed, its URL is changed or the window is closed.

The manifest file represents the active elements and their relevant metadata as a collection of attribute-value pairs in the JSON format. The metadata expresses the trust relations w.r.t. third party content and settings for sandboxing. The top-level properties in the manifest, also called manifest header, contain descriptive information about the web page: its URL, its version number, and optional metadata, e.g. the developer's email address. The domain within the URL determines which keys can be used to sign the manifest, namely, the common name of the signature key's certificate has to match that domain.[6] The developer can decide for any numbering scheme for the version, but they must be strictly increasing with each new manifest published.

A manifest file is accepted if it is *syntactically correct*, i.e. follows the schema (see manifest manual in the supplementary material [58] for details), *complete*, i.e. it contains enough information about the web application and its active elements to enable evaluation, and, most importantly, *consistent with the delivered resource*, i.e. that evaluation succeeds.

4.6.1 Execution Order

An active content is considered dynamic if it is added after the window's load event; otherwise, it is static. The manifest specifies elements as either static or dynamic

[6] The query component of the URL [17] can be excluded, since the browser extension discards that part in the measurement.

using the *dynamic* attribute. SPAs in particular download or preload resources during navigation, rewriting the DOM on the fly depending on how the user navigates.

For static elements, the sequence number *seq* specifies in which order they must appear after browser renders the delivered HTML. It starts from 0 and repetitions are not allowed. Dynamic content is only measured if they are present in the web page, i.e. it is allowed to be injected, but not required to. This mechanism can also be used to declare region-specific active content. The order is ignored for dynamic content.

The measurement procedure will check if the list of the elements in the manifest is in the same order except for elements that will be dynamically added to the DOM. Elements may be removed dynamically, but only if the attribute *persistent* is set to false.

A JS element can be loaded synchronously (*sync*), asynchronously (*async*) or it can be deferred until the HTML parsing is done (*defer*). A synchronous JS element blocks the HTML parsing process and is executed in-order. Asynchronous and deferred elements do not block parsing. Asynchronous elements are loaded in parallel with other HTML elements, while deferred elements are loaded after parsing has finished. Hence, for both, the position on the DOM tree may not be predicted precisely.

4.6.2 Trust and Delegation

With the manifest, the developer provides assurance for the active content in their application. Third-party components, e.g. JS libraries, bootstrappers, advertisements or ad-analytics tools play a significant role in most modern web applications, which are thus a mixture of first-party code and code from multiple third parties. In the manifest, we enable the developers to decide the trust level on each active element imported to their web applications. For instance, they can take the responsibility and provide assurance (i.e. with a cryptographic hash) on first party elements while for the external elements, they may declare a valid source and delegate the trust on the developers of those resources.

We thus require each block in the manifest to have a trust declaration. There are three options to declare the trust level:

- *assert* : The developer provides the hash of the expected active content and asserts it is behaving as intended. It is computed using the standard Subresource Integrity (*SRI*) hash generation method [167], i.e. comprises the hash algorithm used, followed by a dash and the base64-encoded hash value.
- *delegate*: The developer refers the trust to the third party providing this element. Now the third party is taking responsibility for this code and provides a manifest whose location is either declared in the first-party manifest, or delivered in the headers of the third party's response. The third party manifest can likewise delegate trust, thereby constructing a chain of trust delegations.
- *blind − trust*: The developer blindly trusts the third party, without identifying the code they trust. This should only be used in combination with the *sandbox* attribute.

Table 4.1: Trust Relationships by Type of Active Element

	trust			
type	*assert*	*blind − trust*	*delegate*	*sandbox*
inline	●	○	○	○
event_handler	●	○	○	○
external	●	●	●	○
iframe with ...				
src_type =*external*	●	●	●	●
src_type =*srcdoc*	●	○	○	●
src_type =*script*	●	○	○	●

4.6.3 Types of Active Elements

The developer describes the manifest blocks for each active element by their resource type *type* (e.g. *javascript*, *iframe*), trust policy *trust* (e.g. *assert*, *delegate*, *blind − trust*), whether they are dynamic or static and, in case they are static, their sequence number *seq*. There are mandatory and optional directives for writing a manifest and these directives may depend on the resource type. If the developer declared a manifest section including an optional directive, that does not mean this directive is ignored in the evaluation; this directive still is part of the evaluation. For instance, the *crossorigin* directive is optional for *external* resource type, but if the developer declares a *crossorigin* attribute, then it has to match with the active content information. Not all resource types support all trust policies (see Table 4.1). We will discuss them one by one:

- *inline*: Inline scripts are `script` elements without the *src* attribute, i.e. the JS code is included in the HTML document. Therefore, *trust* can only be *assert* and may be omitted. The cryptographic hash covers the included JS code, i.e. the textContent value of the script element.
- *event_handler*: Event handlers are active content included in attributes such as `onClick` that are executed on HTML events. Like inline scripts, *trust* must be *assert* and can be omitted. Unlike *inline*, however, the (Subresource Integrity (SRI)-encoded) hash value covers the entire element, including the HTML tag itself.
- *external* : A `script` element can be outsourced by specifying its URL in the *src* attribute. An *external* script can originate from a different origin (cross-origin) or from the same origin. Trust can be set to *assert* and *delegate* – as sandboxing is not supported for external scripts, *blind − trust* would give little assurance.
- *iframe* : An *iframe* embeds another document within the current document. There are three different ways this can happen, which the manifest file represents using the attribute *src_type*. The most common is to specify a URL (*src_type* = *external*). This type of content can be declared with any *trust* value. For *trust* = *assert*, it is possible to either hash the whole embedded HTML document, or to provide a list of manifest blocks for the active elements inside the embedded document.

  Second, an iframe's content can also be hardcoded in the outside document (*src_type* =

srcdoc), in which case *trust* can only be set to *assert* (and the hash is computed on the *srcdoc* attribute of the *iframe* element).

Third, the embedded document can be created using the *document_write* method [51], via JS code inlined in the iframe's *src* attribute (*src_type* = *script*). Since this JS code is known at this point, *trust* must be set to *assert* and, again, either the JS code's hash be provided, or a list of manifest blocks.

4.6.4 Sandboxing

In addition, iframes permit the use of sandboxing via the attribute with the same name [55]. A sandboxed iframe is considered a cross-origin resource, even if its URL points to the same-origin website. Hence, because of the browser's same-origin-policy [13], the parent window and the iframe are isolated, and they cannot access the DOM of each other. Furthermore, sandboxing blocks the execution of JS and the submission of forms and more. These restrictions can, however, be lifted using an allow list in the HTML tag.

As we will see in the next section, security-critical websites need to use sandboxing to protect data from other browsing contexts; hence we reflect the *sandbox* feature in the manifest file. The measurement procedure ensures that the active element has an equally strict or stricter sandboxing policy than described in the manifest. An allow list is stricter if it is a subset of the other.

4.7 Use Cases, Revisited

We come back to the use cases from Section 4.4 to illustrate how Accountable JS applies to real-world web applications with different trust assumptions.

4.7.1 'Hello World' Application

We begin with the basic 'Hello World' website example, and add a reference to the manifest in its meta tags.

```
<html>
  <head>
    <meta charset="utf-8" name="x-acc-js-link" content="http↷
    ://www.helloworld.com/manifest.sxg">
  </head>
  <body>
    <script>console.log("Hello␣World")</script>
  </body>
</html>
```

Listing 4.3: First example: Hello World.

Alternatively, the manifest can be provided as an HTTP response header. The manifest file provides the URL and version of the website and simply lists the base64-encoded SHA-256 hash of the inline script.

```
{
  "url": "http://www.helloworld.com/",
  "manifest_version": "v0",
  "contents": [
    {
      "seq": 0,
      "type": "inline",
      "load": "sync",
      "trust": "assert",
      "hash": "sha256-AfuyZ600rk..."
    }
  ]
}
```

Listing 4.4: Manifest for first example.

4.7.2 Self-Contained Web Applications

Web applications can be completely self-contained. This may be for security or because they follow the recent serverless computing paradigm (e.g. Amazon Lambda). In serverless computing, a web application developer may only write static user-side code and delegate all the server-side logic to a cloud service provider.

The application of Accountable JS is straightforward in this case: as part of our prototype, we developed our deployment tool `generate_manifest`, which computes the hash values of all active contents in the browser and produces a manifest file that asserts their trustworthiness. The developer can then sign this manifest file.

We tested this methodology on a popular example, the WhatsApp Web client, and provide the manifest file in the supplementary material [58]. It lists nine external and four inline scripts.

4.7.3 Trusted Third-Party Code

The developer can use the manifest file to identify the included third party code by hash and set the order of execution. This expresses that the developer vouches for the third party code. We add the following attribute to the header of 'Hello World' example from Section 4.4.2 and we declare it in the manifest file with $trust = assert$.

```
<script src="https://googleapis../jquery-3.6.1.min.js" ↷
integrity="sha384-i6...">
</script>
```

4.7.4 Delegate Trust to Third Parties

The first party can delegate trust to a third party by embedding their code in an iframe (or linking their JS) and setting *trust* to *delegate*. The extension will verify the third party code based on a manifest file signed by its developer. This expresses that the

main developer vouches for the third party to be trustworthy, but demands that the third party itself can be held to account. This is in contrast to trusting a concrete piece of code provided by the third party.

We tested this technique using Nimiq's Wallet, which can be embedded in third-party web pages. These can now combine the code that they control (e.g. for setting up a shopping cart) with the code that Nimiq provides for signing transactions.

The website's manifest below (Listing 4.5) specifies some inline scripts with *trust* = *assert* (omitted) and an iframe with *trust* = *delegate*. The browser now expects the response to the query for the iframe's content (`https://wallet.nimiq.com`) to point to a URL with a signed manifest.

```
{
  "url": "https://www.example-shop.com/",
  "manifest_version": "v2",
  "contents": [
    [inline script manifests omitted]
    {
      "seq": 2,
      "type": "iframe",
      "src_type": "link",
      "src": "https://wallet.nimiq.com/",
      "sandbox": "allow-scripts",
      "dynamic": false,
      "trust": "delegate"
    }
  ]
}
```

Listing 4.5: Manifest is delegated to a trusted third party

4.7.5 Untrusted Third-Party Code

High-security applications may want to rely on third-party code they cannot vouch for, e.g. when including ads that are dynamically chosen by an ad-bidding process. We developed a small web application that uses Google AdSense and sandboxed this code, but noticed that AdSense and many other ad providers require access to the top-level window [85] for fraud detection, e.g. to detect invalid clicks [119].

We therefore needed to turn the relationship between the secure code and the untrusted code around. We sandboxed the secure code with *trust* set to *assert*, protecting it from the potentially unsecure AdSense code, which is not sandboxed and declared *blind* – *trust*. Now the AdSense code cannot access the secure document in the iframe. The manifest file is shown in List. 4.6. It includes thirteen active elements (six *external*, seven *iframe*) related to AdSense, along with Nimiq's Wallet (seq='6'), for which trust is delegated.

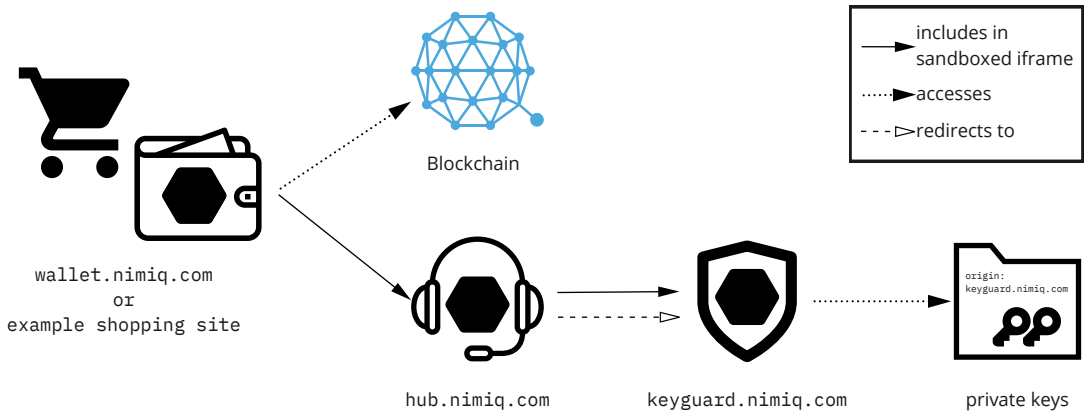

Figure 4.1: Structure of Nimiq Ecosystem.

```
{
  "url": "https://www.helloworld.com/",
  "manifest_version": "v3",
  "contents": [
    [six external scripts for AdSense with trust=blindtrust]
    {
      "seq": 6,
      "type": "iframe",
      "src_type": "link",
      "src": "https://wallet.nimiq.com/",
      "sandbox": "allow-same-origin allow-scripts",
      "dynamic": false,
      "trust": "delegate" // See Listing 4.7
    },
    [six more iframes for AdSense with blindtrust]
  ]
}
```

Listing 4.6: Untrusted AdSense and the Delegated Nimiq wallet at manifest section sequence number '6'.

Note that we relax the sandbox attribute of the secure iframe to allow script execution and to gain access to its own origin (`https://wallet.nimiq.com`) in order to access its cookies. Because of the trust delegation, we create a new signed manifest, which we expand upon in the next section (see Listing 4.7). The communication between the parts of the web application that handle user interaction and the trusted code in the sandbox takes place via postMessage calls.

4.7.6 Compartmentalisation of Code and Development process

We further expand on Nimiq's Wallet application, this time as an example for compartmentalising the code and the signing process. Nimiq's Wallet application at no point has direct access to the users' private keys. It is treated the same way as any other

third party application interacting with the Nimiq ecosystem (see Fig 4.1). It embeds the *Hub* which acts as an interface to the user's addresses and can trigger actions on the private keys. Access to the users' private keys is only possible through the Hub and pre-specified APIs. The Hub will then forward any request that needs to access the private keys to the *KeyGuard* component, which upon user input can decrypt the locally stored keys, perform the requested action, and return the result to the Hub.

The procedure `generate_manifest` produces the following manifest for Nimiq's Wallet. Observe that it heavily employs sandboxing. Both included iframes have the *sandbox* attribute set empty, meaning that there are no exceptions.

```
{
  "url": "https://wallet.nimiq.com/",
  "manifest_version": "v0",
  "contents": [
    [five external scripts]
    {
      "seq": 3,
      "type": "iframe",
      "src_type": "link",
      "src": "https://hub.nimiq.com/iframe.html",
      "sandbox": "",
      "dynamic": true,
      "trust": "assert",
      "manifest": [
        [seven external scripts],
        {
          "seq": 7,
          "type": "iframe",
          "src_type": "link",
          "src": "https://keyguard.nimiq.com/",
          "sandbox": "",
          "dynamic": true,
          "trust": "delegate"
        }
      ]
    }
  ]
}
```

Listing 4.7: Delegated content Nimiq Wallet's manifest.

The Wallet's manifest includes `hub.nimiq.com` in an iframe, containing, among other elements, the KeyGuard, which has a separate origin and thus exclusive access to the user's keys. For transactions, the Hub redirects to the KeyGuard. The KeyGuard is trusted, easy to audit, does not depend on any third party code and changes rarely. The KeyGuard manifest is as follows.

```
{
  "url": "https://keyguard.nimiq.com/",
```

```
    "manifest_version": "v0",
    "contents": [
      { "seq": 0,
        "type": "external",
        "link": "https://keyguard../web-offline.js",
        "hash": "sha256-L8NMxOGkIW...",
        "load": "defer",
        "dynamic": false,
        "trust": "assert"
      },
      [two external scripts w/ same dynamic/trust.]
    ]
  }
```

Listing 4.8: Nimiq Keyguard depends on its own content.

The Wallet manifest file reflects the web applications compartmentalisation: every component – Wallet, Hub and KeyGuard – runs on a different domain, hence locally stored information like the wallet key is inaccessible to the Hub or Wallet due to the same-origin policy.

With this setup, it is easy to compartmentalise the development process, too. A separate developer key could be used for the KeyGuard code given that it is already bound to a second domain. New KeyGuard releases would need to be signed by that key, which, internally, can be assigned additional oversight requirements. Without requesting a new key from the PKI, any bypassing of this procedure would either end up with code that cannot access the user's key or be provable with the signed manifest for the Wallet.

4.8 Measurement procedure

We present a practical active content measurement procedure that can be used to identify active elements and collect their metadata, allowing the client to check whether the web application complies with the provided manifest. In development mode, the same procedure can be used to automatically generate a manifest file from an HTML document.

The measurement procedure is depicted in Fig. 4.2. The browser's rendering engine parses the raw HTML document and creates the DOM, observing the DOM for mutations, e.g. elements that are added at run time. Whenever an active element is appended, edited or removed from the DOM, the metadata agent will be triggered, which keeps a list of the active elements and their metadata.

The extension obtains access to the DOM by defining a *content script*, a script that runs in the context of the current page. This includes all pages loaded in top-level browser windows (e.g. Tabs), but also iframes within those. Content scripts running at the top level are responsible for collecting metadata on all active elements in their context. For nested iframes, they can only collect the metadata *about* the iframe like the attributes *src_type*, *src* and *sandbox*, but not inspect the document *inside* this

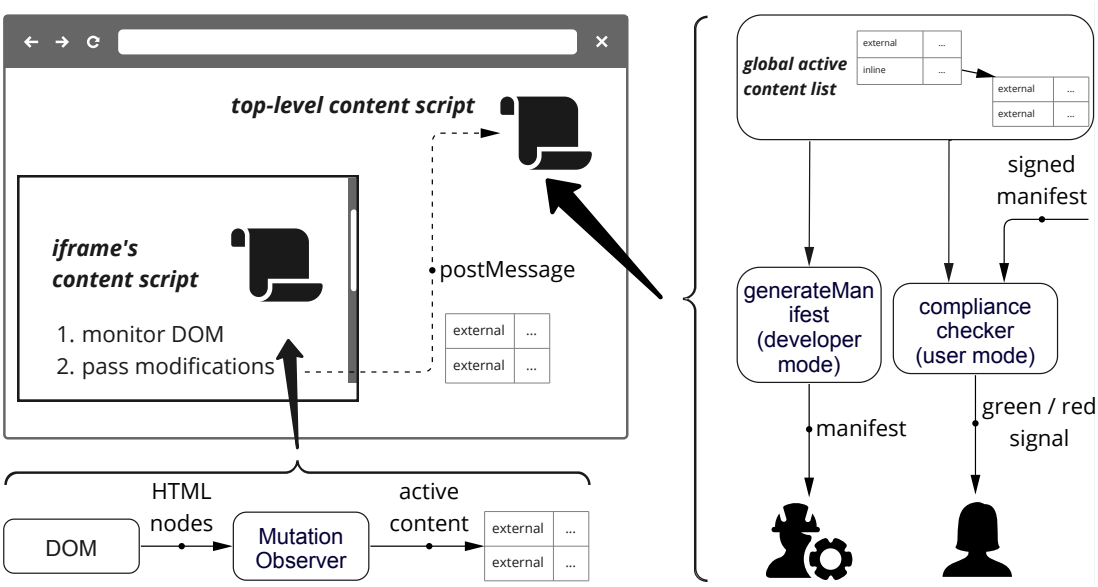

Figure 4.2: Manifest file generation and metadata collection .

iframe. The same-origin policy forbids this in many cases. We therefore use the iframe's content script to gather information: if the content script recognises that it is not at the top-level, it runs statelessly, collecting the metadata as usual, but reporting it to the parent window's content script via postMessage.

The metadata agent distinguishes script and iframe elements by their HTML tags. A script that has *src* attribute is *external* otherwise it is *inline*. For external scripts SRI hashes, crossorigin and load attributes are collected. For inline scripts, hash is computed on the script and the load attribute is collected. Event handlers are searched inside all DOM elements checking if their attributes contain any of the global event attributes e.g. onclick in a given list [81]. For event handlers, the hash is computed on the value of the event attribute. For iframes, the metadata is collected based on *src_type* which is *srcdoc* if the iframe has *srcdoc* attribute, otherwise *script* if the *src* attribute has a script as a value, and *external* if the *src* attribute has a URL value. For iframes with *srcdoc* or *script*, a hash is computed on the *srcdoc* or *src* contents, and crossorigin and sandbox attributes are collected by the metadata agent in the parent window. For iframes with *external*, the metadata agent in the parent window collects the crossorigin and sandbox attributes and gathers the metadata about the document inside the iframe from its content script. In addition, for each active element boolean *dynamic* and *persistent* scores are assigned by the metadata agent. An active content is considered dynamic if it is added after the window's load event; otherwise, it is static. Elements that get removed from the DOM are marked to be non-persistent, but still kept in the active content list for evaluation.

An opt-in website could be opened in a popup or an iframe, the measurement procedure runs as usual in this case. If the opener/parent window is not opt-in, the measurement will only take the popup/iframe website into account. However, if the opener/parent window that is in the same origin, it can cause changes in the popup/iframe context

without triggering the mutation observer in the popup/iframe. This could undermine accountability, hence, we require the opener/parent window must also be opt-in if the popup/iframe is in the same origin.

If the web page opted in, i.e. it has sent the `x-acc-js-link` header in the past and provided a valid manifest, then the metadata collector compares the metadata list with the list of active elements in the manifest. If the web page does not comply with the protocol, the extension reports this to the user.

In developer mode, a failure to comply triggers the manifest generator to collect and generate metadata for the active elements. The `generate_manifest` procedure then produces a manifest file with *trust* = *assert* for each active element based on the collected information, which can be easily adapted to other trust settings. This manifest represents the most restrictive manifest functional for this web application.

4.9 Signing and Delivering a Manifest

A valid signature on the manifest proves that the manifest was created by a known origin, i.e. a developer publicly associated with the website, and that it was not tampered with in transit. To sign manifests, we adopt the SXG standard [196]. SXG is an emerging technology that makes websites portable. With SXG, a website can be served from others, by default untrusted, intermediaries (e.g. a CDN or a cache server), whereas the browser can still assure that its content was not tampered with and it originated from the website that the client requested. This allows decoupling the web developer from the web host and nicely fits our view of websites as software distribution mechanisms. The SXG scheme allows signing this exchange with an X.509 certificate that is basically a TLS certificate with the 'CanSignHttpExchanges' extension. Browsers will reject certificates with this extension if they are used in a TLS exchange, ensuring key separation. SXG certificates are validated using the PKI, allowing Accountable JS to be used with the existing infrastructure, although, currently, Digicert is the only CA that provides SXG certificates [49]. The lifespan of an SXG certificate is at most 90 days [196], limiting the impact of key leaks.

An SXG signature includes the HTTP request, as well as the corresponding response headers and body from the server. The signature is thus bound to the requested URL, in our case, the manifest file's URL. It also includes signature validation parameters like the start and end of the validity period and the certificate URL. If the current time is outside the validity period, SXG permits fetching a new signature from a URL. This URL is also contained in the (old) signature's validation parameters. These features provide a solid foundation for Accountable JS's signed manifests, allowing manifests to be cached during the validity period and enabling dynamic re-fetching and safe key renewals.

A web application in compliance with Accountable JS must deliver the signed manifest. If it is small enough, it can be transmitted directly via the HTTP response header (using the directive `x-acc-js-man`). Alternatively, the response includes the URL of the SXG file, using the HTML meta-tag or HTTP-response header `x-acc-js-link`. The

signature in this file includes the manifest file (as the HTTP response body) and the manifest URL (part of the HTTP request). In addition, the browser needs to check that the URL value in the manifest corresponds to the web application's URL (excluding the query part of the URL).

Providing a signed manifest indicates the website (i.e. the URL) opted into the protocol. From now on, the extension will expect an accountability manifest until the users explicitly choses to opt out.

Apart from the manifest generation, the signing operation and uploading the signature to the ledger can also be automated thanks to existing tool support for SRI and SXG. We stress that the signatures need only be computed if the JS code changes. Techniques like microtargeting are disincentivised by accountability (see Section 4.3.1), hence the performance of the signature generation is of secondary concern.

4.10 Protocol

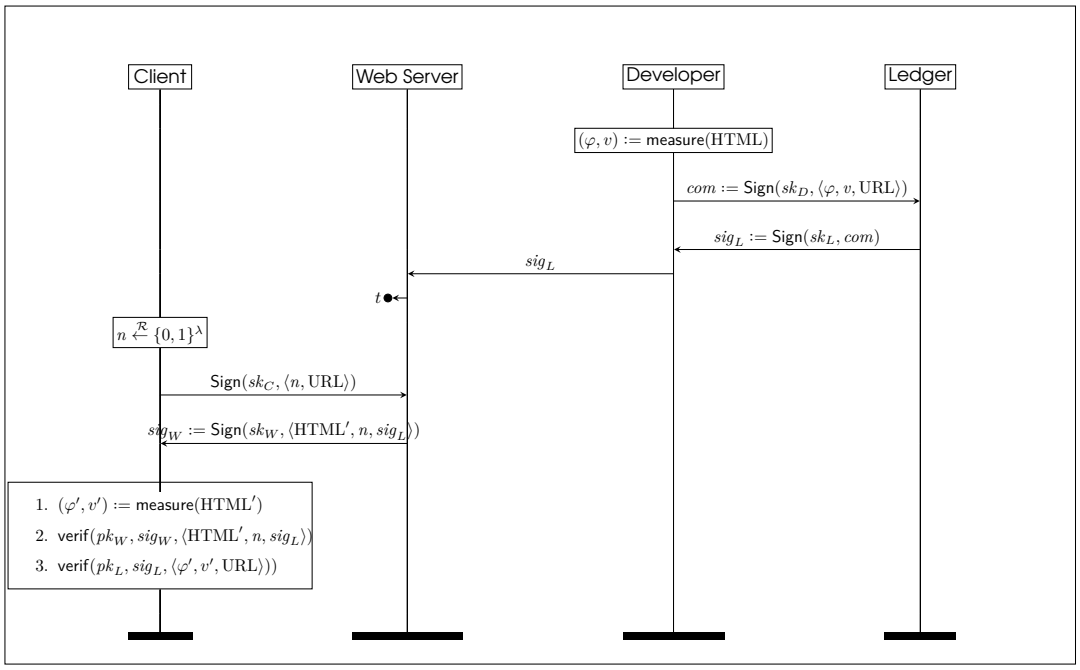

Figure 4.3: Protocol flow: CodeStapling (before t) and CodeDelivery (after t).

.

In this section, we present the Accountable JS protocol. The end-to-end goal is to hold the developer accountable for the active content the client receives. Clients can compare this code with the manifest, hence, for honest clients, we can reformulate this task as follows:

- Clients should only run active content that complies with the manifest. This is a setup assumption.
- Any manifest the client accepts needs to originate from the developer, even if the developer or server is dishonest. This follows from the non-repudiation of origin

property of the signature scheme. A signed manifest was either signed by the developer, or the developer leaked their key.

- Whenever two clients accept a manifest with the same version number, that manifest must be the same, or they can provide non-repudiable proof that this was not the case. This is achieved by including a transparency log that gathers all manifest files with valid signatures.
- Whenever a client accepts a manifest with some version number, this version was the latest version in some client-defined time frame. This is achieved by a timestamping mechanism similar to OCSP-Stapling [53].
- A client can provide non-repudiable proof that they received a manifest from the web server. This is achieved by signing a client-provided nonce.

The developer of the website generates a manifest file for the web page that is identified with a URL, signs the manifest and publishes it in one or more public transparency logs (see Fig. 4.3 before t). The signature proves to the client that the developer takes responsibility of the manifest.

The CodeStapling protocol ensures that, whenever the client accepted a manifest, the developer can be held accountable for publishing it. Nevertheless, the developer cannot be held accountable for delivering it to an individual client, as there is no proof for that. We thus define the CodeDelivery protocol for non-repudiable code delivery (in Fig. 4.3 after t). With the HTTP GET request, the client C sends a nonce n signed with its individual signing key sk_C. The web server W responds with a signature on the HTTP response HTML, the client nonce n, and signed log timestamp sig_L. The client validates the log's signature and the developer's signature within. Should one of these checks fail, the client aborts and displays an error message. Then, the client compares the active content in HTML with the manifest; if they are consistent, the browser decides the web page adheres to the protocol.

4.11 Protocol Verification

We analysed Accountable JS with Tamarin [112], considering the protocol's security w.r.t. a Dolev-Yao adversary that can manipulate messages in the network and corrupt other processes to impersonate them. Using Tamarin's built-in stateful applied-π calculus [92], we were able to model a global state such as represented by the transparency log.

The protocol comprises five processes running in parallel:

$$!P_{Developer} \mid !P_{Webserver} \mid !P_{Client} \mid !P_{Log} \mid P_{Pub}$$

The first three processes model the role of the developer, web server and client, outputting and accepting messages as specified in Figure 4.3. The developer, web server and the client are under replication to account for an unbounded number of parties acting in each role. Any party except the log and the public process can become dishonest. This is modelled by giving control to the adversary, but only after emitting a *Corrupted*

event, which can be used to distinguish the party's corruption status in the security property. A corrupted party remains dishonest for the rest of the protocol execution.

The process P_{Log} models an idealised append-only log using insert and lookup operations to a global store [92]. Moreover, the built-in lock and unlock commands are used to ensure atomicity of the operations. Finally, the process P_{Pub} make the public's ability to validate a client's claim explicit. Upon obtaining a claim (from the client), this process (1) reads from the log the information that concerns the URL mentioned in the claim, (2) verifiers the signatures in the claim and (3) matches the signed values with those in the log.

Using Tamarin, we prove the following properties which are detailed in Accountable JavaScript Appendix Section 8.1.

- **Authentication of origin**: The client executes active content only if the corresponding manifest was generated by the honest developer unless the adversary corrupts the developer.
- **Transparency**: If the client executes code then its manifest is present in a transparency log in a sufficiently recent entry.
- **Accountability**: When the public accepts a claim, then even if the client was corrupted, the code must exist in the logs and the server must have sent that data (either honestly or dishonestly via the adversary).
- **End-to-end guarantee** : Only by corrupting the developer it is possible to distribute malicious code.

4.11.1 Code Verify Protocol

Meta's Code Verify [76] was published in March 2022 and made available as an extension. As of now, it is deployed only by WhatsApp Web. Intuitively, WhatsApp Web (the developer) submits a hash of their JavaScript along with a version number to Cloudflare, which Cloudflare then publishes to the end user. The end user's browser extension computes a hash on the JavaScript delivered from WhatsApp Web and compares it against the hash published by the Cloudflare. Given that the manifest is hashed instead of signed, Cloudflare is trusted for authenticity and thus constitutes a trusted third party, replacing the log. Moreover, users' IP addresses are sent to Cloudflare instead of to WhatApp Web.

We likewise modelled Code Verify in Tamarin, considering the following five processes:

$$!P_{Developer} \mid !P_{Webserver} \mid !P_{Client} \mid !P_{Cloudflare} \mid P_{Pub}$$

Again, we assume the developer is separate from the web server. The protocol does not have a public log and does not include independent auditors. Instead, Cloudflare records the hashes for each version. To our knowledge, Cloudflare does not provide information about the history of submitted versions or which is most recent. As the public cannot inspect how often versions have changed, it relies on Cloudflare to implement countermeasures against microtargeting. Publicly available information [76] did not give information about such measures in Meta's deployment.

Under these considerations, we analysed the same properties, except for transparency, which, due to the lack of a public log, could not apply. We highlight the differences to our original properties below.

- **Authentication of origin**: The client executes active content only if the corresponding manifest was generated by the honest developer unless the developer or Cloudflare is corrupted.
- **Non-Accountability:** The data provided to the client is not sufficient to prove they received certain content from the web server, even if web server and Cloudflare are honest.
- **End-to-end guarantee**: Only by corrupting the developer or Cloudflare it is possible to distribute malicious code. In a separate lemma we show that, the developer by itself can indeed distribute malicious content.

The latter property indicates that Cloudflare's role as trusted party is not fully exploited yet. At least as far as we know [76], Cloudflare neither promises to ensure the code is harmless, nor does it guarantee to collect information to provide transparency or accountability. Nevertheless, the current message flow can be extended to provide such guarantees by having Cloudflare acts as a transparency log. Accountability can likewise be achieved by simply deploying signatures instead of a hashing scheme.

4.12 Logging Mechanism

We would like clients to verify they received the latest and same version of the code as any other user. To this end, we assume a public append-only log to provide a public record of the software published and prevent equivocation attacks. The log does not determine which JS is considered malicious, but it provides proof of receipt and origin, and allows identifying malicious versions.

Such a public log is realistic to deploy: CT Logs [134] are used in the modern internet infrastructure. These logs store certificates, which are signed by CAs. In contrast, our logs need to store manifests signed by the developers. It is thus not possible to simply reuse the existing CT infrastructure, but we can closely follow the structure and properties of CT.

Websites that offer security-conscious services have an incentive to retain their reputation. Similar to how CT logs operate, our log can be run by a party that wants to support such webpages. Third party monitors can keep the monitor honest and we allow third parties to submit signed manifests they observe in the wild.

When implemented naively, a logging mechanism can have significant privacy implications: To confirm that other clients receive the same manifest, the client would need to consult the log on each request and reveal the URL to the log. We can mitigate these privacy issues by allowing the web server, which learns each request anyway, to include a signed and timestamped inclusion statement from the log instead. This is similar to the OCSP stapling for certificate revocation status requests [128]. While it mitigates the privacy issues of consulting the log, it instead requires the user to trust

the specific log selected by the web server. We outline other approaches to solve the trade-off between trust and privacy in Section 4.15.

Overall, our transparency log needs to provide interfaces to at a minimum:

- store the signed manifest file (including its version number) bound to a URL,
- query the latest signed manifest file for a URL in the logs,
- form a signed response for a query that can be pre-fetched by the web server to staple it to each request from the clients.

A possible implementation of this functionality could be based on Verifiable Log-Based Maps [56]. An implementation of this structure for Trillian [161], the software running Google's CT server, is currently in progress [162], with the goal of supporting transparency in certificate revocation [98].

Availability, scalability and the size of the transparency logs are other implications. Be it submitting a new manifest to the log or collecting the latest version of manifest for a URL, low latency to access the network of transparency logs can be achieved by eliminating the single point of failure by adding multiple logs that will provide load balancing. The mechanism proposed for query privacy will also decrease the number of requests to the logs since the web server will provide the stapled result in most cases.

Websites that frequently update their active contents can create significant burden on the log size. We calculate approximately how many times each log can be updated for a limited time and space. We assume a non-leaf node overhead is approximately 100 bytes and for the leaf nodes it is 700 bytes(signature 600 bytes + 100 bytes). If a log provider has 100 TB of space for 5 years, it can contain 137 billion signatures in total. To make sense of this number, take the following example. We start with a log of 10M URLs with eight updates per month on average. The number of URLs also increases exponentially at a rate of 1% with each update (i.e. also eight times per month)[7]. This number would be well below 137 billion signatures.

4.13 Evaluation

We implemented Accountable JS in a Chrome extension [58] for demonstration and prototyping. Ideally, the measurement procedure should be part of the browser's rendering engine, since it can access the response body and observe mutations to elements first-hand. Our measurements here can thus be seen as (promising) upper bounds. We elaborate on the technical limitation imposed by the extension SDK in Section 4.14.

We come back to the use cases from Section 4.7 and measure how the extension affects the following metrics: 1. number of additional requests, 2. size of additional traffic, 3. time until the browser paints the first pixel / the largest visible image or text block[8]

[7] e.g. after the first update, 10M updates along with 100k new URLs are appended to the existing 10M, resulting in a total of 20.1M.

[8] More precisely: the 'largest contentful paint'.

/ until the web page is fully responsive. 4. total blocking time, i.e. time during which web page cannot process user input. We consider differences below 100 ms to be imperceptible to the users, differences of 100-300 ms barely noticeable and differences above 300 ms noticeable.[9]

Table 4.2: Evaluation results on case studies: The second and third columns show the number and total size of additional requests made by the extension, i.e. the number of signed manifest and certificate. Each subsequent block provides Lighthouse performance metrics for rendering time and the total time that the browser spends unresponsive. For each metric, we compare the baseline (no CSP, no Accountable JS) with the overhead incurred by enabling CSP and enabling the Accountable JS extension (leaving CSP disabled). For compartmentalisation, the baseline is with the extension activated but the same signing key for all Nimiq components. All the time values are averages over $n = 200$ runs and given in milliseconds. The additional traffic(kB) value is affected by the size of the signature and SXG certificate. Signatures are generated on uncompressed manifest JSON files.

	additional network ...		time to ... baseline + CSP overhead + Accountable JS overhead											
case study	requests	traffic (kB)	first pixel			largest element			reactive			blocking time		
Hello World	2	2.06	196	+1	+20	197	+0	+23	196	+1	+24	0	+0	+0
Trusted Third-Party	2	2.46	462	+0	+21	462	+0	+21	462	+0	+21	0	+0	+0
Delegate Trust (Nimiq A)	3	9.93	262	+3	-10	262	+3	-10	5591	-29	-144	172	+4	+87
AdSense + Nimiq B	3	15.62	747	+2	+91	901	+5	+68	6034	+1	-82	159	+3	+77
Compartmentalisation	2 + 2	8.66 +1.10	2200		-17	4675		+20	5321		+115	212		+7

Evaluation environment Measurement took place on a MacBook Pro with 2 GHz Intel Quad-Core i5, 16 GB RAM and macOS Monterey 12.5.1 with Google Chrome 107.0.5304.121. The results are compiled in Table 4.2. We measured the number of additional requests and traffic using Chrome's developer tools and the rendering metrics using Lighthouse (set to 'desktop simulated throttling'). Unfortunately, WhatsApp Web is incompatible with Lighthouse, so we instead computed the combined duration of all tasks performed by the browser using Puppeteer Page metrics [125]. We automated this process using Puppeteer and NodeJS and perform $n = 200$ trials per website and configuration to minimise the impact of network latency on page loads.

Configurations For performance evaluation, we compare the CSP built into the browser with the Code Verify and Accountable JS extensions as follows:

1. **Baseline:** disabled CSP and extensions.
2. **CSP:** CSP active, no extension.
3. **Accountable JS:** CSP inactive, only Accountable JS extension active.
4. **Code Verify:** CSP inactive, only Code Verify extension active. This configuration only applies to WhatsApp Web, as Code Verify currently only supports Meta websites.

[9] We derive these performance categories from the RAIL model [108]. According to RAIL, users feel the result is immediate if < 100 ms and feel they are freely navigating between 100-1000 ms (see also [122]). However, we found this gap is too wide to ignore, and split the category at 300 ms for an unusually common delay in web apps due to the 'double tap to zoom' feature on iPhone Safari [169].

Experiments We consider the examples from Section 4.7: Hello World, WhatsApp Web, Trusted Third-Party, Delegate Trust to Third Parties (Nimiq A), Untrusted Third Party (Google AdSense and Nimiq B). For the compartmentalisation experiment on Nimiq's Wallet, we use a different baseline that we will discuss below. For the CSP measurement, we defined CSP headers for each website that listed all active content in the Accountable JS manifest files. We collected all valid sources of external scripts and hashes for the external and inline scripts in CSP's `script-src` directive, hashes for event handlers in `script-src-attr` and sources for iframes in `child-src`. For the Accountable JS experiment, we first navigate to the target website and wait for ten seconds for the page to load. Thereafter, using the `generate_manifest` in the extension, we download the manifest file and self-sign it using the $gen - signedexchange$ command line tool [148]. For Nimiq A+B and AdSense, we changed the *trust* attribute for the external element(s) to *delegate* before signing. We publish this signed manifest via a local web server and configure the web server to provide a response header pointing to a URL. We also ensure the website provides *SRI* tags for *external* scripts. Evaluation procedures of each case study are elaborated in the Accountable JavaScript Appendix Section 8.5.

Results The CSP configurations show an imperceptible overhead in all case studies. This is hardly surprising, as CSP is built into the browser built-in and can validate resources during rendering. A detailed CSP defined for Nimiq A (Nimiq including its own CSP) increases the reaction time by about 65 ms. The Accountable JS configurations likewise have an imperceptible overhead in all case studies. Moreover, the traffic requirements are modest and incur only modest blocking time. For Nimiq A, the traffic requirements are about 9.9 kB for the additional signature. In terms of performance, CSP and Accountable JS' overhead are comparable. The time to interactive value unexpectedly decreases more with Accountable JS than CSP. However, the difference is minimal and could possibly be explained by a) network latency, (b) side effects of the browser's just-in-time compilation or scheduling or (c) a side effect of the former two on how Lighthouse evaluates the reactive metric. Nimiq is a complex web application heavily dependent on external data, in particular the remote blockchain it connects to.

Discussion The Accountable JS configurations have an imperceptible overhead which is slightly higher than the CSP configurations. Recall that the CSP is built in the rendering engine whereas Accountable JS runs as a browser extension. Accountable JS has to perform signature validation, meta data collection and a final compliance check. The prototype achieves good performance overheads by measuring all elements simultaneously and combining their results. The browser extension panel displays the results instantaneously, while the evaluation is in progress, although the evaluation is usually too quick for the user to notice. Moreover, the traffic requirements are modest and incur little blocking time.

For AdSense + Nimiq B, the network overhead is slightly higher than Nimiq A. This is due to the larger size of the manifest, which now also includes AdSense. We again observe an imperceptible impact on performance with Accountable JS.

The difference between Code Verify (220ms) and Accountable JS (244ms) on WhatsApp Web is small. This is remarkable, because Code Verify only applies *SRI* checks on external scripts but not event handlers or iframes. In contrast to Accountable JS, the order of active elements is ignored, attributes are not checked (e.g. `load='async'` for scripts) and a short hash value is downloaded from Cloudflare, rather than a signature.

Compartmentalisation For compartmentalisation, we evaluate the impact of the additional signing key. We signed Nimiq Keyguard, which is embedded in Nimiq Wallet, with a different signing key and set the Keyguard's *trust* attribute to *delegate* in the Wallet's manifest. The baseline therefore also has the Accountable JS extension activated, but uses the same signing key on all Nimiq components. The Wallet's manifest includes the Hub's manifest inside and the Hub's manifest declares the Keyguard with *trust* = *delegate* in its manifest section. Thus a separate manifest is required for the KeyGuard. Also, this time a separate signing key is used for the KeyGuard manifest. For the baseline performance, we inline the KeyGuard's manifest as an entry for its iframe in the Wallet's manifest, thus having one manifest and one signing key, and activate the extension.

In the compartmentalisation evaluation, we observe that there are two more round trips and slightly higher traffic overhead (relative to the overhead of Accountable JS, w.r.t. the overall page traffic of 4.6 MB). This is due to downloading the extra SXG certificate and manifest for Keyguard. The effect on the rendering metrics is small; the barely noticeable increase for time-to-reactive value can again be explained with network latency and side effects described above. This is due to the fact that the delegated manifest can be validated in parallel to rendering, while it is inlined in the baseline scenario and thus validated in sequence.

Due to stapling, the overhead for clients to verify that they received the latest version of the code (and thus the same as any other user), is negligible. The web server staples a query result, i.e. the log's signature on the signed manifest, to each request. The signatures use 2048-bit RSA keys and are 256 Byte long.

4.14 Limitations of Prototype

The browser extension is a prototype to evaluate performance and applicability of the approach. The advantage of an extension (as opposed to modifying the browser's source or writing a developer plugin) is that users can easily experiment with its code. On the other hand, extensions cannot interrupt the browser's rendering engine. Thus we inject a content script [45] that can apply the client-side operations of the protocol to the browser window. The content script runs in the same context as the web page; hence it can observe changes to the DOM via the Mutation Observer. Since the extension cannot access to the browser's rendering engine, there is the possibility that some active elements are added within a small time frame before the Mutation Observer is registered. This race condition is a limitation of using the extension and fixable by closer integration into the browser.

Another limitation is that other browser extensions may interfere with the measurement

by injecting active content to the web page. Since extensions cannot distinguish website code from the code that other extensions injected to the web page, this can break the measurement. This is the correct behaviour, as the website developer cannot attest to every possible modification of the active content by other extensions, however, there are various client-side solutions: (a) closer integration into the browser could distinguish active content injected by websites, (b) the extension could provide an API for third party extensions to register modifications or (c) an allowlisting for the most common extensions that gives a warning to the user.

4.15 Related Work

We first discuss how Accountable JS relates to other (proposed) web standards with seemingly similar goals, before discussing related academic proposals.

CSP was introduced to counter Cross-Site Scripting (XSS) attacks. They specify runtime restrictions for the browser, typically the set of allowed sources for scripts, iframes, stylesheets, etc., including eventual requirements for sandboxing. Like accountability manifests, CSPs can specify which sources are allowed and, combined with SRI, fix their content. This is comparable to a manifest file that includes types with *trust* set to either *assert* (if SRI is employed) or $blind - trust$ (otherwise). By contrast, CSPs do neither cover the order nor possibly nested active contents (e.g. iframe within iframe). Mixed ordering of active content may create malicious activity, a site loading script A before script B may mean something different from loading B before A. A site that only uses CSP cannot catch that behaviour, whereas in Accountable JS, we take the order into account. Most importantly, in CSP, there is no means of delegating trust and no distinction between web server and developer. Steffens et al. [153] show that outsourced content is one of CSP's major deployment obstacles. Instability in third party inclusions (e.g. adbidding code that delivers code from different resources) forces first parties to continuously update the CSP. Techniques like in Section 4.7.5 allow developers to delegate trust to the third party. It is conceivable to incorporate features like trust delegation into CSP, along with a mechanism for signing CSP headers. By contrast, capturing the order or the nesting between active elements and providing non-repudiation appears to clash with the design of CSP. Moreover, CSP tries to mitigate XSS attacks throughout the web, whereas Accountable JS targets websites willing to allow for an audit. The ability to identify the code that is run is a key requirement for that. Overall, the goals of CSP and Accountable JS are orthogonal and can be combined. It is possible to generate a CSP from a manifest file.

The Web Package proposal (currently in draft status [197], see Fig. 4.4) aims at packaging web applications for offline use. Web packages provide a declaration of the web application's metadata via *Web App Manifests* [105], a serialisation of its content via *Web Bundles* [198], and authenticity via SXG [196]. We likewise employ SXG to provide authenticity of origin via signatures.SXG, like Accountable JS, decouples web developer from website hoster. Web App Manifests, despite their name, are only superficially related. They contain startup parameters like language settings, entry points and application icon, e.g. for 'installable web application' displayed in a smartphone's

launcher. Web Bundles are a serialisation format for pairs of URLs[10] and HTTP responses. They represent a web application as a whole, but a signature on a web bundle would change with every modification of a web pages' markup. Web Packages are thus not competing with Accountable JS. Instead, both standards are compatible. A web bundle can contain `x-acc-js-link` in the header of its entry point's HTTP response, triggering the browser to validate the manifest. The manifest is specified via a URL that also included in the web bundle. This URL maps to an HTTP response that contains the manifest in its content part.

Signature-based SRI [172] proposes easier maintainable SRI tags to protect against script injections, by including signature *keys* instead of hashes. These enable validating the provider of the third party script, instead of their content, similar to the trust relationships expressed with *trust* = *delegate*. The tags are part of the HTML code, instead of the manifest file. Signing the HTML files is impractical, as they are frequently changing.

Service Workers [138] are Network proxies programmable via JavaScript, often used to perform URL response caching, separate from the browser cache. Theoretically, a compliance check similar to our measurement could be implemented in a service worker, but (a) the service worker would need to be delivered correctly and (b) service workers lack access to the DOM and thus information about how active elements used.

We will now discuss related academic work. Accountability in the web requires non-repudiable proof. For static assets, this can, in principle, be provided by digital signatures (e.g. via SXG and web bundles, see above), but recreating the signature for each exchange is costly. We solve this via a simple challenge-response mechanism. Ritzdorf et al. [136] provide a full-fledged solution, giving non-repudiation for the entire communication, optionally hiding sensitive data. The statement we prove is that the client has *obtained* certain active content, not that they *execute* it. Ensuring a remote partner runs certain software is the goal of remote code attestation (e.g. [74]). Outside

[10] More precisely, HTTP *representations* [64].

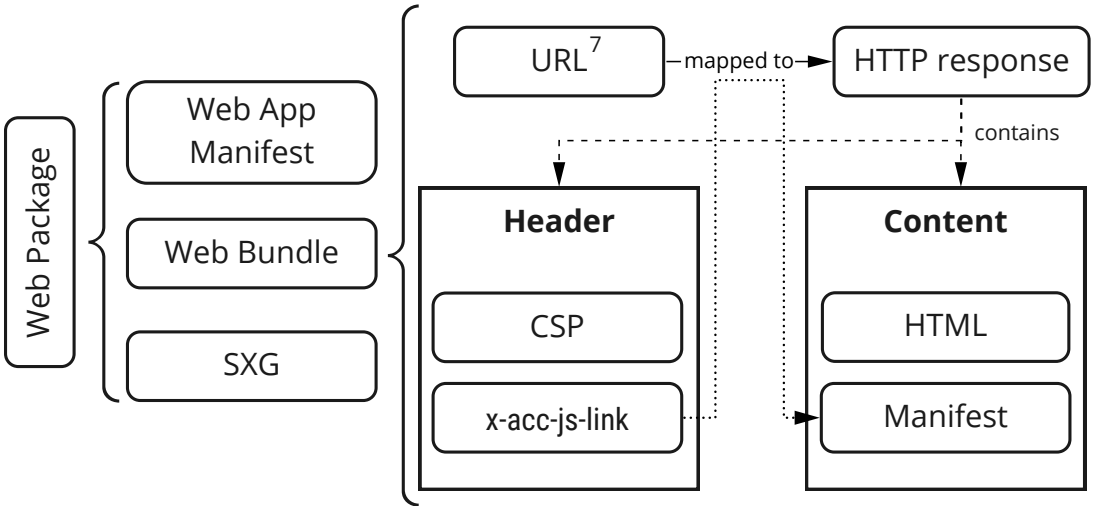

Figure 4.4: Accountable JS in the context of other web technologies.

embedded systems, this is typically based on a trusted execution environment (e.g. TPM, SGX). While the browser (and for that matter, our extension) could provide a trusted execution, establishing trust in the correctness of the browser is the crux.

Our work relies on a transparency log. As mentioned before, Trillian's [161] verifiable log-based maps would fit the bill, but there are many ways to implement such a store. The most interesting aspect is privacy. We propose an approach based on stapling, an established method for revocation management [53], but other techniques promise privacy, too. CONIKS [114] provides a log, mapping user identities to keys and keeping the list of all user identities in the system private. This would not help in our case, as the URLs (the domain of our mapping) are not secret, but which URL a user accesses. Multiparty protocols for Private Information Retrieval [66], Private Set Intersection (e.g. [130]) or ORAM [68] lack efficient database updates, mechanisms to efficiently update precomputation steps, or only preserve k-anonymity for URLs. K-anonymity is often not enough if we consider that an attacker, e.g. a censor, tries to punish access to a few critical URLs, each of which may end up in a bucket with uncritical, but also not frequently visited URLs. Finally, Accountable JS may be an enabler for formal verification of web applications, as users are potentially able to link the code they receive to code to published verification results. Various static and dynamic analyses target JavaScript already [126, 139].

Although we showcased only a single approach to code compartmentalisation (as it is being deployed by our real-world example), other approaches are also compatible with Accountable JS. Language-based isolation methods like BrowserShield [132]) rewrite JavaScript into a safer version preventing or mediating access to critical operations like `createElement` or `eval`. If the code is rewritten on the client (typically using JavaScript), the developer declares the wrapper that fetches the code and deals with the code rewriting in the manifest file. If the code is rewritten on the server, the developer declares the transformed JavaScript code that will be delivered to the user. Frame based isolation methods (e.g. AdJail [103]) that isolate the third party code inside iframe are also compatible with our proposal, see the use case for untrusted third party code in Section 4.7.5.

4.16 Discussion

We provide a solution that allows users to detect if they are microtargeted by developers and to prove this to the public if it is the case. Sending different codes to classes of users might not be outlawed in many countries, but sending malicious code is. Our solution neither provides a code audit tool nor does it propose a framework, legal or otherwise, for the punishment of malicious code distribution. It provides, however, verifiable data that authorities can use to evaluate which code was published and whether that code was delivered to a specific user. Moreover, the protocol provides users with a claim that includes the delivered code and the identity of the developer.

The transparency logs can be used as a point of reference for the public code for auditing and evaluating. Honest developers aim to make their code easy to audit; dishonest developers thus risk loss of reputation if they microtarget users (as frequent updates are

visible on the transparency logs), silently opt out of the system (as this will be caught by users that received a previous opt-in), or provide obfuscated code (due to the log).

Honest developer will benefit from a good reputation and their ability to provide proofs for any efforts they make toward independent audit or formal verification. Clients, who often debate a website's reputation in a public forum (e.g. the case of ProtonMail or Lavabit) obtain data to substantiate positive and negative claims.

We stress that accountable code delivery is necessary to connect auditing results to the code users actually run, but does not by itself guarantee the safety of this code. Realistically, it will take some time until software analyses are mature enough to handle this at scale. Assuming, however, that such analyses may not necessarily run at each browser independently, authentic code delivery appears to be a necessary first step.

Moreover, Accountable JS only authenticates the active content, thereby exposing the active content to data-only attacks, e.g. modified button labels or form URLs. A signature on the content of a web application could be achieved by building on Web packages/Web bundles (which we discussed in Section 4.15); however, this approach would be too static and inflexible for the requirements of the current web ecosystem. Thanks to accountability, the developer would take responsibility for the active content that they published, in this case, for code that is vulnerable to data-only attacks. Realistically, there would not be consequences, because they can plausibly point to the dire state of verification of JavaScript—which is at least partially because users could thus far not be sure to receive the verified code anyway. Accountable JS choice to validate the active content only is a compromise and possible starting point for future work, as we discuss in the next section.

4.17 Conclusion

With Accountable JS, we provide a basis for the accountable delivery of web applications, and thus a first step towards re-establishing the trust between a user and the web application code they run on their computers. How to actually achieve security – via audit, code analysis or formal verification – is a question that we left open on purpose. Accountable delivery is, nevertheless, a requirement for any non-instantaneous analysis.

A key question for verification and audit is how to relate the web page's user interface to the active content. As some desirable security properties concern user input, we would like to give guarantees about, e.g. form fields. We can account for the JavaScript code that address them by ID, but those are invisible to the user. Future work may investigate how to establish stronger ties between the manifest and the user interface.

5

Formal Browser Model for Security Analysis

As web technologies continue to evolve, ensuring the security and privacy of browsers has become increasingly complex. This chapter presents a novel framework that employs formal methods to identify security and privacy vulnerabilities in browser implementations with respect to Request for Comments (RFC) standards. By model checking detailed browser models against RFC properties, we systemically and exhaustively explore potential vulnerabilities in browsers. Our framework automates the process of testing contemporary browsers against these vulnerabilities to validate their real-world implications.

We validate the efficiency of our approach through two case studies. The first case study demonstrates a potential vulnerability that could allow unauthorised camera access by exploiting a browser's vulnerabilities in URL parsing, origin assignment and secure context assignment. The second case study shows a scenario where a website might engage in side-channel attacks between browsing contexts by abusing Cross-Origin Isolation State in the browser. In the first case study, we find a previously known camera attack in an old version of Safari browser and our analysis also shortens the original attack process. Our contributions include a comprehensive browser model, implemented as a state machine in Alloy, and an automated framework that integrates model checking, counterexample parsing, test-case generation and test-case validation in browsers. Our findings showcase the effectiveness of such rigorous analysis in the realm of browser security, particularly as web technologies advance every day.

5.1 Problem Statement

Web browsers serve as gateways to the digital world, connecting users with content, services and applications from the web. They rely on multiple standardised protocols and guidelines that are detailed in RFC documents. RFCs outline the ideal standards, best practices and informational guidelines for web protocols and browser functionalities. They offer a consistent and secure online experience in theory. However, in practice, real-world browser implementations may have different interpretations of the RFC specifications and different optimisations throughout the code base with respect to security, privacy, performance and user experience, resulting in inconsistent behaviours and potential security and privacy vulnerabilities across different browsers.

The change is also complicated. The web is not a static entity. It evolves with new technologies, user demands, and security and privacy challenges emerging regularly. RFCs need to encapsulate these evolving needs while balancing backward compatibility and progressive innovation. As these standards evolve, so do the browsers need to adapt to this ongoing cycle of changes.

Furthermore, browsers can rightly be considered as operating systems for the web, orchestrating multiple layers of functions from parsing URLs, establishing connections to websites, interpreting HTML, executing JavaScript to handling multiple windows, managing web resources, multitasking, accessing media devices, security and origin isolation and others. These functionalities are not just independent components operating in isolation, they are deeply interconnected, with each component influencing others. Even a minor change in one component may impact another, leading to unforeseen vulnera-

bilities. Therefore, it is imperative to undertake an exhaustive and holistic approach to testing browsers that encompasses all components and their complex interrelations.

To address these concerns, there is an evident need for frameworks that can bridge the gap between the theoretical world of RFC standards and the practical world of browser implementations. Such tools must be robust and adaptable to fulfil the ongoing change demands on web browsers. Beyond the robustness and adaptability, they should have an exhaustive search capability diving deep to uncover vulnerabilities within standalone browser components and the web of interrelations they form. Only by employing such comprehensive tools can we ensure the level of security and reliability that users expect from modern web browsers.

5.2 Overview

In this chapter, we propose a framework that uses formal methods to unearth vulnerabilities in browser implementations relative to RFC standards. We model check on detailed browser models that encompass browser functionalities, policies and APIs, against properties highlighted by the respective RFCs. The model checking systematically and exhaustively checks whether the browser model adheres to the property. When model checking identifies a violation of the property (also referred as a counterexample), our framework automatically creates a test case scenario from the violation and validates it on browsers to find out whether actual browsers exhibit this flawed behaviour.

Our framework is designed with adaptability in mind, new browser features or changes on existing browser functionalities, policies and APIs can be easily integrated to our framework. This is achieved through our model's modular architecture. This modular architecture enables rapid and accurate validation of the security and privacy implications of updated browser features. Moreover, new browser features and their interrelations with existing features can be accommodated to the framework and their implications can be validated before these features become part of the real-world browsers, contributing to ongoing improvement of browser security and privacy. Integrating new features to the framework requires considerably less effort with the modular architecture. Furthermore, the modular architecture of our framework also enables to easily represent interactions between browser components. Each module is designed with an interface that allows to communicate with other components, showing interdependencies between browser functionalities. This design helps capturing behaviours exhaustively that can arise from the interactions between functionalities and it enables more accurate and comprehensive validation of potential security and privacy vulnerabilities.

The framework executes the process in multiple phases:

1. **RFC Formalisation :**

   We begin with formally modelling the RFC behaviour as a basis for the standardised behaviour. This model serves as the reference against which all browser behaviours are evaluated. We build the model as a state machine in Alloy specification language. The model changes from one state to another state in response to an event using transition functions defined for the event.

2. **Browser Behaviour Analysis:**

   Subsequently, we examine different browser implementations to analyse and pinpoint their deviations from the RFC behaviours and individual characteristics.

3. **Divergence Integration:**

   We then integrate these identified divergences into our browser-specific models, ensuring the models reflect the current state of the specific browsers.

4. **Model Checking:**

   The rest of the process is mechanised by our framework. Utilising model checking which the framework provides, we automatically check to discover if the deviations from the RFCs can lead to any potential vulnerabilities on the browser models. For every vulnerability that is discovered, the model checker provides a model execution trace that shows the states of the browser components and applied events that led to the changes on the state in each step.

5. **Attack Validation:**

   Our framework generates test-case scenarios from these traces. The test-case scenarios consist of generic JavaScript codes to apply the events in the trace to the browser components. The test-case scenarios are then run on real-world browsers to ascertain the practical implications of the detected issues.

6. **Incremental Solving:**

   If the test-case scenario execution results in an actual attack on the browser, we store the counterexample in the system's directory. Subsequently, we resume the 'Model Checking' phase to identify additional counterexamples from the same model and property. This iterative process continues until no further counterexamples emerge from the model.

7. **Feedback Loop:**

   If the test-case scenario execution does not yield an attack, we return to the 'Divergence Integration' phase and refine our model based on the new findings of the browser behaviours and apply the same processes from this phase until we find an actual attack on the browsers.

Our formal browser models formulate the following concepts in browsers:

- Browsing environment elements : browsing contexts, browsing context groups and browser tabs [190],
- Document and HTML content : Document [191], DOM API [177], HTML elements (JavaScript and non-active elements), iframes and sandboxed iframes [182],
- Security and isolation elements : secure contexts [193], origin [192],
- Navigation elements : History API [185],
- Network elements : web servers [174], DNS servers [52], absolute URLs (HTTP/S) [16],
- Local resource elements : local URLs (about:blank [120], data [106], blob [1]),

- Security policies : Same-origin Policy [13], Cross-Origin Resource Sharing (CORS) Policy [179] and Cross-Origin Isolation State [183],
- Browser APIs (Events) : WindowOpen, Navigate and RenderResource [135],
- JavaScript APIs (Events) : History.pushState [186], Location.replace [187], CreateBlob [94], Popup [188], ReadDOM, WriteDOM, CreateIframe [178], AddSandbox [182], Document.write [184], AccessToMedia [109], SharedArrayBufferAccess [146].

To showcase our framework's effectiveness, we employ it on two distinct case studies. Here, we examine if our models could potentially open the way for unauthorised access to sensitive browser data. In the first case study, we analyse whether the browser model can result into an unauthorised media device access e.g. camera. This scenario explores how an untrusted website might exploit vulnerabilities in URL parsing, origin assignment and secure context assignment, to gain unwarranted access to the camera. Whereas, in the second case study, we examine whether the browser model can lead to side-channel attacks between browsing contexts. This case study delves into mechanisms in which an untrusted website in one browsing context can unjustly gain access to the powerful JavaScript functionalities that enable side-channel attacks on other browsing contexts. Notably, the browser models deployed in each case study are rigorously derived and consistent subsets of our detailed formal browser model denoted by M_{rfc}. The camera access model in the first case study is denoted by M_{cam} and the side-channel access model in the second case study is denoted by M_{sch}. This approach is adopted to circumvent the challenges posed by the state-space explosion problem inherent in model checking. We elaborate on this in Section 5.4.

We summarise our contributions in this chapter as follows:

- We develop a detailed browser model conceptualised as a state machine in Alloy. For practical applications in our case studies, we craft two distinct and consistent subsets of this model. Our analysis has identified a previously known camera attack in an older browser version and even made the attack process shorter in the first case study.
- We introduce a framework equipped to automate tasks such as model checking, counterexample parsing, test-case generation and test-case validation on browsers.

5.3 Outline

In Section 5.4, we describe the methodology used for the formal modelling of web browsers. Subsequently, in Section 5.5, we present the design and structure of the detailed formal browser model. Next, we outline the browser camera access model for the first case study in Section 5.6. Thereafter, in Section 5.7, we describe the Cross-Origin Isolation State and in a following subsection we outline the model for the second case study. In the following, in Section 5.8, we discuss the security properties and vulnerabilities targeted in analysis of the case studies. Next, in Section 5.9 we describe the process of generating test cases based on the identified counterexamples and explain how these test cases are used to simulate potential attack scenarios in the

browser. In the following, in Section 5.10, we present our evaluation of the effectiveness and accuracy of the formal models and analysis techniques we employed. Thereafter, in Section 5.11, we discuss the existing related work on web browser formal models and security analysis. In the following, in Section 5.12, we discuss the strengths and weaknesses of the proposed approach and any potential areas for improvement. Then, in Section 5.13, we summarise the main findings of the study and their significance.

5.4 Methodology

Our main objective in this chapter is to develop a formal model of web browsers and to evaluate the model's capability in detecting vulnerabilities that lead to unauthorised access to critical data in browsers. We construct a detailed model that encapsulates the essential features and behaviours of the browser, and we enhance the model by incorporating security-critical policies e.g. same-origin policy, cross-origin resource sharing policy and cross-origin isolation state. We generate two consistent subsets of the general model and employ these sub-models on two distinct case studies, recognising that verification of two different mechanisms on one large model can be limiting in model checking because of the state-space explosion problem [44]. These sub-models are tailored to explore distinct security and privacy concerns in greater depth and they are fine-tuned to detect specific kinds of threats.

For instance, the first specialised sub-model ($M_{\mathbf{cam}}$**, Camera Access model**) is tailored to detect vulnerabilities involving unauthorised access to user's media device e.g. camera, via the browser. This model mainly comprises various types of URLs, browsing contexts, documents, iframes, sandboxing, secure contexts, origins, history, web servers and DNS servers, and it incorporates the Same-Origin Policy. It is further refined to focus on specific functionalities that are required to access the media devices such as URL parsing, origin assignment, secure context assignment and media device access control. Our analysis investigates whether an attacker through a malicious website could exploit these functionalities in some cases to gain unauthorised access to a user's camera.

By contrast, the second specialised sub-model ($M_{\mathbf{sch}}$**, Side Channel Access model**) is crafted to detect vulnerabilities involving side-channel attacks between browsing contexts using powerful JavaScript APIs. This model includes essential browser elements like browsing contexts, browsing context groups, documents, iframes, HTML elements (JavaScript and non-active elements), DOM API, origins, web servers, DNS servers and absolute URLs. Moreover, it incorporates the Same-Origin Policy, Cross-Origin Resource Sharing Policy and Cross-Origin Isolation State. In this model, powerful JavaScript APIs (e.g. SharedArrayBuffer) are only accessible when a browsing context is in Cross-Origin Isolation State. Our analysis in this model examines for instance whether an attacker in a browsing context that is not in Cross-Origin Isolation State can gain access to the powerful APIs such as through a different browsing context that is in the Cross-Origin Isolation State.

We apply model checking to rigorously identify the vulnerabilities in browsers. To this end, we formulate the model using the Alloy formal modelling language. Model

checking is a systematic method that allows to exhaustively search all possible states to determine whether properties hold for the current model. By exploring the entire state space, model checking provides guarantees in detecting flaws or verifying the absence of them in the model. On the other hand, fuzzing is another prevalent technique to detect vulnerabilities in software. It involves providing random or grammar-based inputs to the software under test and observing its behaviours to identify weaknesses in a software. Fuzzing is a more computationally lightweight and automated approach in software testing. However, it has limitations, it cannot provide the same level of guarantees as model checking because it is not exhaustive, therefore it may not cover all possible cases and some edge cases that could compromise the software might go untested. Hence, we choose model checking to provide stronger guarantees. Nevertheless, we discuss how our platform can be enhanced by incorporating both the model checking and fuzzing in the Section 5.12.

In this work, we take the distinction between the RFC specifications and practical implementations into account. RFCs define standards for web protocols and functionalities, and the practical implementations from different vendors such as Safari and Chrome, realise those standards for their end users. RFCs are published by the Internet Engineering Task Force (IETF) and they describe the best practices to implement web features. However, different browser vendors may have different interpretations of the specifications and different optimisations throughout the code base with respect to security, privacy, performance and user experience resulting in inconsistent behaviours and potential security and privacy vulnerabilities across different browsers.

Our methodology includes a rigorous analysis on RFCs and browser implementations to identify those potential vulnerabilities. We start by formally modelling M_{rfc} which is the RFC behaviour as a basis for the standardised behaviour. Next, we analyse the behaviour of browser implementations e.g. Safari, noting any deviations or unique characteristics they exhibit from the RFC-defined standards. We then incorporate these discrepancies into our browser-specific model denoted by M_{cam} to assess whether they can lead to actual attacks.

To manage the complexity in the model, we apply an iterative refinement strategy in our modelling. Building a detailed model from the beginning would cause an excessive amount of counterexamples, making the model challenging to manage. Therefore, we firstly construct an abstract browser model that only represents very basic functionalities and behaviours of the browsers. Then, we iteratively refine the model by adding more details of the browser functionalities and behaviours based on the spurious counterexamples we encounter which do not lead to any actual attacks in the browser. This process goes on until the model provides a more comprehensive and accurate representation of the real-world browser.

To construct the model, we use Alloy [6] which is a declarative formal modelling language and is based on first-order logic and relations. Alloy's declarative language enables concise representation of complex models. It has been successfully used in various domains. In addition, it provides visualisation of models and its backend Kodkod [160] provides a library for Java development environment that we utilise for mechanising the whole model checking and counterexample validation process. We specify constraints

and relations in first-order logic. Alloy, in turn, translates the model into a boolean formula using Kodkod. Then, the formula is transformed into a satisfiability problem in which an off-the-shelf SAT solver attempts to find solutions for a given property. Alloy also provides bounded model checking that allows to efficiently explore a finite number of states in the model. This is useful for a large and complex system like the web browser, as we can define a smaller scope and analyse the properties within that scope and thus reduce the computational burden which can be extensive in exhaustive state space exploration. However, we also note that there may exist additional solutions in larger scopes which we did not explore.

Furthermore, Alloy offers a variety of SAT solvers. We choose the MiniSat solver [54] for its efficiency and performance. MiniSat solver also supports incremental solving which helps to efficiently find subsequent instances of the model by using the work that is done in previous analysis, reducing redundant computations. This optimisation enabled us to explore more instances of the model within a shorter timeframe. Alloy also provides three different decomposition strategies that can be parallel, batch or hybrid for efficient, scalable and flexible problem solving. In the parallel strategy, Alloy splits the problem into multiple subproblems and tries to solve each problem with concurrent processes. The batch strategy solves the problem by looking at all the signatures and fields together at once, covering every possible instantiation. The hybrid strategy is a combination of the parallel and batch strategies which performs parallel decomposition, but reserves one process for batch problems. We experimented with both parallel and hybrid methods. However, for our model, we found that these techniques led to slower model checking. Hence, we decided to select the batch strategy for decomposition.

In Alloy, there are two ways of analysing properties. One is finding an instance to a property that provides an instantiation of the model that satisfies a property. The other way is asserting the property cannot be refuted in the model for the selected scope. In the latter case Alloy attempts to find an assignment of relations in the model called counterexample in which the property is not satisfied. We generally use the second method to identify counterexamples to our properties, reserving the first method for sanity checks to test that our model is correctly implemented. This approach guarantees that the property holds for all possible instantiations of the model in the selected scope.

The counterexamples show potential vulnerabilities in the model. We further analyse these counterexamples to determine whether the vulnerability can actually be exploited in the browser. To mechanise the process, we implement a Java application that incorporates essential components such as Kodkod for translations of the boolean formula, SAT solvers to tackle the satisfiability problem, a parser to interpret the counterexample, a test case generator to replicate the same configurations and actions in the browser and a test case executor that attempts to execute the attack scenario. The application automatically analyses the specified properties and checks whether the identified attack can be realised in the browser environment. The application framework is explained in Section 5.9.

5.5 Formal Model Design

In this section, we present our detailed browser model designed to investigate unauthorised access to critical data within web browsers. This model serves as the foundational framework for our analysis and is further elaborated in the subsequent sections. In Section 5.6, we shift our focus to the specialised Camera Access model M_{cam}. This model is specifically engineered to identify vulnerabilities that could lead to unauthorised access to users' media devices, such as cameras, via browser interfaces. Following that, Section 5.7 explores the Cross-Origin Isolation State model M_{sch}. This standalone model is oriented towards security features that mitigate unauthorised data access at the microarchitectural level.

To implement the models, we use Electrum [104] which is an extension to the Alloy specification language. State machines can be modelled more easily with Electrum as it allows to model signatures and relations that can evolve from state to state in each model execution step. This enables an easier representation of dynamic systems.

```
one abstract sig Browser extends Client {
    var bcs : set BrowsingContext
    ...
}
```

Listing 5.1: Browser signature in Alloy

For instance, we can define a 'Browser' signature that has a mutable 'bcs' relation representing the set of browsing contexts in the 'Browser'. In this context, the 'var' keyword is used to indicate mutability. A new browsing context can be inserted to this 'bcs' set every time a new 'WindowOpen' action occurs in the model execution trace. Electrum also supports temporal logic operators within properties. Using the temporal logic operators, we can reason about properties of the system that involves time. For instance, we can define a safety property that says when a 'Navigate' action occurs on a browsing context, there must have been an earlier 'WindowOpen' action on the same browsing context in the trace.

We build the model as a state machine. The model changes from one state to another state in response to an event using transition functions defined for the event. In addition, some events are neutral and have no effect on the current state, hence the system will remain in the same state when those events occur. In fact, Electrum considers traces as infinite. Hence, all execution traces must end with a final state that repeatedly stays in the same state in response to a neutral event. This behaviour is sometimes called *stuttering*.

All browser components and browsing contexts are stored in the 'Browser' signature which represents the state in our model. To optimise the model-checking process, we aim to minimise unnecessary complexity in the model. Considering that all traces must start with at least one tab open in the browser, we adapt a simple initial state in our model that consists of a single tab opened in the browser but not navigated to any documents yet. In the model, that initial state is represented with a single browsing context appended to the 'Browser' signature and associated with StartupUrl

and StartupOrigin. Moreover, it is unaffiliated to any document, but ready to initiate navigation or other actions in the first step. Additionally, we initialise all other browsing contexts and associated relations empty. These elements are not immediately considered as part of the active state. Instead, they are appended to the state as the model transitions in response to the events with each iterative step.

To have a dynamic system in which the 'Browser' evolves with each step, we define a mutable 'Call' signature that serves as an interface for the transitions. The policies in the browser, e.g. same origin policy, are also enforced within this 'Call' interface in the model. It describes where the call originates from, its destination, arguments sent and results returned from this call. Furthermore, it is associated with an event that shows which function will be triggered from this call.

The 'Function' signature represents an event (or action) in the model. It is an abstract signature for all possible events e.g. 'WindowOpen', 'Navigate' and others. The 'rootBc' relation refers the root browsing context that is associated with the event and presence of this relation ensures that any event can only be called by a browsing context that is part of the state. In other words, it prevents a browsing context from triggering an event, when it is not related to one of the root browsing contexts in the 'bcs' relation in the 'Browser' signature. The 'bc' relation refers to the actual browsing context that the state transition will be applied when the event occurs. As stated before, it has to be related to one of the root browsing contexts in the state. The 'party' relation refers to the browsing context that triggers this event on the 'bc' relation. The browsing context associated with the 'party' relation should have a reference to the browsing context associated with the 'bc' relation e.g. the 'party' is the browsing context of a window that was previously opened as a popup by the window of 'bc'. Also, the Same-Origin Policy applies between the 'bc' relation and 'party' relation.

```
var one abstract sig Call {
    var from : lone (BrowsingContext + Script),
    var to : lone (BrowsingContext + Browser + Server),
    var args : set HtmlResource,
    var returns : set HtmlResource,
    var event : one Function
}
var lone abstract sig Function {
    var rootBc : one BrowsingContext,
    var bc : one BrowsingContext,
    var party : one BrowsingContext
}
```

Listing 5.2: Alloy Definitions for Modeling Browser Function Calls

We ensure that only one 'Call' and one 'Function' exist in each step in the model execution trace. The events trigger state transitions in the model. State transitions are predicates that describes the changes in the state. Each predicate is associated with an event. The switch below ensures the associated predicate is called depending on the event selected in the trace.

```
fact {
```

```
        always (all c : Call | exec[c.event, c])
    }

    pred exec [f : Function, c : Call] {
        f in WindowOpen implies window_open[f, c]
        f in Navigate implies navigate[f, c]
    }

    pred window_open[..]{}
    pred navigate[..]{}
```

Listing 5.3: Alloy State Transitions

We build the model on general web concepts in the browser and the interactions between browser features, web servers, DNS servers and JavaScript APIs. To capture the relationships between the browser, the web servers and the DNS servers, the communications in the model take place between endpoints which are servers (web servers) and clients (browser and JavaScript elements), though some interactions can also take place between the browser and JavaScript APIs.

There is only one single browser in the current design. It contains multiple browsing contexts modelled based on the HTML standard [190]. Browsing contexts serve as containers for the tabs and iframes in the browser. Therefore, every browsing context is associated with a tab or an iframe. The browsing contexts are responsible for displaying documents and non-active contents such as stylesheets, images and HTML elements. Additionally, scripts are executed within the browsing contexts. We elaborate on the browsing context features in the model as the following:

- A URL and an origin are set for a browsing context depending on the URL visited in its current document. Every browsing context starts with an initial URL that varies across different browser vendors in real life. We simplify these differences and we assign a 'StartupURL' in the beginning of each browsing context. Additionally, we assign a corresponding 'StartupOrigin' to each browsing context, which represents the origin of the 'StartupURL'. This origin does not have any capabilities and it is essentially a placeholder that is used only for the initial setup of the browsing context. The origin determines the access rights that a browsing context has. These access rights are enforced with the Same-Origin Policy [13] which regulates the interactions between different entities from different origins.
- Each browsing context is associated with a single active document that is currently being presented to the user when the user is visiting a website. This document may contain various HTML elements e.g. JavaScript or non-active elements. Initially, a URL corresponding to each element is sent in the first HTTP request to the server. Subsequently, the browser initiates additional resource requests to these URLs to retrieve the associated resources from the respective servers. Once a document has successfully acquired all its child elements from their corresponding servers, it reaches a 'DOM rendered' state. At this point, the document's elements can be manipulated: they can be edited or removed using the Document Object Model (DOM) API, and new elements can also be added through the same API. Notably,

in the real browsers, HTML elements can also be altered during rendering, we do not consider this case in our model.

- Every browsing context maintains a session history that keeps track of the documents that it has visited and is currently visiting.
- A browsing context can open multiple nested child browsing contexts representing the behaviour that a browser document contains iframe documents inside. A browsing context without an opener is a top level browsing context.
- A boolean secure context value is assigned for each browsing context. Certain browser APIs and features require stronger security measures and they are only accessible for browsing contexts that are in secure contexts. For a website to be in secure context, its code must be delivered through an authenticated and encrypted channel i.e. over TLS. In the model, we abstract away the channel authentication and encryption, and we assume that the browsing context is a secure context when a valid absolute URL [194] with HTTPS scheme is visited in its current document. However, for ultimately being considered in secure context by the browser, the entire opener chain is involved in the decision process. We detail other requirements that a browsing context to ultimately be in a secure context in Section 5.6.2.
- The iframes can be sandboxed in the browsers to isolate the iframe content. Hence, a browsing context that is associated with an iframe can be sandboxed in our model. We introduce a boolean variable called 'isSandboxed' for those browsing contexts to indicate whether they are sandboxed.

In this section we only describe the Same-Origin Policy mechanism in our detailed model M_{rfc}. The Cross-Origin Resource Sharing Policy and the Cross-Origin Isolation State are presented in Section 5.7. In addition, the absolute URLs and local URLs are detailed in Section 5.6.1.

We integrate our model with the Same-Origin Policy which is a security mechanism inherent to browsers to restrict data access between different origins to ensure data isolation. In the model, this policy is constructed based on an integrity property that prevents unauthorised information exchange between distinct origins. In simpler terms, this property functions as an access control mechanism between browsing contexts that intend to initiate events on each other's document. The integrity property is associated with the Function signature in the Call interface and it regulates the event calls to the browsing contexts. The 'party' relation in the Function signature identifies the browsing context initiating the event and the 'bc' relation identifies the browsing context on which the transition function will be applied in response to the event. If the 'party' relation is different from the 'bc' relation, then the browsing contexts in each relation must be in the same origin. Additionally, the browsing context in the 'party' relation should hold a reference to the browsing context in the 'bc' relation e.g. a self-opened popup window or embedded iframe.

In browsers, origins mainly fall into three categories: tuple origins, opaque origins and special scheme internal origins. Tuple origins are the most prevalent. They are derived from scheme, hostname and port number in the URLs. They represent different web

server addresses on the internet. Opaque origins are unique origins generated in specific scenarios e.g. sandboxed iframes and the local scheme URLs e.g. about and data URLs. In addition, browsers may have special internal origins for specific functionalities in the browsers e.g. startup or error page origins like 'safari-resource://'. We represent all special internal origins with StartupOrigin in our models. This also makes it feasible to apply the same model across different browsers e.g. Chrome, Firefox.

The following is a breakdown of the Same-Origin Policy's restrictions across different origins:

- We prevent StartupOrigin from accessing the document of any browsing context, because they are only placeholders for the error documents or the browsing context that is in the initial state and lacking a document yet.
- A browsing context that has an opaque origin cannot be accessed by any other browsing context. Opaque origins are serialised into 'null' in the browser. However, the browser assigns a unique opaque origin to each document even if the documents have the same URL. Hence every opaque origin is a different origin and access is prevented. An exception applies to about URLs (when they are not sandboxed), they can be accessed by any browsing context sharing the same origin this includes those with opaque origins.
- Tuple origins are only accessible by which browsing contexts that have the same tuple origins, namely the scheme, host and port numbers must align.
- There is a unique case called the 'BlankOrigin' which is not part of the RFC specifications. It is only available in our Safari v13.04 model M_{cam}. It is assigned to a blob URL whose creator origin is an opaque origin or a blank origin when it is opened in a top-level window in some scenarios that we elaborate on in Section 5.8. A blank origin is a blank scheme, blank hostname and blank port number which serialises into '://'. Blank origins can only be accessible by the browsing contexts that have the same blank origin.

5.6 Camera Access Model

This specialised sub-model M_{cam} is derived from the detailed browser model described in Section 5.5. As mentioned before, it mainly incorporates various types of URLs, browsing contexts, documents, iframes, sandboxing, secure contexts, origins, history, web servers and DNS servers, and it integrates the Same-Origin Policy in the model. This model is utilised to detect vulnerabilities involving unauthorised access to users' media devices e.g. camera via the browser. The model is further refined to focus on specific functionalities that are required to access the media devices such as URL parsing, origin assignment, secure context assignment and media device access control.

Browsers need to parse strings into URLs and due to the complex syntax variations of URLs, browsers can make mistakes in this process. Correct URL parsing is one of the most important tasks in browsers, primarily because origins are determined based on accurately interpreted URLs. Incorrect origin assignment can lead to security and privacy violations. Furthermore, secure contexts are assigned to local resources or

external resources that are delivered confidentially i.e. via TLS. Additionally, errors in URL parsing not only affect the correct origin assignment but also secure context assignment especially since the evaluation of local resources and those delivered through TLS depends on the correct interpretation of URLs.

In the model, for the URL parsing concept, we aim to determine how browsers handle different URL formats, including their schemes, delimiters, hosts, ports, paths, queries and fragments, as well as how they assign origins for URLs. In the secure context assignment concept, we aim to analyse how browsers establish and maintain secure contexts for different browsing contexts in different scenarios e.g. when they are in sandboxed iframe or when they are opened in a popup window. Then, we model the correct behaviours and the vulnerabilities associated with these concepts in Safari browser (v13.04) i.e. how it handles error URLs with malformed components and how it assigns origins and secure contexts to the browsing contexts that attempt to navigate such URLs. We select the Safari v13.04 to demonstrate that our model can detect a media devices attack taking place on this and preceding versions, as identified by Ryan Pickren [129].

We aim to evaluate whether entities, such as JavaScript elements or browser tabs, that do not qualify the required permissions and secure context requirements, can still gain access to users' camera by exploiting vulnerabilities in URL parsing, origin assignment and secure context assignment. In the following two subsections, we present the URL manipulation and secure context assignment concepts in our model.

5.6.1 URL Manipulation

We firstly model the valid definition of URLs in RFC [17] specification. We only model the URLs in the following schemes, and we exclude other schemes that are not in this list e.g. File, JavaScript. Furthermore, we also remark that the origins for these URLs, we state in the following for each, are based on when they are navigated in a new tab i.e. they are not opened as a popup window or iframe in another tab or sandboxed, because the origin assignments in the browser are subject to these conditions.

- Absolute HTTP and HTTPS URLs: The browser and web servers communicate to deliver web content in these cases. The URL specifies the web address, that comprises the scheme, host name, and optional port number, path, query, and fragment components. The host name is used to search DNS servers to obtain the web server's address. The origin of an absolute URL is categorised as a 'TupleOrigin'. A tuple origin consists of the scheme, host name and port number. The port number is included if it is applicable in the URL.
- About URLs: Local URLs used internally to access browser specific features e.g. 'about:blank' opens a new blank document. The origin of an about URL is classified as 'OpaqueOrigin'. The opaque origins serialise into 'null' in the browsers.
- Data URLs: Local URLs that are not used to point to any internal or external resources. Instead, they embed data inside the URL. The URL contains the mime type of the data, encoding type and the actual data. Their origin is opaque origin.

- Blob URLs: Local URLs that point to data that is called Binary Large Objects (Blob) in the browser's memory. They can be created by JavaScripts. The URL includes the creator browsing context's serialised origin and a random identifier. The origin equals to the creator browsing context's origin in the URL. Safari v13.04 fails to assign the correct origin to top-level blob URLs whose creator origin is an OpaqueOrigin in some cases we explain in Section 5.8. It assigns a 'BlankOrigin' and we cover this behaviour in the model.

Subsequently, we cultivate URL manipulation techniques and their behaviours in the model. Browsers provide multiple methods in JavaScript realm to manipulate the URLs. Those methods are useful to create dynamic web applications e.g. Single Page Applications (SPA). Using those methods, the developers can modify the URL and change the content of the document without a page refresh or new web request to the server. However, it is crucial to implement these methods correctly to avoid potential security vulnerabilities. Improper implementation of URL manipulation methods can lead to security risks. For instance, a browser assigning an incorrect origin to some content might result in attacks like cross-site scripting (XSS), information leakage and script injection. In the following, we provide the URL manipulation interfaces that we model. Each of these interfaces is represented as events in the model that result into transitions between states e.g. the URL of the browsing context changes, the old document gets discarded, a new history gets appended to the list and others :

- Navigate : Refers to the action where the user enters a URL in the address bar to visit a web page in the browser or clicks to a link in the current web page. A new browsing context is created and appended to the browser. The origin of the new browsing context is computed from the visited URL.
- Popup : A top-level window or an iframe opens a new tab in the browser and navigates to a specified URL. A new browsing context is created and appended to the browser. The origin of the new browsing context is computed from the visited URL, except for about URLs. About URLs opened in a popup inherit the origin of the opener.
- History.pushState : Inserts a new record to the browsing context's history list and changes the URL displayed in the address bar in the browser. It includes a URL to be appended to the list. This URL in the new record must be in the same origin as the original URL of the browsing context. Additionally, the new URL can also be a relative path. In this case, the relative path is either added to the original URL or it replaces the path in the original URL if a path already exists. As Safari v13.04 fails to parse the blob URLs and to assign the correct origins to them, attackers can use the history.pushState to replace the blob URL. This is detailed in Section 5.8.
- Location.replace : Allows to replace the current web page with the one in the specified URL. The new URL is not appended to the browsing context's history list, hence the user cannot use the back button to navigate back to it. In addition, 'Location' object that represents the current URL of the web page points to the new URL. However, the Same-Origin Policy restricts the cross origin replace requests

to the 'Location' object. The JavaScript object that calls the 'Location.replace' method on the web page must have the same origin as the current page that will be replaced. For instance, in the popup context, if an opener browser tab attempts to replace the location of the opening tab, then the document of the opener tab and the document of the opening tab must have the same origin.

- Document.write : Permits to directly replace the content inside a document with the string specified in the method. It is currently discouraged to use by RFC [82] because it may affect the HTML parser in some cases and result into DOMException errors. When the Document.write operation is applied by another entity, e.g. a top-level window, an iframe or a popup, then it also replaces the URL and the origin of the document with the URL and origin of the applying entity. In general, the Same-Origin Policy applies in this and the other interface contexts. The document and the applying entity must have the same origin.
- Add sandbox flag : Being in a sandboxed iframe will affect the origin of the browsing context. When an iframe is sandboxed, it will always have an opaque origin, regardless of the URL it contains. Additionally, a sandbox flag can dynamically be assigned to an iframe after it is rendered in the document. However, the sandbox flag will only take effect after the iframe navigates to a new page. Once the iframe loads the new page, the new document inside the iframe will have the opaque origin. In cases where the new page navigation attempt in the iframe fails due to a reason such as the Content Security Policy (CSP) [173] or XFrameOptions [137] preventing the content from being presented in an iframe, the iframe document will be replaced with an error document, its URL will be the 'StartupURL' and the origin will become 'OpaqueOrigin'. However, there is a deviation in how Safari v13.04 handles this situation. Instead of properly updating the iframe's document and origin, the iframe document remains in the previous state, and the sandbox flag is still applied to the iframe. We cover this behaviour in the M_{cam} model.

5.6.2 Secure Context Assignment

In the model, we assign a boolean 'secure context' value for each browsing context. This value represents whether a particular browsing context is considered to be in a secure context or not. A context is considered secure when data was delivered with integrity guarantees i.e. delivered over TLS which provides a confidential and authenticated channel. To simplify the model, we abstract away the details of the TLS infrastructure. Instead, we assume for a browsing context that is associated with a top-level window and navigates a valid absolute URL with the HTTPS scheme, then it is in a secure context [193]. For other URL schemes, e.g. HTTP, Data, Blob, except the About scheme, we assume that the browsing context is in a non-secure context. This means that when a browsing context navigates to a URL with these schemes, certain restrictions may apply accordingly to access browser APIs.

The secure context algorithm in the specification [193], includes ancestral checks to determine the secure context of a browsing context in an iframe. Browsing contexts in

iframes are considered in secure context when their documents are delivered over TLS and they are embedded in a document that is also in a secure context. For instance, an iframe document that has a URL with HTTPS scheme is ultimately considered non-secure context when one of its chains of ancestors is in a non-secure context. In the model, for such a scenario, we assign the browsing context in the iframe as secure context, but when we evaluate the browsing context's secure context value for accessing to the browser APIs, we compute the 'ultimate secure context' value that shows the real secure context value of the browsing context. This simplifies the model as some browser (e.g. Safari v13.04) whose behaviours we modelled performs an erroneous behaviour with secure context assignment. The ultimate secure context value is computed with 'decideUltimateSecureContext[bc]' function in the model and the secure context value in the execution trace does not represent the ultimate secure context value.

Furthermore, URLs with the About scheme inherit the secure context value of the opener, when they are opened in an iframe or in a popup window. It is crucial to assign the correct secure context value of a popup window with About scheme when it is opened by an iframe. The secure context values of the ancestor chain of the iframe must be taken into consideration when the secure context value is copied to the popup window that has the About scheme. Safari v13.04 deviates from the expected behaviour in the secure context assignment procedure in this scenario and we cover this behaviour in the model.

5.7 Cross-Origin Isolation State

Processors apply speculative executions by guessing future paths in the software code and executing the instructions in those paths beforehand to improve performance. However, the speculative executions can leave traces in the cache. In Spectre side-channel attacks [89], the attackers can infer knowledge from these traces by for instance measuring the time it takes to access certain data in the memory. In the context of the browser, this could mean a JavaScript code in a web page in one browser tab can gain knowledge about other tabs opened in the browser by accessing memory locations that other tabs have which bypasses the Same-Origin Policy and results into data leaks. Browsers like Chrome introduced site isolation that assigns a different process for each browsing context to avoid sharing processes with attackers. However, process isolation is not a complete safeguard to prevent side-channel attacks like Spectre as they facilitate the underlying hardware that includes the CPU cache which is shared among processes. Hence, in addition, browsers disable powerful APIs such as SharedArrayBuffer (SAB) and performance.now() which provide high precision timers and are available in the JavaScript realm to be used by the attackers.

The browsing contexts that are in Cross-Origin Isolation State [117] are exceptions for those APIs. Cross-Origin Isolation State is a state-of-the-art security feature extending the principles set by the Same-Origin Policy and it re-enables the powerful APIs for the browsing contexts that are in this state. There are two features that browsing contexts need to opt in by sending the response headers presented below to achieve this state.

```
Cross-Origin-Opener-Policy : same-origin
```

```
Cross-Origin-Embedder-Policy : require-corp
```

Listing 5.4: Cross-Origin Isolation State headers

When a browsing context opens a new popup window, it retains a reference to this window's browsing context which means they can interact with each other using the reference via the DOM API. This can present a security risk. For instance, without the user noticing, the opening website can redirect the opener website to another website that looks like the original website but is malicious instead. Setting the Cross-Origin-Opener-Policy (COOP) [176] to 'same origin' prevents the opening window and the opener window from holding references to each other if they are not in the same origin. This measure effectively reduces the attack surface by limiting the cross-origin interactions e.g. when a Spectre attack takes place.

Furthermore, websites frequently embed third-party contents in their applications. These third-party contents share the same browser process since they reside within the same browsing context. Setting the Cross-Origin Embedder Policy (COEP) [180] to 'require-corp' will require the cross-origin content to explicitly permit being embedded in a cross-origin document. In other words, with COEP set to 'require-corp', the browser will block a cross-origin content to be embedded in a document if it did not allow to be embedded using Cross-Origin Resource Policy (CORP) [181] or Cross-Origin Resource Sharing (CORS) [179] policies. This is mainly for protecting the third-party content from the main document. In this way, the main document cannot attack the third-party content that did not opt-in to the COEP feature. This measure minimises the attack surface. The CORP response header allows to set the origins that can access the resource. Possible directives are 'same origin', 'same site' or 'cross-origin'. On the other hand, CORS is a policy defined by the Access-Control-Allow-Origin response header [189]. The value for this response header represents the permitted origins.

5.7.1 Side Channel Access Model

The second specialised sub-model M_{sch} is also derived from the detailed browser model, as detailed in Section 5.5. This M_{sch} model includes key browser components like browsing contexts, browsing context groups, documents, iframes, HTML elements (JavaScript and non-active elements), DOM API, origins, web servers, DNS servers and absolute URLs. Moreover, it incorporates the Same-Origin Policy, Cross-Origin Resource Sharing Policy and Cross-Origin Isolation State. In this model, some powerful JavaScript APIs (e.g. SharedArrayBuffer) are only accessible when a browsing context is in Cross-Origin Isolation State in the browser. However, a browsing context can share data with or even allow scripting to other browsing contexts. In this model, we explore whether the powerful APIs like SharedArrayBuffer can be shared between cross-origin browsing contexts.

In the refined M_{sch} model, we exclude various local URL definitions present in the original model and we only consider Absolute URLs that require HTTP requests to and responses from servers. The model is constructed upon two principal HTTP requests : main requests and resource requests for content embedded within the main

document. Additionally, the model incorporates the 'popup' operation i.e. opening a new window from the main website. The main request is dispatched to a server identified by its domain name in the URL. The response to this primary request provides a raw HTML document containing only URLs pointing to embedded resources at this stage. Additionally, the response includes the response headers that indicate the policies and their associated directives e.g. 'COOP=same-origin' and 'COEP=require-corp' that the server opted for. When sending resource requests or opening a new popup window from the main website, the browser considers these policies. For instance, with a policy like 'COEP=require-corp', all cross-origin resource requests will inspect the response for CORP and CORS response headers. Subsequently, the browser issues iterative resource requests for each embedded resource URL listed in the main HTML document to their corresponding servers. In response, servers provide either a JavaScript or a non-active element, accompanied by specific response headers, e.g. CORP or CORS indicating which origins are authorised to embed this element. Taking these policies into account, the browser then decides whether to incorporate the element into the document's 'elements' list if it is deemed compliant or into the 'blocked' list otherwise.

Drawing from the RFC specification [190], we also introduce 'browsing context groups' in our model. A browsing context group collects a list of browsing contexts that are directly accessible from each other. For instance, a typical browsing context group might comprise a primary browsing context and its associated nested child browsing contexts (e.g. iframes) and the browsing contexts that it holds references to (e.g. popup windows). When a browsing context initiates a new popup, the associated COOP policy of the browsing context's document is considered by the popup operation. Specifically, if 'COOP=same-origin' is set then cross-origin popups are not added to the same browsing context group with the initiating browsing context.

Furthermore, we introduce a 'SharedArrayBufferAccess' event in this model to permit scripts to access the SAB, which provides a shared memory among same-origin documents [146]. For a website to access SAB, it must operate within a secure context and be in a Cross-Origin Isolated State. In this model, we abstract away the scheme part of URLs. Hence, we simply assume all websites are within a secure context and only cross-origin isolated state is required to access the SAB. We incorporate this functionality to explore whether a script can access a cross-origin document's SAB. For the 'SharedArrayBufferAccess' event, our model imposes the following constraints:

1. script is legitimate: within the model's execution trace, the script should be an element of a document, which in turn should be the active document of a browsing context. This browsing context should be part of a browsing context group present in the 'bcgs' relation of the state,
2. the document is legitimate: the document must be a recognised as part of the state, following the chain relation as outlined above,
3. the document has access to SAB: the document is in cross-origin isolated state,
4. the origin of the script's associated document and the origin of the document itself must be identical.

5.8 Security Analysis

Proper implementation of URL parsing, origin assignment and secure context assignment are crucial aspects for preserving the security and privacy in web browsers. Fundamental browser security features such as the Same-Origin Policy rely on correctly determining the origin of a web document which further relies on correct URL parsing. The Same-Origin Policy regulates interactions between a resource in one origin to a resource in another origin. An incorrect origin assignment can inadvertently undermine this policy, leading to potential unauthorised actions on a resource or undesired information leak. Moreover, in some browsers, certain sensitive APIs, for instance, those granting access to a user's media devices require origin-based permissions explicitly defined by the user beforehand. An incorrect origin assignment can therefore violate users' privacy. On the other hand, a secure context ensures the sensitive APIs are only accessible to websites that have proven some level of trustworthiness i.e. only the websites that are accessed over a secure communication, preventing man-in-the-middle attacks, can access to camera or geolocation of the user. Misassignment of secure context to an untrustworthy website can also jeopardise the privacy of the user.

As browsers continue to evolve incorporating even more complex functionalities, ensuring data privacy and integrity becomes increasingly challenging. With new cases being added constantly, browser testing becomes more complicated every day. The vast expanse of the test case scenarios can be daunting to navigate and it is becoming evident that covering all edge cases in traditional testing methodologies might fall short. Consequently, employing formal methods, which offer a systematic and comprehensive approach to understanding and verifying countless test case scenarios, can assist in identifying security and privacy vulnerabilities in the browsers.

In Section 5.5, we present our formal browser model that deeply examines the implications of URL manipulation, origin assignment and secure context assignment. In this section, we specifically explore their collective impact on accessing sensitive APIs that control accesses to users' media devices e.g. camera and require origin-based permissions defined by the user and secure context for access. Our objective is to uncover methods that could mislead the browser into mistakenly granting access to malicious websites by presenting them as if they are from permitted sources within a secure context using the URL parsing, origin assignment and secure context assignment vulnerabilities. We show that vulnerabilities that seem minor can actually lead to significant security and privacy breaches in the browser.

In our formal analysis, we find variations of a previously known camera attack by Ryan Pickren [129] on Safari v13.04 and our model even finds a shorter version of the attack.

5.8.1 Ryan Pickren's Webcam Attack

We selected Ryan Pickren's webcam attack [129] on Safari v13.04 as our case study due to its complexity. It requires multiple steps and many browser functionalities to be modelled. As we show the attack plan in the following, detecting such attacks through conventional testing can be hard. On the other hand, formal methods permit an exhaustive search for potential vulnerabilities, at least up to a certain boundary.

In 2019, Ryan Pickren showed that Safari v13.04 browser can be hacked to gain unauthorised access to the webcam by exploiting the URL parsing, origin assignment and secure context assignment vulnerabilities. We elaborate on the attack plan in the following :

- The attacker is a web attacker who operates a potentially legitimate website and seeks to exploit vulnerabilities in Safari browser to gain unauthorised access to the user's webcam by leveraging browser APIs, e.g. MediaDevices Web API [109]. Browsers, by design, grant access to specific APIs only within a secure context and to trusted websites that are explicitly set by the users. If the attacker's website can masquerade as a trusted entity and bypass the secure context checks, it can exploit these permissions to access an unsuspecting user's camera without a proper consent.
- Apple's native applications such as Safari have default access to the camera without requiring explicit permission from the operating system. In Safari, when the website attempts to use the camera, the users are prompted with a bar asking users if they grant the camera access to that website. On the other hand, users have the flexibility to provide persistent camera access to specific websites through Safari's settings. This means that users can designate trusted websites to access their camera without being prompted for permission each time. If an attacker manages to trick Safari into believing that a malicious website is among the trusted websites, they can access to the camera without direct user consent.
- The attack, as depicted in Figure 5.1, hinges on the nuances of tracking websites on the Safari. Safari v13.04 does not use origins to monitor 'currently open websites', instead it employs a generic URL parsing on all open windows that removes the URL's scheme and scheme delimiter part and strips 'www.' from the hostname. That leads to malformed URLs such as 'blob://example.com' to be interpreted as legitimate 'example.com'. If a user navigates such a URL directly, Safari will display an error document and script execution will be halted, preventing an attack. However, if an attacker begins with a valid webpage URL and later finds a way to alter it, script execution is not interrupted, opening the way for a potential attack.
- The goal of the first part of the attack is to create a URL that will be interpreted as a trusted URL e.g. example.com, by the genetic parser which considers the 'blob://example.com' and 'https://example.com' are the same websites. Concurrently, the aim is to produce a document that allows script execution. As outlined in the formal model Section 5.5, when Safari encounters a blob URL created by an opaque origin and navigated in a top-level window, it assigns a blank origin to this URL. Additionally, when it is opened in a popup window, it inherits the opener's origin. The attacker starts by navigating to a data URL. Subsequently, in the document encoded in this URL, he generates a blob whose creator origin is an opaque origin. Then, he replaces the present document with a document that includes the blob content using the location.replace API. As a result, the current document not only supports script execution but also possesses a blob URL with a blank origin. This part is presented with the state transitions from

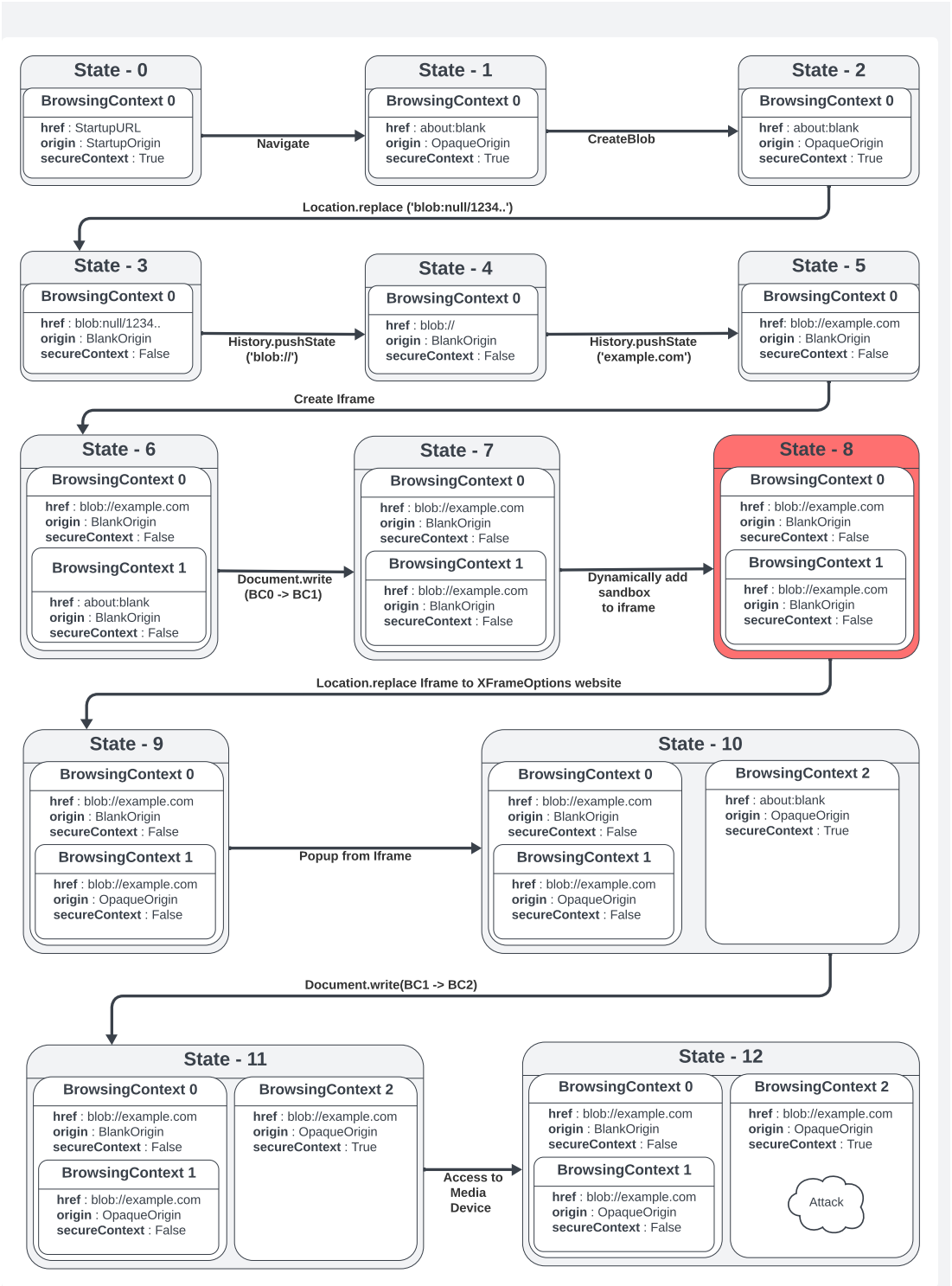

Figure 5.1: Ryan Pickren's Webcam Attack (*State-8* is absent in our model's attack trace, elaborated in Section 5.8.2)

State-0 to *State-3* in the Figure 5.1.

- To craft a URL that is recognised as trustworthy, the attacker uses the history.pushState API to modify the existing URL without triggering Safari to navigate to the actual URL. As detailed in the formal model section, the history.pushState API facilitates two main functions: it can replace the entire URL with a provided URL in the parameters or simply replace the path of the current URL using the path supplied to the API. For the first function, the history.pushState API evaluates whether the existing URL holds the same origin as the URL that is used for modifying. Given that the current URL's origin is blank origin (encompassing a blank scheme, hostname and port number) that serialises only to the delimiters '://', an opportunity arises. The attacker can swap the current URL with 'blob://' whose origin is similarly interpreted as a blank origin. Subsequently, using the history.pushState API's second functionality, Safari can be deceived into believing that it is merely replacing the path when any sequence of characters that lacks the scheme delimiters provided to the API. For instance, when supplied with the 'example.com' parameter, the API will modify the URL as 'blob://example.com'. Since this URL is interpreted as 'example.com' by the Safari, the document will have direct access to the camera if it satisfies the secure context requirement. In the Figure 5.1, the transition from *State-3* to *State-5* illustrates the modifications in browsing contexts resulting from the application of history.pushState events.
- The objective of the second part of the attack is to have secure context for the existing document whose URL is derived from a local URL. As outlined in the formal model section, to attain a secure context, the website must be delivered in TLS. Additionally, a document in a secure context embedded within a non-secure context (e.g. within an iframe) will ultimately be regarded as non-secure. Hence, it is essential to consider the opener chain. However, Safari diverges from these standards by assigning secure context to any document whose origin is opaque origin and failing to check the opener chain when making secure context determinations. The attacker can leverage these vulnerabilities by opening a popup with the crafted URL and setting its origin as an opaque origin within a JavaScript enabled document.
- As highlighted in the formal model section, sandboxed iframes always have opaque origins. Leveraging this can help to establish a secure context for the document. Moreover, the Document.write operation allows for the modification of the target document's content and this operation propagates the URL to the targeted document. As depicted in the Figure 5.1, in deploying an attack, the sequence will include the steps in the following:

  1. The attacker begins by creating a standard iframe within the existing document, initialising it with an about URL. This is depicted in the transition from *State-5* to *State-6*.
  2. Subsequently, the Document.write operation is executed on the iframe, copying the malicious camera exploit code into it. As an outcome, the iframe

inherits the crafted URL from the parent document (from *State-6* to *State-7* transition).

3. To escalate the exploit, the attacker dynamically appends a sandbox attribute to this iframe, aiming an opaque origin within the iframe (from *State-7* to *State-8* transition). However, as emphasised in the formal model section, sandbox attributes appended dynamically only become active after a new website is navigated in this iframe. A challenge here is that if a fresh website is navigated within the iframe, any JavaScript contained within the iframe will be purged. Hence, a careful orchestration is required to ensure the malicious code persists in the iframe. To address this, the attacker lets the browser to navigate a website with an X-Frame-Options response header set to deny to be embedded in an iframe. Upon this attempt, the navigation fails. However, it ensures that the iframe adopts a sandboxed state with an opaque origin, while the JavaScript remains intact inside the iframe (from *State-8* to *State-9*).
4. The last step is to open a popup with about URL (from *State-9* to *State-10*) from this iframe and apply Document.write() operation to the popup document (from *State-10* to *State-11*). Consequently, this new popup will have the crafted URL equipped with an opaque origin and a secure context. Within this environment, the attacker can seamlessly exploit the camera API (from *State-11* to *State-12*).

5.8.2 Shorter Version of Ryan Pickren's Webcam Attack

Upon further analysis through our formal model checking, we discovered variations of Ryan Pickren's webcam attack. Central to our analysis was the evaluation of a specific security property, formulated as: "Is it feasible for an attacker to gain access to a user's media devices, through a browsing context whose URL does not conform to an Absolute URL?"

```
assert cameraAttack {
    always {
        no bc : BrowsingContext | some m : Media {
            bc.currentDoc.src !in AbsoluteUrl and
            m in bc.accesses
        }
    }
}
check cameraAttack for 1 but 6 Url, 3 Path, 3 Domain,
3 BrowsingContext, 3 Document, 4 Endpoint, 6 Origin,
7 History, 14 Function, 14 steps
```

Listing 5.5: Alloy Assertion Property for Verifying Media Access

In Section 5.8.1, we discussed Ryan Pickren's Webcam Attack and how Safari v13.04 has a parsing flaw concerning URLs. This flaw allows for malformed URLs, such as 'blob://example.com', which isn't a true Absolute URL, to be misinterpreted by Safari as a legitimate Absolute URL like 'example.com'. If a user has previously

marked the 'example.com' as a trusted site for media device access, then the deceptive 'blob://example.com' URL would also gain access to the media devices. Hence, in our security property, our investigation focuses on assessing whether a URL containing a trusted domain (e.g. 'example.com'), but not being a genuine Absolute URL, can indeed access media devices.

In the following, we present the constraint in our model that must be true for all media accesses. Recall that the 'bc' relation in events ('Access2Media' in this case) indicates the browsing context that is attempting media access. The event also contains a 'media' relation representing the target media which further has 'permitted' relation listing the domains authorised for access. Therefore, a browsing context can only access the media device if:

1. the browsing context is among the currently open browser tabs,
2. the URL of the browsing context is considered a ValidUrl,
3. the origin of the browsing context is not a StartupOrigin,
4. the domain name present in the URL is listed in the media's set of permitted domains,
5. the browsing context is in a secure context.

```
fact mediaAccess {
    always (
        all f : Access2Media |
        let nbc = f.bc | let m = f.media |
        let dom = find_domain[nbc.currentDoc.src] |

            f.canAccess = nbc implies (

                nbc in Browser.bcs and
                nbc.currentDoc.src in ValidUrl and
                nbc.origin !in StartupOrigin and
                some dom and dom in m.permitted and
                decideUltimateSecureContext[nbc] = True
            )
    )
}
```

Listing 5.6: Alloy Property for Restricting Media Access

With our analysis, we also undercover a deviation from the initial understanding of iframe origin assignments in the Ryan Pickren attack. In the original attack, it was believed that to achieve an opaque origin within an iframe, it was essential to dynamically append a sandbox attribute to the iframe to get an opaque origin in the end (transition from *State-7* to *State-8* in the Figure 5.1). However, our model has brought forward an easier approach, potentially simplifying the attacker's task. Our findings indicate that navigating the iframe to a website with an X-Frame-Options header set to deny inherently results in the iframe obtaining an opaque origin, irrespective of the presence or absence of a sandbox attribute. This revelation means that the additional step of

sandboxing the iframe becomes redundant. The iframe, once denied navigation due to the X-Frame-Options, will naturally acquire the desired opaque origin, also ensuring that the malicious JavaScript remains operative within the iframe. Hence, the transition from *State-7* to *State-8* is absent in our observed attack trace.

By understanding and leveraging this behaviour, attackers can potentially reduce the steps and complexity involved in their exploit, making their attacks more efficient. It also underscores the importance of consistent formal model checking in identifying nuances and deviations that might not be immediately obvious during traditional software testing.

5.8.3 Security Analysis of Cross-Origin Isolation State

As outlined in the Section 5.7, we extended our formal model to encompass the Cross-Origin Isolation State, a vital security feature designed to reinforce the isolation of resources (e.g. SAB) between origins. The intention behind this modelling was to ascertain whether there exist any vulnerabilities or unexpected behaviours that could potentially be exploited i.e. a script can access SAB through a cross-origin document. Upon rigorous analysis, our model did not identify any counterexamples. The absence of counterexamples supports that the existing design of the Cross-Origin Isolation State effectively maintain the expected isolation of resources between origins, validating the robustness of the feature against potential attacks. It is also important to note that this absence of counterexamples is confined to the scope of our defined model. Counterexamples may exist in broader scopes that we did not explore. The following is the security property that we check in our analysis which by default analysed for 10 steps.

```
assert sharedBufferAccess {
  always {
    no script : Script | some buffer : SharedArrayBuffer |
    let crossOriginDoc = buffer.~sharedbuffer |
    let scriptDoc = script.context |
      legitScript[script] and
      buffer in script.sabAccess and
      crossOriginDoc.src.origin != scriptDoc.src.origin
  }
}
check sharedBufferAccess for 4 but 2 BrowsingContextGroup,
3 BrowsingContext, 3 Server
```

Listing 5.7: Alloy Assertion Property to Verify Cross-Origin SharedArrayBufferAccess

5.9 Test Case Generation and Simulation

We further analyse the counterexamples generated by the model checking of the properties using our test case generator and simulator. These tools collectively assess the feasibility of exploiting the potential attack path highlighted in the counterexamples within an actual browser environment. To automate the process, we implemented a Java

application framework that not only executes the model checking but also interprets the model checker's output. The application then formulates a test case to replicate the same configurations and actions within the browser, subsequently simulating the potential attack scenario. This automation provides a comprehensive analysis of the defined properties and verifies the viability of the detected attack within the browser's environment. The framework is depicted in Figure 5.2.

Kodkod [160], a boolean satisfiability (SAT) based constraint solver, is Alloy's backend. In our application, we utilise the Kodkod library [6] to apply model checking to our model for finding solutions (counterexamples) to our security properties. When a counterexample emerges from Kodkod, it is presented as a state transition trace. This trace is a sequence of states each influenced by transition functions in response to specific events. This representation offers insights into the progression of potential attacks, showing how state components (known as signatures in the model) and their relations evolve in each step.

We implement a parser in our application that parses the counterexample output and breaks down the output into its constituent states. Following this division, it generates the same representation of elements and their relations within the Java application for each distinct state. Drawing from principles of object-oriented programming, elements are represented as objects whereas their relations are expressed as attributes. Specifically, we define a 'State' object within our application encapsulating all objects and their associated attributes for a given state, applicable across all steps. This approach results in the application creating a series of 'ModelState' objects, ranging from the initial state to the final state.

Following that, we find the differences between the ModelState objects utilising the Javers [86] tool in Java. This tool computes the differences between consecutive ModelState objects and highlights the additions, deletions and modifications of objects and collections. Then, we store any differences between consecutive states along with the event and the specific browsing contexts (the 'rootBc', 'bc' and 'party' relations as explained in the formal model Section 5.5) in an object called ModelDiffState. We keep the ModelDiffStates in an ordered list.

To craft a test case, we firstly take the initial state represented by the first ModelState object in the state collection. Subsequently, we reproduce these initial conditions within the browser using the BrowserStack test automation tool [28]. BrowserStack is a platform for web applications that allows cross-browser compatibility testing on various browsers, operating systems and devices. It further streamlines the testing process by automating it using the Selenium WebDriver scripts [145]. Using the Selenium WebDriver scripts on BrowserStack, we automate the counterexample validation process in the browser.

As highlighted in the formal model Section 5.5, all traces must start with at least one tab open in the browser. Hence, the initial state consists of a single tab (a browsing context) opened in the browser but not navigated to any documents yet. Moreover, the browsing context in the browser has other properties e.g. its URL, origin, secure context value, sandbox value and others which are extracted from the ModelState object.

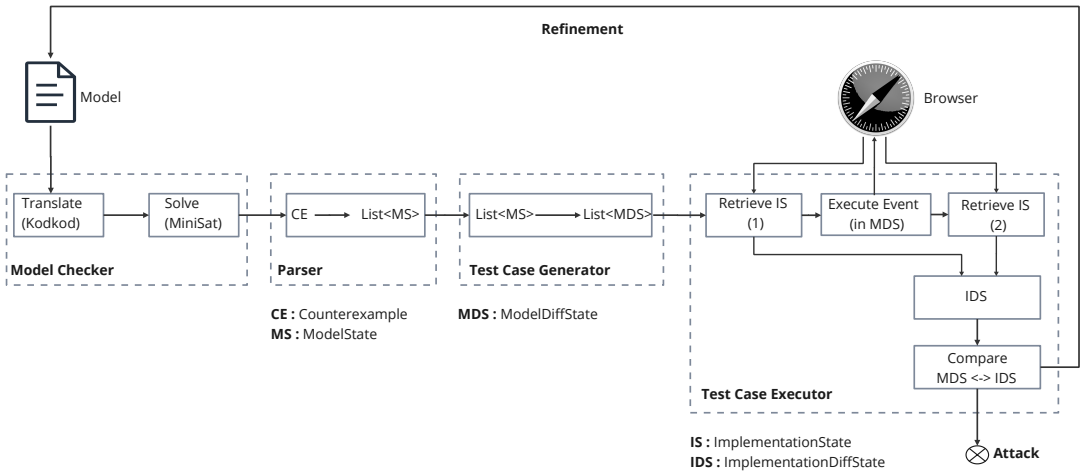

Figure 5.2: Application Framework Flowchart Diagram

Using BrowserStack and Selenium tools, we emulate the initial state in the browser by launching a new tab without navigating to a URL. Each state includes an event leading to the transition from the current state to the succeeding state. We maintain a list of Selenium scripts that can apply various events to the browser tabs. In addition, the 'rootBc', 'bc' and 'party' relations in the ModelDiffState are used to navigate to specific browser tabs or frames and to apply the events based on these relations.

Upon replicating the initial state to the browser, we take the implementation state of the browser and keep it in an 'ImplementationState' object. We have Selenium scripts that take the properties of the affected browser tabs i.e. the origins, the secure context values are captured from the browser and stored in the ImplementationState object. Next, the event associated with the initial state is executed on the browser. Subsequently, we take a second implementation state to capture the new properties of the browser tabs after the event is applied. Using Javers tool, we compare these implementation states and keep them in an 'ImplementationDiffState' object. Then, we compare the ModelDiffState explained above (that shows the changes on the model state) and ImplementationDiffState (that shows the changes on the implementation state). Any inconsistency between these objects indicates a spurious counterexample, prompting us to return to our model for further refinement manually and restart the model checking procedure. If the changes on the model state align with the changes on the implementation state, we continue with the succeeding state, applying the same verification process. This process continues until a counterexample actually realises in the browser e.g. a browsing context gaining access to the camera.

5.10 Evaluation

Our models (M_{rfc}, M_{cam} and M_{sch}) and the framework implementation are available in [60]. In this section, we examine the efficiency of our framework, focusing on model

checking and test case validation. We benchmark our system based on three primary criteria: abstraction level in the model, efficiency of both model checking and the framework, and correctness. The level of abstraction metric evaluates the degree to which our model abstracts the real-world browser behaviour. It should be balanced that the model should not be overly abstract, losing important details of the browsers, nor excessively detailed, becoming impractical for computational analysis. The efficiency metric measures the computational resources e.g. CPU cores and memory, and the duration needed for model checking and test case validation. For correctness, our main concern is whether our model accurately mirrors the real browser and the framework reliably executes the test cases on the browser as intended.

Level of Abstraction In our model implementation, the issue of finding the correct level of abstraction proved to be more straightforward to address, thanks in large part to our iterative approach. We initiated the process with a very simplified model and then systematically enriched it, incorporating more details and features until we reached an abstraction level that accurately reflected real-world browser behaviour. However, this was still not without its challenges. One notable hurdle was our initial too detailed definition of URLs in the model. We aimed for comprehensive representation and thus included numerous components like schemes, delimiters, hosts, ports, paths, queries and fragments for every type of URL. Although this granularity seemed promising for capturing a wide range of behaviours, the model checking process became noticeably slower as a result. Hence, we applied a reduction in the level of detail by abstracting away the components like delimiters, ports, queries and fragments. Instead of using a granular representation, we employed abstract signatures to clearly define the special aspects of URL. The listings below serve as an example for this adjustment. In addition, while we have streamlined the URL definition in the main model to improve efficiency, we have retained the detailed URL version. This detailed version is available for researchers who are particularly interested in a more granular abstraction of URL in the Alloy framework.

```
sig Url{
    scheme : Scheme,
    sch_delim : lone Delimiter,
    auth_precede : lone Delimiter,
    authority : lone Authority,
    path_delim : lone Delimiter,
    path : lone Path,
    query_delim : lone Delimiter,
    query : lone Text,
    frag_delim : lone Delimiter,
    fragment : lone Text
}
sig BlobPath extends Path {
    creator_origin : lone (OpaqueOrigin + TupleOrigin),
    uuid_delim : lone Delimiter,
    uuid : lone UUID,
    media_type : lone MimeType,
    data : lone HtmlResource
}
```

Listing 5.8: A Detailed Blob URL Definition (old)

Listing 5.9: A Sufficiently Abstract Blob URL Definition (new)

```
abstract sig Url {
    scheme : lone Scheme
}

abstract sig ValidUrl extends Url {}

abstract sig BlobUrl extends ValidUrl {
    creator_origin : lone Origin
}{
    scheme = Blob
}
```

Table 5.1: Performance Evaluation Results for Case Studies

Case study	Cores	RAM	Time to first solution	Time to validation	#Lines of code
Camera Access (w/o WindowOpen)	1	30 GB	8 h, 43 min	32 seconds	2260
Camera Access (w/ WindowOpen)	1	30 GB	52 h, 58 min	32 seconds	2300
Side Channel Access (scope for 3)	1	30 GB	10 seconds	-	630
Side Channel Access (scope for 4)	1	30 GB	22 h, 24 min	-	630

Efficiency As discussed in the Methodology Section 5.4, Alloy offers three distinct decomposition strategies that can be parallel, batch or hybrid for efficient, scalable and flexible problem solving. The parallel and hybrid methods led to slower model checking. Thus, we used the batch strategy that facilitates only a single-core processor. We conducted our experiments on a high performance machine equipped with 64 cores an Intel(R) Xeon(R) CPU E5-4650L running at 2.60GHz, and 756GB of RAM. However, as we used batch strategy only one core of CPU is used in both case studies. We outline the performance results for each case study in the Table 5.1. The RAM values refer to the maximum heap size allocated for the Java Virtual Machine (JVM).

The duration of the model checking is denoted by the 'time to first solution", measured in hours. For the first case study, Camera Access model M_{cam}, based on our experiments, we determined that the 'WindowOpen' event is not required to achieve the attack. Hence, we conducted the model checking with and without the 'WindowOpen' event and presented the results in the Table 5.1. With 'WindowOpen' event enabled, the model checking takes approximately 53 hours to find the same attack, whereas without this event it just takes 8 hours and 43 minutes. These results suggest that excluding

certain unrelated events from the model can streamline the model checking process. The 'WindowOpen' event reflects the user action that opens a new window (a new browsing context), that is not related to existing browsing contexts in the browser i.e. it will not hold a reference to any other browsing context. Our model does not consider transitions that would relate the unrelated browsing contexts. This implies that the 'WindowOpen' event may not be pivotal in this attack scenario, therefore it can be omitted.

For the first case study, the time taken is justifiable given that 11 steps required to achieve the attack. The model checking time increases exponentially based on the number of steps and selected scope which determines the size of signatures in the counterexamples (e.g. 3 BrowsingContext, 6 History etc.). We did not reduce the number of steps (by abstracting some events away) to maintain a detailed model. However, we chose the minimum signature sizes that could still find the counterexample in the first case study. In addition, we employed restrictions (expressed as facts in Alloy) that the model checking must conform. Many of these restrictions arose from eliminating spurious counterexamples, reducing the number of traces that the model checker needs to evaluate. For instance, we defined facts stating that a document cannot be the 'currentDoc' of two distinct browsing contexts at the same time or a browsing context cannot be in its own 'nestedBcs'. In total, we established 18 such facts to eliminate spurious counterexamples in the Camera Access model.

To highlight the impact of the selected scope for signature sizes on the duration, the second case study, Side Channel Access M_{sch}, serves as a suitable example. Initially, we executed this model with a scope of 3 for all signatures for the 'sharedBufferAccess' assertion property, as detailed in the Listing 5.7. This configuration resulted in a 10 seconds of solution time. However, when we increase the scope for 4 signatures for all signatures with the exception of 2 BrowsingContextGroup, 3 BrowsingContext and 3 Server, the computation time took more than 22 hours. This contrast underscores the exponential growth in computation time with slight increase in the scope.

The time required to validate the first case study in the browser is again reasonable. This includes the steps of parsing the counterexample from the model checking output, setting up the same configurations in the framework, launching the browser via BrowserStack (accounting for internet-related delays), executing events on the browser and comparing the model state with the implementation state. In addition, to ensure accurate application of events to the browser, we incorporated Thread.sleep(time) operations prior to fetching the implementation state from the browser. Retrieving the implementation state prematurely, before the event is fully processed by the browser, might result in inaccurate behaviours. Nevertheless, adjusting the duration of Thread.sleep operations could potentially minimise the overall time.

Correctness Iterative refinement can be an indicator of the model's and the validation framework's accuracy. By deriving test cases from counterexamples and attempting to realise these counterexamples in the browser, one can validate the model against real-world scenarios. If a counterexample cannot be realised in the browser, refining the model ensures the model captures the actual behaviour of the system. This iterative process assists in progressively eliminating inaccuracies in the model and the framework,

thus enhancing the confidence in their correctness. In our experiments, the Camera Access model underwent about 10 refinement iterations. While we did not implement the validation framework for the Side Channel Access model, it was refined 3 times based on the counterexamples produced by the Alloy Analyser tool [7].

5.11 Related Work

In this section, we highlight the existing academic research focused on enhancing browser security through formal modelling and analysis. Concurrently, we position our research within this framework, explaining how our work in this chapter adds to the current literature. There is a considerable academic work on formally modelling and analysing browsers in recent years. A thorough review of formal methods for web security is available in [30], covering various topics within the realm of web security including formal browser models.

Grier et al. introduced an experimental browser named OP which later evolved to OP2, claiming enhanced security features [69]. They employ Maude [40] to formally validate components of OP2 and message passing APIs for the communication between those components. The verification was conducted under the assumption of attackers that can compromise these components and the message passing APIs. Their model encapsulates possible browser attack routes e.g. address bar spoofing. In such an attack, the attacker could take control of a web page subsystem component and exploit the message passing API, coercing the compromised component into transmitting false URL data to the user interface component. In addition, the Same-Origin Policy in the OP2 is designed to apply access control policies between browser components. Again in the presence of an attacker, they check whether the Same-Origin Policy can be violated by a compromised component. In contrast, the attackers in their model operate at the software level, while our model's attackers are all external websites, with the exception of trusted ones.

Sasse et al. developed a browser, named IBOS, that is designed with security in mind [140]. In the architecture of IBOS, multiple components operate concurrently and kernel of the browser is the sole entity that is trusted. All communication (message-passing) in the browser goes through the kernel and policies applied by the kernel on the messages, ensuring controlled flow of information. They carried out the formal verification with Maude and they formulated the same properties with OP2 in model checking. Notably, while both browser models, OP2 and IBOS, formulate the low-level browser components and message passings between them, our focus with respect to our case study is intentionally more specific to show the applicability of our modelling, automated verification, test case generation and validation approach on a concrete example in prevalently used standard browsers. We delve deep into the JavaScript APIs and the implications of their actions on the browsing contexts concerning the assignment of URL, origin and secure context.

Gross et al. provided a detailed browser model for reasoning about browser-based security protocols e.g. identity federation [70]. Their model delves deeper into HTTP requests and response transactions, capturing the nuances through request and response headers e.g. Referer, Accept_Language and others. In addition, they cover local storage,

Table 5.2: Comparison of Web Concepts in Formal Models

Concepts	$\mathbf{M}_{\text{rfc}}$ (Alloy)	**Featherweight** (Coq)	**Bauer et al.** (Pen & paper)	**WebSpec** (Coq)
Browsing environment				
Browsing Context (and BC.Group)	●	○	○	○
Browser tabs (windows)	●	●	●	○
HTML content				
Document	●	●	●	●
DOM API	●	●	●	●
HTML elements	●	●	●	●
HTML forms	○	●	○	●
JavaScript	●	●	●	●
Iframes	●	○	●	●
Isolation content				
Cookies	○	●	●	●
Secure Cookies	○	○	○	●
Secure contexts	●	○	○	○
Origin	●	●	●	●
Sandboxing	●	○	○	○
History API	●	○	●	○
Bookmarks	○	○	●	○
Web servers	●	○	●	○
DNS servers	●	○	○	○
URLs				
HTTP URLs	●	●	●	●
HTTPS URLs	●	○	○	●
About URLs	●	●	○	●
Data and Blob URLs	●	○	○	●
HTTP headers	○	●	●	●
Security policies				
SOP (DOM access + AJAX req.)	●	●	●	●
CORS	●	○	○	●
COIS	●	○	○	○
CSP	○	○	●	●
Noninterference	○	●	●	○
Browser APIs				
WindowOpen	●	●	●	○
HttpRequest	●	●	●	●
JavaScript APIs				
XMLHttpRequest	●	●	●	●
History.pushState + Location.rep.	●	○	○	○
Create blob	●	○	○	●
Popup	●	●	○	○
Document.write	●	○	○	○
Access to media	●	○	○	○
SharedArrayBuffer	●	○	○	○
PostMessage	○	○	●	●
User input/output	○	●	●	○
Service Workers	○	○	○	●
Browser extensions	○	○	●	○
Storage APIs (Cache+Local s.)	○	○	● (Local s.)	●

cache and history objects in their model. In our model, we only cover the history objects from this list in the context of the History API of JavaScript. We should also note that their model operates under the assumption that JavaScript is disabled in the browser. Given the integral role of JavaScript in modern web applications, as Bugliesi et al. also pointed out in the survey [30], this can be considered unrealistic in many real-world scenarios.

Bohannon et al. proposed a formal browser model called Featherweight Firefox in Ocaml [23] and Bohannon himself later transcribed a limited and updated version of it in Coq [155] in his PhD thesis [22]. The original model formulates multiple windows in the browser (excluding iframes), the HTTP protocol, about URLs, HTTP request/response, cookies, HTML elements e.g. div, img and script (inline or external), scripting functionalities like eval, user actions e.g. opening a window and clicking on a link, AJAX requests via XMLHttpRequest, DOM manipulations and others. In comparison, our model provides a more comprehensive list of JavaScript functionalities e.g. opening popup windows, creating iframes, creating sandboxed iframes, applying document.write operation, creating Blobs in the browser, managing browsing history, secure contexts, etc. We present a comparison of web concepts across various browser models, including our own model M_{rfc}, in the Table 5.2.

Furthermore, there are other applications that utilise the Featherweight Firefox framework as a basis. For instance, Bielova et al. augmented the original framework, focusing on exploring non-interference policies for web browsers which act as alternatives or supplements to the traditional Same-Origin Policy [20]. The Same-Origin Policy isn't foolproof against scenarios where, for instance, website-specific JavaScript might covertly leak users' private information back to its originating website. Bielova et al. implemented their non-interference policies on the Featherweight Firefox platform as it provides a detailed user input and output events. Similarly, Bugliesi et al. used the Coq model to validate non-interference [31]. Their research investigated how the HttpOnly and secure cookie attributes could mitigate the unintentional exposure of cookies, used for web session identification. Although our foundational model does not encompass user input events e.g. user inputs a secret text to a document, it is designed with extensibility in mind. User inputs can easily be incorporated in our model by describing the behaviours of the components after the user input event e.g. how those inputs are validated, processed, where they are directed, etc. Even script injection, e.g. Cross-Site Scripting, can be integrated into our framework akin to the approach for user inputs. Building upon this user input functionality, our model can also detail and analyse a non-interference-based Same-Origin Policy. For those interested in a more detailed framework to verify the confidentiality and integrity properties of browser-based security protocols or to extend the model for other properties like non-interference focused Same-Origin Policy, we have made the original version of our model available, including key components of the browser e.g. DOM API, JavaScript elements, XmlHttpRequest. These components are not used in our case studies (M_{cam} and M_{sch}), but they are made available in the generic model. As previously highlighted in our comparison with the Featherweight Firefox model, our model also provides a more detailed list of JavaScript functionalities.

Bauer et al. presented a comprehensive formal model aiming to prove non-interference using taint tracking in browsers [14]. Their model covers user actions, servers, browser tabs, cookies, event handlers, scripts, bookmarks, history and browser extensions. In addition, it supports policy specifications e.g. Content Security Policy (CSP). Their model is not mechanised with a model checking tool. They also implemented a prototype in the Chromium browser to validate that their design enforces additional protection in the browser. In contrast, our prototype serves a different purpose, it is implemented to create test cases for browsers from the counterexamples unearthed during model checking.

The WebSpec framework introduced by Veronese et al. [170] is closest to our framework, particularly in its pursuit to establish a browser model centred on security analysis for identification of security vulnerabilities and ensuring cross-browser validation of test cases. The aim shared by both frameworks is to identify potential security loopholes in browsers before they can be exploited by malicious entities. Both frameworks, developed independently, aim to proactively identify and mitigate potential browser security vulnerabilities. The WebSpec framework is similarly formulated as a state transition system. It similarly captures various elements of the browser, encompassing aspects like browser windows, documents, the DOM API, cookies, HTTP request-response mechanism that is further complemented with CORS and CSP response headers. Notably, it integrates various security policies ranging from CSP and Service Workers to Cache API and Local Storage API. These choices are reflective of the case studies and specific security concerns that WebSpec aims to address. In contrast, while our model sidesteps these specific security policies, it takes into account other pivotal components and security features that are equally important, like iframes, sandboxes, resource rendering, URL manipulation, origin assignment, secure contexts, media device accesses and cross-origin isolation state in the browser. These components and features are selected based on the specific case studies and security scenarios we aim to explore. While both WebSpec and our framework focus on ensuring web security by discovering potential vulnerabilities, they employ different tools and approaches. WebSpec provides a formalisation with Coq and utilises the Z3 theorem prover [199] for model checking. In contrast, our methodology employs a more direct model-checking approach with Alloy. One significant advantage of this choice is the accessibility and familiarity that Alloy offers to browser testers and developers. Unlike the more abstract constructs used in Coq, Alloy's input language is based on Object-Oriented Programming constructs, making it more intuitive for those with a background in software development. In addition, Coq employs the logical constructions to facilitate manual proofs, offering stronger guarantees in the process. On the other hand, Alloy focuses on automated model checking within bounded models, providing a more direct approach to verification. For cross-browser validation, WebSpec utilises Web Platform Tests [157], which are aimed at standards compliance. On the other hand, we employ BrowserStack [28] to ensure that our test cases are not only reliable but also applicable across a broader range of browser environments. Given the complementary scopes of the models and methodologies of the two frameworks, they stand as valuable contributions to the field of browser security. When considered together, they offer a multifaceted approach to understanding, analysing and improving browser security and privacy.

The browser models that we highlighted so far, including our model, can be used to validate new browser-based security protocols, allowing to preemptively address vulnerabilities, instead of reacting after exploitation. Our framework and WebSpec include test case generation which assists in replicating intricate attack scenarios in a real-world browser environment, bridging the gap between theoretical and practical worlds. Other research formulates the browser in order to prove higher-level properties e.g. Bhargavan et al [19]. They defined Defensive JavaScript, that can safely be embedded into untrustworthy web pages to isolate sensitive information even if some JavaScript running in the same browsing context compromises the execution environment. They verify their components using formal methods. However, such security protocols rely on browsers' implementing their core components correctly. When the core components function as intended, the protocols layered on top of them can also operate effectively. Consequently, careful design and thorough validation of these core browser components are crucial.

5.12 Discussion

Our findings indicate that our end-to-end framework has significant implications for web browser security, in particular in the context of the gap between RFC documents and real-world implementations. Our framework can detect security and privacy bugs in browsers due to discrepancies between RFCs and actual implementations and also confirm their absence. This makes the framework an invaluable tool for browser developers and testers who seek to align their software more closely with existing RFC standards and improve security.

One of the main strengths of our framework is that the model takes into account a wide range of browser functionalities, from basic navigation tasks to complex interactions with various APIs and security policies. This holistic approach allows us to identify and analyse a diverse set of security vulnerabilities with the model that might be overlooked in less exhaustive models. Furthermore, the other standout strength of the model is its extensiveness. Its modular design makes it easier to integrate new browser features, APIs and security policies into the model. For example, incorporating a new JavaScript API into the model can be easily accomplished by appending a new 'event' to the 'Function' signature and crafting a corresponding predicate that handles the event's associated inputs, and preconditions and postconditions within the state.

On the other hand, while our model offers detailed insights, it may also face scalability challenges. Model checking is computationally intensive task in its nature. The more detailed model means the more computational resources it requires for effective verification. Also, in some cases, model checking may not even terminate within a reasonable time, especially when the model is highly detailed or the scope given for the property under examination is too large. These present limitations for the framework's scalability when quick verification is required. However, the modular nature of our model offers a workaround to this limitation. Components of the model that are not directly related to the property of interest can be removed, allowing for a more focused and efficient model checking process, as demonstrated in the case studies presented in

this chapter. This adaptability partially addresses the scalability issue.

Additionally, the parser, test case generator and test case executor modules in the framework are currently specialised for the camera access model in our first case study. A more generic implementation of these modules is required for broader applicability. Specifically, developing a universal Alloy-to-Java parser capable of interpreting Alloy output regardless of the underlying formal model structure can extend the framework's adaptability. Following this, a more robust architecture can be defined for the framework in which the parser, test case generator and test case executor can be developed as modular, loosely coupled components. Each component can have its own internal architecture but would operate independently, allowing for seamless integration with a variety of model outputs.

Moreover, Fuzzing can enhance the formal modelling step in our framework which currently relies on manual testing for browser fault detection and incorporates known vulnerabilities into the formal model. Fuzzing can automate the process of discovering faulty behaviours in browser by generating random or grammar based inputs and observing the browser's response. Fuzzing can efficiently help finding a wider spectrum of faulty behaviours. Once the fuzzer identifies certain faults, they can be analysed, categorised and incorporated into the model. Then, the model checker can verify whether these faults could lead to more significant attack vectors.

Incorporating Counterexample Guided Abstraction Refinement (CEGAR) [43] is another avenue for enhancing our framework. Currently, our framework relies on the manual process that the model is iteratively refined based on the spurious counterexamples that do not lead to the same behaviour in the browser. The refinement process could be automated with CEGAR. CEGAR is a method that constructs a more precise model of a software system from a less precise model in an iterative process using the spurious counterexamples.

Furthermore, machine learning language models could also enhance the modelling process in our framework, by facilitating automated model construction directly from RFC documents. Since RFC documents are often well structured, language models could be trained to analyse these documents and automatically generate formal models. This can reduce the manual effort required to specify the model and help to minimise human errors in the model.

5.13 Conclusion

With our framework, we provide a basis for a comprehensive and exhaustive formal analysis of web browsers, thereby enhancing trust in their underlying architecture. The effectiveness of any security protocols designed for browsers fundamentally relies on the accurate implementation of the browser's core components. By offering a robust mechanism for detecting software bugs and verifying the absence of security and privacy vulnerabilities, our framework ensures these components function as intended. This contributes to elevating the overall trustworthiness and reliability of browsers, providing both developers and users with increased confidence in the security measures in place.

Our framework has the potential for near-full automation by integrating techniques such as Fuzzing, CEGAR and machine learning language models as outlined in the Discussion Section 5.12. Beyond these, the browser model can be further elaborated to include additional browser functionalities such as security policies e.g. Content Security Policy, browser extensions e.g. Accountable JavaScript [57], as well as user-triggered actions e.g. form submissions and other advanced JavaScript functionalities e.g. Service Workers. Building on this, after achieving comprehensive verification of low-level browser features and security policies, we consider the integration of real-world web applications into the model. This would offer more practical insights into how our framework could serve as a valuable tool for enhancing browser security in real-world scenarios.

6

Conclusion

In this thesis we addressed trust requirements in three key domains of the web. In the first domain, social media discourse, we introduce a protocol named "Trollthrottle", as detailed in Chapter 3. This protocol is designed to support the credibility of online information by setting limitations on the number of comments a user can post on websites that participate in the protocol. By imposing such limitations, the protocol raises the cost of astroturfing. Additionally, Trollthrottle ensures that participating websites are held accountable for any censorship of content. This serves to combat the creation of echo chambers by social media platforms, thereby contributing to a more balanced and trustworthy online discourse.

From a technical standpoint, the Trollthrottle protocol is not embedded within the browser, but is required to be delivered by the participating web servers. Therefore, the protocol's effectiveness relies on the accurate transmission and execution of its code by the browser. For instance, a malicious web server can target users by sending different codes to different users without detection. To address this trust issue on all websites, we introduce our second protocol named "Accountable JS" in Chapter 4. This protocol is designed to ensure trust in the underlying web application code. For this the website developers are required to declare a manifest for the active content in their application in a public log. This enables users to verify that the active content in the web application code delivered by a website is consistent with respect to the version declared in public logs, thus confirming the active content in the web application is consistent for all visitors. This architecture substantially mitigates the risk of code tampering and creates a higher level of trust among end users. The protocol implementation runs as an external feature in the browser and ensures the active content is compliant with the guidelines declared in the manifest.

The Accountable JS protocol also requires trust in the web browser itself for faithful execution. Our third proposal called "Formal Browser Model for Security Analysis" in Chapter 5 targets this issue. This end-to-end framework facilitates a formal model of web browser based on RFC specification to identify potential security and privacy vulnerabilities through model checking. If a counterexample is found during model checking, the framework generates a test case scenario from this counterexample and executes this counterexample in the browser to validate if the vulnerability exists in the real-world browser. This rigorous approach enables browser developers and testers to align their browser implementation closely with RFC guidelines, effectively identifying and mitigating vulnerabilities. By ensuring that browsers have been put through this rigorous testing and validation process, we significantly increase user trust, as they can be confident that their browser adheres to established security benchmarks outlined by RFCs.

Possible future work for this thesis includes enriching the manifest in Accountable JS protocol by incorporating user interface information e.g. form fields. This enhancement of the protocol could further ensure that specific information declared in the manifest cannot be used by some third-party JavaScript and such information never leaves the user's browser. Additionally, the formal model in "Formal Browser Model for Security Analysis" framework could be extended to encompass a wider range of browser functionalities such as security policies e.g. Content Security Policy, browser extensions

e.g. Accountable JS, as well as user-initiated actions e.g. form submissions and other more advanced JavaScript functionalities e.g. Service Workers. A compelling case study for this extended formal model would be to examine the integration of the Accountable JS protocol with the enriched manifest.

7

Throllthrottle - Appendix

7.1 Instant linkability

In Brickell and Li's scheme [27], signatures are composed of the actual signature and the corresponding pseudonym. To check whether two valid signatures are linked, i.e., were created by the same signer with the same $dom \neq \perp$, one merely compares the nyms. We exploit this property to instantly check whether a new comment is linked to any of the previous comments and thus avoid 'double spending' of basenames. To generalise to other DAA schemes, we formalise this requirement as follows:

Definition 5 (Instantly linkable DAA scheme). *We call a DAA scheme* instantly linkable *if:*

1. *A nym can be generated without knowledge of the data m, i.e. there is a deterministic poly-time algorithm* NymGen *s.t.* $\mathsf{NymGen}(sk_S, dom) =$

$$\mathsf{NymExtract}(\mathsf{Sign}_{\mathsf{DAA}}(sk_S, cred, dom, \cdot)).$$

2. *Signatures contain a nym that links them, i.e., there is a deterministic poly-time algorithm* $\mathsf{NymExtract}$ *s.t. for all signatures* σ_1, σ_2, $\mathsf{Link}(\sigma_1, \sigma_2) = 1$ *iff* $\mathsf{NymExtract}(\sigma_1) = \mathsf{NymExtract}(\sigma_2)$.

3. *The basename used to create a signature can be checked without knowing the data m and is uniquely defined by the signature, i.e. there exists a poly-time algorithm* $\mathsf{VerifyBsn}$ *s.t. for all PPT adversary A, the following probability is negligible in λ*

$$\Pr\Big[(dom, dom', \sigma, pk_I, m) \xleftarrow{\$} A(1^\lambda) : \\ \mathsf{VerifyBsn}(\sigma, dom') = 1 = \mathsf{VerifyBsn}(\sigma, dom) \ \wedge \\ \mathsf{Verify}_{\mathsf{DAA}}(pk_I, m, dom, \sigma, RL_\emptyset) = 1\Big],$$

where $RL_\emptyset$ corresponds to an empty revocation list.

7.2 Security Analysis

Here we show the intuition that the protocol we propose here enforces the threshold τ, valid comments cannot be forged, users remain anonymous and its accountability mechanism is sound and complete.

We define the security goals of TrollThrottle in terms of five properties within an experiment. The adversary has access to oracles for user creation, honest execution of the commenting procedure and the $\mathsf{Join} - \mathsf{Issue}$ protocol, and she can corrupt both users and verifiers. The formal model (Section 7.4) and full proofs (Section 7.6) are available in TrollThrottle Appendix.

1. *Correctness,* intuitively: honest users should always be able to create and publish a comment (acceptable by the policy of the website) and the comment should appear on the website. Moreover, if a comment is not published, the user should be able to generate a claim that can be publicly verified.

2. *Protection against trolling,* intuitively: the number of valid comments that the adversary $\mathcal{A}$ can produce *per basename* is at most the number of users that she corrupted plus the number of users maliciously verified by a corrupted verifier. Let n be the number of user identities under adversarial control (either by bribing the user or by bribing the verifiers) this directly results in the bound $\tau \cdot n$ for the number of comments the adversary can emit per epoch.
3. *Non-frameability,* $\mathcal{A}$ cannot create comments that can be linked to a nym of an honest user.
4. *Anonymity,* intuitively: When challenged with distinguishing a comment produced by a user of her choice from a freshly created user, the adversary can do no better than a guess.
5. *Accountability,* intuitively: Whatever the adversary does, for any honestly generated comment one can produce a verifiable claim that this comment ought to be published. Furthermore, it is not possible to produce such a claim in the name of an honest user unless the comment has been produced by her.

Theorem 1 (Protection against trolling). *The TrollThrottle protocol satisfies protection against trolling if the DAA scheme is user-controlled traceable and instantly linkable, h is collision resistant and we have proofs of knowledge for the relation* $\mathcal{R}_{\mathsf{Join}}$.

Sketch. An adversary can fake comments in three ways:

- by creating a fresh digital identity,
- using an existing signature under a different message, or
- forging valid pseudonyms with acceptable basenames for an existing signature (this would allow her to publish the same comment multiple times).

An adversary that uses the first strategy can be used to break the user-controlled traceability of the DAA scheme. The second attack would break collision resistance. Finally, due to instant linkability of the DAA scheme, we conclude that the adversary cannot find a second nym that is valid under the same basename. □

Theorem 2 (Non-frameability). *The TrollThrottle protocol satisfies non-frameability if the underlying DAA scheme is user-controlled traceable and instantly linkable, the function h is collision-resistant and the proof system for relation* $\mathcal{R}_{\mathsf{Join}}$ *is a proof of knowledge.*

Sketch. We use a similar observation. An adversary that can post in the name of an honest user generates a forgery for the DAA scheme and thus can be used to break user-controlled traceability or break the collision resistance of the hash function. □

Theorem 3 (Anonymity). *The TrollThrottle protocol satisfies anonymity if the underlying DAA scheme provides user-controlled anonymity and the proof system for relation* $\mathcal{R}_{\mathsf{Join}}$ *is zero-knowledge.*

Sketch. Since user-controlled anonymity of the DAA scheme ensures that the adversary cannot tell which of two uncorrupted users signed a message it follows that this implies that an adversary cannot also tell which user commented (since comments are signed using DAA signatures). Note that because of the zero-knowledge property the proof for relation $\mathcal{R}_{\mathsf{Join}}$ can be simulated. □

Theorem 4 (Accountability). *The TrollThrottle protocol satisfies accountability if it is correct and h is collision-resistant.*

Sketch. Correctness of TrollThrottle ensures that honest users can always generate a valid claim even if an adversary tries to prevent this. The evidence produced can be compared with the ledger state, as the message is given. If the message claimed is accepted, but different from a valid entry in the ledger, it constitutes a collision. □

7.3 Holding the Issuer Accountable

In this section we consider an extended version of our protocol that copes with an untrusted issuer, i.e. how can we protect against trolling even if the Issuer is untrusted. The high-level idea is to use so-called genesis tuples for every new user, which are signed using a standard signature scheme by the verifier checking the personal data of the user. Then while commenting, the user proofs that there exists one genesis tuple that corresponds to her identity. A malicious issuer can create an unlimited number of DAA credentials but cannot generate genesis tuples at will without colluding with a verifier. We cannot protect against such a collusion but our approach allows the public to track suspicious behaviour, i.e. one verifier is signing a high volume of genesis tuples.

We begin this section by recalling cryptographic primitives used in this version. We then show how to extend the existing notion of our accountable commenting scheme and define a property we call *credibility*, which will formally capture a dishonest issuer trying to selectively troll the system.

7.3.1 Preliminaries

In addition to an instantly linkable DAA scheme, this scheme assumes a standard existentially unforgeable digital signature scheme $(\mathsf{KG}_{\mathsf{sig}}, \mathsf{sig}, \mathsf{ver})$. The user has access to signing oracle for a verifier with key pair (sk_V, pk_V). Moreover, we recall the definitions given by Groth et al. [72].

Definition 6 (Zero-Knowledge). *A proof system* Π *is called zero-knowledge, if there exists a PPT simulator* $\mathsf{Sim} = (S_1, S_2)$ *such that for all PPT algorithms* $\mathcal{A}$ *the following probability, denoted by* $\mathbf{Adv}[\mathcal{A}_{\Pi}^{\mathsf{ZK}}]$*, is negligible in the security parameter* 1^λ*:*

$$\Big|\Pr\left[\rho \leftarrow \mathsf{Setup}(1^\lambda) : \mathcal{A}^{\mathsf{CreateProof}(\rho,\cdot,\cdot)}(\rho) = 1\right] - \Pr\left[(\rho,\tau) \leftarrow S_1(1^\lambda) : \mathcal{A}^{S(\rho,\tau,\cdot,\cdot)}(\rho) = 1\right]\Big|,$$

where τ *is a trapdoor information,* $S(\rho, \tau, x, w) = S_2(\rho, \tau, x)$ *for* $(x, w) \in \mathcal{R}$ *and both oracles output* $\perp$ *if* $(x, w) \notin \mathcal{R}$*.*

Definition 7 (Soundness). *A proof system* Π *is called sound, if for all PPT algorithms* $\mathcal{A}$ *the following probability, denoted by* $\mathbf{Adv}[\mathcal{A}_{\Pi}^{\mathsf{sound}}]$*, is negligible in the security parameter* 1^{λ}*:*

$$\Pr\begin{bmatrix} \rho \leftarrow \mathsf{Setup}(1^{\lambda}), \\ (x,\pi) \leftarrow \mathcal{A}(\rho) \end{bmatrix} : \begin{matrix} \mathsf{VerifyProof}(\rho,x,\pi) = \mathtt{accept} \\ \wedge \quad x \notin L_{\mathcal{R}} \end{matrix}\Bigg].$$

Definition 8 (Knowledge Extraction). *A proof system* Π *is called a proof of knowledge for* $\mathcal{R}$*, if there exists a knowledge extractor* $\mathsf{Extr} = (E_1, E_2)$ *as described below. For all algorithms* $\mathcal{A}$*:*

$$\begin{aligned} &|\Pr[\rho \leftarrow \mathsf{Setup}(1^{\lambda}) : \mathcal{A}(\rho) = 1] - \\ &\Pr[(\rho,\tau) \leftarrow E_1(1^{\lambda}) : \mathcal{A}(\rho) = 1]| \leq \mathbf{Adv}[\mathcal{A}_{\Pi}^{E_1}] \end{aligned}$$

$$\begin{aligned} &\Pr[(\rho,\tau) \leftarrow E_1(1^{\lambda}), (x,\pi) \leftarrow \mathcal{A}(\rho), w \leftarrow E_2(\rho,\tau,x,\pi) : \\ &\quad \mathsf{VerifyProof}(\rho,x,\pi) = \mathtt{reject} \quad \vee \quad (x,w) \in \mathcal{R}] = 1] \end{aligned}$$

$\mathbf{Adv}[\mathcal{A}_{\Pi}^{E_1}]$ *is negligible in* 1^{λ}*.*

7.3.2 Accountable Commenting Scheme with Credibility (ACSC)

We define an accountable commenting scheme with the additional property of credibility as follows:

Definition 9 (ACSC). *An accountable commenting scheme with credibility consists of a tuple of algorithms* $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Comment}, \mathsf{Verify}, \mathsf{Claim}, \mathsf{VerifyClaim}, \mathsf{Attribute})$ *and an interactive protocol* $(\mathsf{Join} - \mathsf{Issue})$ *with inputs and outputs specified as follows.*

All algorithms are defined in a similar way to the ones for the standard scheme presented in Def. 1. The only differences are as follows:

1. the Join algorithm of the $\mathsf{Join} - \mathsf{Issue}$ protocol additionally outputs a genesis tuple gb,
2. the $\mathsf{Comment}$, Verify and $\mathsf{VerifyClaim}$ algorithms take as an additional list GB containing genesis tuples.

In addition, we define the PPT algorithm $\mathsf{Attribute}(gb)$ that allows the public to attribute a genesis tuple gb to a verifier V by outputting the verifier's public key, which uniquely identifies the verifier. Even if the issuer is colluding with selected verifiers, the public can attribute users to verifiers, and thus gather statistics on how many users were verified by which V that could expose cheaters.

7.3.3 Instantiation

We will now define an efficient instantiation of an accountable commenting scheme with credibility. The scheme closely resembles the scheme presented in Def. 3, but

includes the generation and verification of genesis tuples. In particular, let use define the following relation that users will use to prove knowledge of genesis tuples:

$$\begin{aligned}&((nym, dom, \mathrm{GB}), (sk_S)) \in \mathcal{R}_{\mathrm{GB}} \iff \\ &\exists(\cdot, nym_1, \cdot) \in \mathrm{GB} \quad \wedge \quad nym_1 = \mathsf{NymGen}(sk_S, 1) \\ &\qquad\qquad\qquad\quad \wedge \quad nym = \mathsf{NymGen}(sk_S, dom).\end{aligned}$$

Definition 10. *Extended TrollThrottle Protocol*

$\mathsf{Setup}(1^\lambda)$ - *compute* $\rho_{\mathrm{GB}} \xleftarrow{\$} \mathsf{Setup}_{\mathsf{ZK}}(1^\lambda)$,
$\rho_{\mathsf{Join}} \xleftarrow{\$} \mathsf{Setup}_{\mathsf{ZK}}(1^\lambda)$ *and*
output $\rho = (1^\lambda, \rho_{\mathsf{Join}}, \rho_{\mathrm{GB}})$.

$\mathsf{KeyGen}(\rho)$ - *equal to* KeyGen *in Def. 3.*

$\mathsf{Join}(pk_{\mathrm{I}}, sk_U, U)$ - *parse* $pk_{\mathrm{I}} = pk_{I,\mathsf{DAA}}$ *and* $sk_U = sk_{U,\mathsf{DAA}}$.
Execute $\mathsf{com} \xleftarrow{\$} \mathsf{Join}_{\mathsf{DAA}}(pk_{I,\mathsf{DAA}}, sk_{U,\mathsf{DAA}})$ *and*
compute proof $\Pi_{\mathsf{Join}} = \mathsf{CreateProof}(\rho_{\mathsf{Join}}, (\mathsf{com}, pk_{I,\mathsf{DAA}}), sk_{U,\mathsf{DAA}})$.
Send $(\mathsf{com}, \Pi_{\mathsf{Join}})$ *to the issuer and receive* $cred_U$.

Compute the pseudonym $nym_1 = \mathsf{NymGen}(sk_U, 1)$, *and*
set $gb_U = (pk_V, nym_1, \mathsf{sig}(sk_V, nym_1))$,
where $\mathsf{sig}(sk_V, nym_1)$ *was created by an identity verifier.*

Return $(cred_U, sk_U), gb_U)$.

$\mathsf{Issue}(sk_{\mathrm{I}}, \mathsf{ver}, U)$ - *equal to* Issue *in Def. 3.*

$\mathsf{Comment}(pk_{\mathrm{I}}, sk_U, cred_U, dom, m)$ - *set and return* $\gamma = (\sigma, nym, dom, h(m), \Pi)$
where $\Pi = \mathsf{CreateProof}(\rho_{\mathrm{GB}}, (nym, dom, \mathrm{GB}), sk_U))$,
$\sigma = \mathsf{Sign}_{\mathsf{DAA}}(sk_U, cred_U, dom, h(m))$ *and*
$nym \xleftarrow{\$} \mathsf{NymGen}(sk_U, dom) = \mathsf{NymExtract}(\sigma)$.

$\mathsf{Verify}(pk_{\mathrm{I}}, nym, dom, \mathrm{GB}, m, \gamma)$ - *Parse* $pk_{\mathrm{I}} = pk_{I,\mathsf{DAA}}$ *and* $\gamma = (\sigma, nym, dom, h, \Pi)$.
Output 1 *iff*

- $\mathsf{Verify}_{\mathsf{DAA}}(pk_{I,\mathsf{DAA}}, h, dom, \sigma, RL_\emptyset) = 1$,
- *if* $h(m) = h$,
- $\mathsf{NymExtract}(\sigma) = nym$,
- $\mathsf{VerifyBsn}(\sigma, dom) = 1$.
- $\mathsf{VerifyProof}(\rho_{\mathrm{GB}}, (nym, dom, \mathrm{GB}), \Pi)) = 1$.

$\mathsf{Claim}(pk_{\mathrm{I}}, sk_U, cred, dom, m, \gamma)$ - Claim *in Def. 3.*

$\mathsf{VerifyClaim}(pk_{\mathrm{I}}, \mathrm{GB}, dom, m, \gamma, evidence)$ - *return 1 iff* γ *is valid for* m,
i.e., that $\mathsf{Verify}(pk_{\mathrm{I}}, nym, dom, \mathrm{GB}, m, \gamma) = 1$.

$\mathsf{Attribute}(gb)$ - *parse* $gb = (pk_V, nym_1, \mathsf{sig}(sk_V, nym_1)$ *and return* pk_V.

7.3.4 Security Analysis

Here we will formally define what it means for an accountable commenting scheme to have the credibility property and proof that the scheme presented above fulfils it.

Definition 11. *We say that the system is credible if for every adversary $\mathcal{A}$, every 1^λ, the probability* $\Pr[\mathsf{Exp}_{\mathcal{A}}^{\mathsf{credibility}}(1^\lambda) = 1]$ *is negligible* 1^λ.

Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{credibility}}(1^\lambda)$**:**

$\mathrm{CU} \leftarrow \emptyset$; $V \leftarrow \emptyset$; $\mathsf{ver} \leftarrow \emptyset$;
$(sk_\mathrm{I}, pk_\mathrm{I}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
$\mathcal{O} = \{\mathsf{CorruptVer}(\cdot, \mathsf{ver}), \mathsf{JoinSystem}(\cdot, \mathsf{ver}, sk_\mathrm{I})\}$
$\mathrm{OUT} = \{(nym_i^*, dom_i^*, m_i^*, \gamma_i^*)\}_{i=1}^k \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, sk_\mathrm{I})$
Return 0 if $\mathsf{Verify}(pk_\mathrm{I}, nym_i^*, dom_i^*, m_i^*, \gamma_i^*) = 0$ for any $i \in \{1, \ldots, k\}$
$S = \{(nym, dom) : ((\cdot, nym, dom, \cdot)) \in \mathrm{OUT}\}$
$t = \max_{(\cdot, dom) \in S} \ |\{(nym) : (nym, dom) \in S\}|$
Return 1 iff $t > |\mathrm{VM}|$

Figure 7.1: Trollthrottle Credibility

Theorem 5 (Credibility). *The Extended TrollThrottle protocol (see Def. 10) satisfies credibility (see Def. 11) if the underlying DAA scheme is instantly linkable (Def. 5) and the proof system for relation $\mathcal{R}_{\mathrm{GB}}$ is sound.*

Proof. The idea behind the proof is as follows. Because we know that to win an adversary has to return $t > |\mathrm{VM}|$ valid signatures for one basename but at the same time there exist only $|\mathrm{VM}|$ genesis tuples. It is easy to see now that if the adversary wins, then there must exist at least proof Π_i^*, where $\gamma_i^* = (\cdot, \cdot, \cdot, \cdot, \Pi_i^*)$ that is false and can be used to break the soundness property of the proof system. Note that this follows from the fact that because of instant linkability there can only exist $|\mathrm{VM}|$ secret keys that form the pseudonyms nym_1 in a genesis tuple. □

7.3.5 Efficient instantiation of the proof for relation $\mathcal{R}_{\mathrm{GB}}$

Pseudonyms in the scheme by Brickel and Li are of the form $h(dom)^{sk_S}$, where h is a collision-resistant hash function that maps elements from $\{0,1\}$ to elements of a group $\mathbb{G}$ of order q and the secret key sk_S is an element in $\mathbb{Z}_q^*$. It follows that in such a case we have $nym_1 = h(1)^{sk_S}$ and $nym = h(dom)^{sk_S}$.

To generate Π we will make use of the proof system by Groth and Kohlweiss [71]. They showed an interactive Σ-protocol for the following statement. Given n commitments $c_1, \ldots, c_n$ at least one opens to 0. The communication size is logarithmic in n, which means that by applying the Fiat-Shamir transformation we receive a non-interactive zero-knowledge proof of the same size. Note that this proof system requires the commitment scheme to be homomorphic and the message space to be $\mathbb{Z}_q$. In particular, Groth

and Kohlweiss show that their proof system works for Pedersen commitments where $com(x, r) = g^x \cdot \hat{g}^r$ for some elements $g, \hat{g} \in \mathbb{G}$.

We will now show an efficient proof system for the following statement: Given dom, nym and ledger L with genesis tuples $gb_1, \ldots, gb_b$ (where $gb_i = (\cdot, nym_1^i, \cdot)$) there exists a secret key sk_S and an index j such that $nym = \mathsf{NymGen}(sk_S, dom)$ and $nym_1^j = \mathsf{NymGen}(sk_S, 1)$. First we notice that by setting $g = h(1)$ and $\hat{g} = h(2)$ we can use $g, \hat{g}$ as parameters for a Pedersen commitment scheme. What's more, for all $i \in \{1, \ldots, n\}$ we have $nym_1^i = com(sk_S^{(i)}, 0)$ where $sk_S^{(i)}$ is the secret key of the user that generated tuple gb_i.

To create proof Π the Prover with secret witness sk_S, j proceeds as follows:

1. computes a commitment $c = com(sk_S, r)$ using random coins $r \in \mathbb{Z}_q$,
2. for all $i \in \{1, \ldots, n\}$ computes $c_i = c/nym_1^i = com(sk_S - sk_S^{(i)}, r)$,
3. computes proof π using the system by Groth and Kohlweiss that one of $c_1, \ldots, c_n$ is a commitment to 0,
4. returns proof $\Pi = (c, \pi)$.

To verify the proof a Verifier proceeds as follows:

1. for all $i \in \{1, \ldots, n\}$ computes $c_i = c/nym_1^{(i)}$,
2. verifies proof π using $c_1, \ldots, c_n$ as part of the statement and returns true if and only if this proof is valid.

7.4 Formal analysis of the deferred verification and auditing protocol

```
theory TrollThrottle
begin

builtins: hashing

functions:
    pk/1,sk/1[private],
    sign/3, verify/3,extrmsg/1,
    true/0

equations:
    verify(sign(m, r, sk(i)), m, pk(i)) = true,
    extrmsg(sign(m,r,x))=m

let Issuer =
    let
    skI = sk('I')
    skV = sk('V')
    pkV = pk('V')
    m1 = sid
    pat_m2 =  <login,h(<rU,nbd,'1'>)>
```

```
      cI = h(rI,sid,h(<rU,nbd,'1'>))
      m3 =  sign(cI,r3,skI)
      s1 = h(<rI,sid,'2'>)
      s2 = h(m5)
      claim = <rI,sid,pat_m2,m5>
      in
      out (m1);
      in (pat_m2);
      new rI;
      new r3;
      out (m3);
      in (m5);
      if verify(m5,cI,pkV)=true() then
      event Accepted(nbd);
      ( // case distinction: the attacker may decide to play a ↷
      game
      ( in(xs1); // (a) the dishonest verifier tries to
                 //     predict s1 to avoid auditing
        if xs1 = s1 then
          event VerifierPredict()
      )
      +
      ( // (b) only if the verifier does not want to play this ↷
      game,
        //     she learns rI
      out(<sid,rI>); // we may use a public channel here,
                     // because V considered dishonest anyway
      out(claim)
      )
      )


  let Verifier =
      let
      skI = sk('I')
      pkI = pk('I')
      skV = sk('V')
      cI = h(<rI,sid,h(<rU,nbd,'1'>)>)
      pat_m4 = <nbd,cI,rU,xm3>
      m5 = sign(cI,r5,skV)
      s1 = h(<rI,sid,'2'>)
      s2 = h(m5)
      in
      in(pat_m4);
      if verify(xm3,cI,pkI)=true() then
      lock cI; // make sure that for the same cI,
               // only one execution is started
      in(evidence);
      event Verified(nbd,evidence);
      new r5;
      event InResponse(rI,sid,m5);
      ( // As before, adversary can decide
      ( in(xs2); // (a) to let a dishonest issuer try to ↷
      predict s2
                 //     as to chose m5 in a way that avoids ↷
                 auditing
```

```
        if xs2 = s2 then
          event IssuerPredict()
      )
      +
      ( // (b) continue the protocol, as usual
      out(m5);
      in(<sid,rI>);
      // starting from here, assume an audit takes place
      // (adversary can decide whether to run this step)
      out(<rU,nbd>)
      )
      )

  let CheckClaim = let
  // A dishonest issuer tries to claim that V needs to send
  // evidence, although the values do not match.
  // I controls s1 but honest V choses s2
  // Hence the issuer wins if she can send s1 and s2 that pass
  // checking, but s2 was not the answer of verifier to sid and
  // rI used to computed it
      skI = sk('I')
      skV = sk('V')
      pkV = pk('V')
      m3 =  sign(cI,r3,skI)
      pat_claim = <rI,sid,xm2,xm5>
      cI = h(rI,sid,xm2)
      /* s1 = h(<rI,sid,'2'>) // not used, but this can be used↷
       to recompute the check */
      s2 = h(xm5) //
      in
      in(pat_claim);
      if verify(xm5,cI,pkV) = true() then
          event ClaimAccept(rI,sid,xm5)

  !(
   ( new sid; Issuer)
  |Verifier)
  |CheckClaim
  | !(in(p:pub); event Corrupted(p); out(sk(p)))

  lemma sanity : // considers dishonest user
      exists-trace
        "Ex nbd #i. Accepted(nbd) @ i
        &
        not (Ex #j p. Corrupted(p)@j)
        "

  lemma authenticity_accept : // considers dishonest user
        "All nbd #i. Accepted(nbd) @ i ==>
          (Ex #j evidence. Verified(nbd,evidence) @ j)
       | (Ex #j. Corrupted('V')@j)
       | (Ex #j. Corrupted('I')@j)
        "

  lemma unpred_issuer:
      " All #i. IssuerPredict()@i ==> Ex #j. Corrupted('V')@j"
```

```
lemma unpred_verified:
    " All #i. VerifierPredict()@i ==> Ex #j. Corrupted('I')@j↷
    "

lemma authenticity_claim : // if claim is accepted, then the
                           // verifier indeed responded to ↷
                           values
                           // that determine the decision
      "All ri sid m5 #i. ClaimAccept(ri,sid,m5) @ i
             ==> (Ex #j . InResponse(ri,sid,m5) @ j)
             |   (Ex #j. Corrupted('V')@j)
             "

end
```

Listing 7.1: Formal model of the deferred verification and auditing protocol in Trollthrottle

7.5 Review and adoption of the security model

We review Brickell and Li's security model [27], including the user-controlled-anonymity and user-controlled-traceability experiment. Their DAA scheme satisfies both notions under the decisional/ strong Diffie-Hellmann assumption. We slightly simplify their model, as our protocol's computations are performed by a single host and not split between a TPM and an untrusted device.

Definition 12 (User-controlled-anonymity). *A DAA scheme is user-controlled-anonymous if no PPT adversary can win the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$, i.e. if* $\mathbf{Adv}[\mathcal{A}_{\mathsf{DAA}}^{\mathsf{anonymity}}] = \Pr[\mathcal{A}\ wins]$ *is negligible:*

- Initial*: $\mathcal{C}$ runs* $\mathsf{Setup}_{\mathsf{DAA}}(1^{\lambda})$ *and gives the resulting sk_I and pk_I to $\mathcal{A}$.*
- Phase 1*: $\mathcal{C}$ is probed by $\mathcal{A}$ who makes the following queries:*
  - *Sign: $\mathcal{A}$ submits a signer's identity S, a basename dom (either $\perp$ or a data string) and a message m of her choice to $\mathcal{C}$, who runs* $\mathsf{Sign}_{\mathsf{DAA}}$ *to get a signature σ and responds with σ.*
  - *Join: $\mathcal{A}$ submits a signer's identity S of her choice to $\mathcal{C}$, who runs* $\mathsf{Join}_{\mathsf{DAA}}$ *with $\mathcal{A}$ to create sk_S and to obtain a set of valid credentials $cred_S$ from $\mathcal{A}$. $\mathcal{C}$ verifies the validation of $cred_S$ and keeps sk_S secret.*
  - *Corrupt: $\mathcal{A}$ submits a signer's identity S of her choice to $\mathcal{C}$, who responds with the value sk_{DAA} of the signer.*
- Challenge*: At the end of phase 1, $\mathcal{A}$ chooses two signers' identities S_0 and S_1, a message m and a basename dom of her choice to C. A must not have made any Corrupt query on either S_0 or S_1, and not have made the Sign query with the same dom if dom $\neq\perp$ with either S_0 or S_1. To make the challenge, $\mathcal{C}$ chooses a bit b uniformly at random, signs m associated with dom under $(sk_{S_b}, cred_{S_b})$ to get a signature σ and returns σ to $\mathcal{A}$.*

- phase 2: *$\mathcal{A}$ continues to probe $\mathcal{C}$ with the same type of queries that it made in phase 1. Again, $\mathcal{A}$ is not allowed to corrupt any signer with the identity either S_0 or S_1, and not allowed to make any Sign query with dom if dom $\neq\perp$ with either S_0 or S_1.*
- Response: *$\mathcal{A}$ returns a bit b'. We say that the adversary wins the game if $b = b'$*

Definition 13 (User-controlled-traceability). *A DAA scheme is user-controlled-traceable if no probabilistic polynomial-time adversary can win the following game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$, i.e. if $\mathbf{Adv}[\mathcal{A}^{\mathsf{trace}}_{\mathsf{DAA}}] = \Pr[\mathcal{A}\ wins]$ is negligible:*

- Initial*: $\mathcal{C}$ runs $\mathsf{Setup}_{\mathsf{DAA}}(1^\lambda)$, gives the resulting pk_I to $\mathcal{A}$ but keeps sk_I.*
- Phase 1*: $\mathcal{C}$ is probed by $\mathcal{A}$ who makes the following queries:*
  - *Sign: The same as in the game of user-controlled-anonymity.*
  - *Join: There are two cases of this query. Case 1: $\mathcal{A}$ submits a signer's identity S of her choice to $\mathcal{C}$, who runs $\mathsf{Join}-\mathsf{Issue}_{\mathsf{DAA}}$ to create sk_{DAA} and cred for the signer. Case 2: $\mathcal{A}$ submits a signer's identity S with a sk_{DAA} value of her choice to C, who runs $\mathsf{Join}-\mathsf{Issue}_{\mathsf{DAA}}$ to create cred for the signer and puts the given sk_{DAA} into a revocation list RL. $\mathcal{C}$ responds the query with cred.*

    Suppose that $\mathcal{A}$ does not use a single S for both of the cases.
  - *Corrupt. This is the same as in the game of user-controlled-anonymity, except that at the end C puts the revealed secret key into the list RL.*
- Forge*: $\mathcal{A}$ returns a signer's identity S, a signature σ, it's signed message m and the associated basename dom.*
  We say that the adversary wins the game if
  1. *$\mathsf{Verify}_{\mathsf{DAA}}(pk_I, m, dom, \sigma, RL) = 1$(accepted), but σ is no response of the existing Sign queries,*
     and/or
  2. *In the case of dom $\neq\perp$, there exists another signature σ' associated with the same identity and dom, and the output of $\mathsf{Link}_{\mathsf{DAA}}(\sigma, \sigma')$ is 0 (unlinked).*

We will use the following notation to denote the queries made by the adversary:

$\mathsf{Sign}_{\mathsf{DAA}}(S, dom, m)$ - On input of a signer's identity S, a basename *dom* (either $\perp$ or a data string), a message m the oracle returns signature σ.

$\mathsf{HJoin}_{\mathsf{DAA}}(S)$ - On input of a signer's identity S of her choice, the adversary obtains credentials $cred_S$. The secret key sk_S is kept secret by the oracle. This oracle corresponds to Case 1 joining.

$\mathsf{Join}_{\mathsf{DAA}}(S, sk_S)$ - On input of a signer's identity S of her choice and a secret key sk_S, the adversary obtains credentials $cred_S$. This oracle corresponds to Case 2 joining.

$\mathsf{IHJoin}(S)$ - On input of a signer's identity S of her choice, this interactive honest user joining oracle allows the adversary to issue credentials for an honest user in the name of the issuer. This oracle represents the Join oracle defined in user-controlled-anonymity.

$\mathsf{CorruptUser}_{\mathsf{DAA}}(S)$ - On input of a signer's identity S, the value sk is returned to the adversary.

Definition 14 (correctness). *If both the signer and verifier are honest, then the signatures and their links generated by the signer will be accepted by the verifier with overwhelming probability, i.e. for any secret key sk_S in the user's secret key space if*

$$\begin{aligned}(pk_I, sk_I) &\leftarrow \mathsf{Setup}_{\mathsf{DAA}}(1^\lambda),\\ (com) &\leftarrow \mathsf{Join}_{\mathsf{DAA}}(pk_I, sk_S),\\ (cred_S) &\leftarrow \mathsf{Issue}_{\mathsf{DAA}}(sk_I, com),\\ \sigma_0 &\leftarrow \mathsf{Sign}_{\mathsf{DAA}}(sk_S, cred, m_0, dom), \text{ and}\\ \sigma_1 &\leftarrow \mathsf{Sign}_{\mathsf{DAA}}(sk_S, cred, m_1, dom),\end{aligned}$$

then, with overwhelming probability,

$$1 \leftarrow \mathsf{Verify}_{\mathsf{DAA}}(pk_I, m, dom, \sigma_i), i \in \{0,1\}$$

and

$$1 \leftarrow \mathsf{Link}_{\mathsf{DAA}}(\sigma_0, \sigma_1).$$

Brickell and Li's scheme is easily shown to fulfil our requirement that the Link function can also be represented using the pseudonym that is included in the signature. This pseudonym is fixed per identity and per basename.

Theorem 6. *For any cryptographic collision resistant hash function $h : \{0,1\}^* \to \mathbb{G}$, Brickell and Li's scheme [27] is an instantly linkable DAA scheme if we define:*

$$\begin{aligned}\mathsf{NymExtract}(\sigma) &:= \sigma_2\\ \mathsf{NymGen}(sk_S, dom) &:= h(dom)^{sk_S}\\ \mathsf{VerifyBsn}(\sigma, dom) &:= \begin{cases}1 & \text{if } \sigma_1 = h(dom)\\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

where $\sigma = (\sigma_1, \ldots, \sigma_9)$.

Proof. It is easy to see that pseudonyms of the form $h(dom)^{sk_S}$ are already used by the Brickell and Li scheme but are hidden as part of the signature (i.e. as σ_2). Thus, NymExtract and NymGen work according to the definition of instant linkability. Lastly, we note that the first element of the signature σ_1 is actually the base under which we compute the pseudonym, i.e. $h(dom)$. Note that in our system we always use domain-based DAA signatures and this element is in the range of the hash function and not a random element (as also allowed in the Brickell and Li scheme). □

Theorem 7. *Brickell and Li's scheme [27] with a minor modification is an updatable DAA scheme.*

Proof. The main observation is that in the $\mathsf{Join} - \mathsf{Issue}_{\mathsf{DAA}}$ protocol the user computes a Diffie-Hellman public key $F = h_1^f$ and computes a Schnorr like proof for f, i.e. it computes $R = h_1^{r_f}$, challenge $c = h(pk_I||\mathsf{nonce}||F||R)$ and proof $s_f = r_f + c \cdot f$. The generator h_1 is part of the public key pk_I, nonce is some nonce send by the issuer to prevent replay attacks and $u = (F, c, s_f, \mathsf{nonce})$. What's more, the setup algorithm the generator h_1 by directly sampling a group element, i.e. it is generated using public coins .

One can easily notice that we can set $gpk_1 = (h_1)$ (including the group definition), where gpk_2 contains the remaining parameters (see [27] for a full specification). The only problem we have to tackle with is that the challenge c, used to generate the proof s_f, contains the full public parameters pk_I. Indeed, there is no reason to include the full pk_I besides to protect against cross issuer attacks, i.e. a malicious man-in-the-middle could register a user into a different DAA system (with a different pk_I). However, in our case we want this to be true. What's more important here is that changing the challenge does not break the soundness of the proof. Since the used proof is a standard Fiat-Shamir instantiation of a sigma protocol, it is sufficient that challenge contains remaining values.

□

7.6 Proofs of Security

We first define the adversarial model in terms of the oracles at the attacker's disposal. Then we treat introduce and prove each security property, one by one.

7.6.1 Model Oracles

To model the security of the protocol, we first define the adversarial capabilities in terms of a set of oracles that will be used in the following security definitions. The challenger in all these definitions is defined in terms of these oracles and the winning condition of the adversary.

Definition 15. *We define the following oracles and global sets* CU, HU, VM, USK, CH, COMM *that are initially set empty:*

CorruptUser(U) - *on input of the user identifier this oracle, checks if there is a tuple* $(U, sk_U, cred_U) \in$ USK *then output* $(sk_U, cred_U)$. *Otherwise it outputs* $\perp$. *Finally, it adds* U *to the set* CU *and sets* HU = HU $\setminus \{U\}$.

CorruptVer(U, ver, V) - *on input of the user identifier and database* ver, *this oracle sets* ver$[V, U] = 1$ *and adds* (V, U) *to the set* VM.

CreateHonestUser(U, ver, sk_{I}, V) - *this oracle first checks that* $U \notin$ HU $\cup$ CU *and returns* $\perp$ *if not. Then it sets* ver$[V, U] = 1$ *and runs the* $\mathsf{Join} - \mathsf{Issue}$ *protocol, receiving* $(sk_U, cred_U)$. *Finally, it adds* $(U, sk_U, cred_U, V)$ *into* USK *and* U *to* HU.

$\mathsf{JoinSystem}(U, \mathsf{ver}, sk_{\mathrm{I}}, V)$ - *this oracle first checks that* $U \notin \mathrm{HU} \cup \mathrm{CU}$, $\mathsf{ver}[V, U] = 1$ *and that* $(\cdot, U) \notin \mathrm{VM}$. *It returns* $\perp$, *if both checks fail. Then, it interactively executes* $\mathsf{Issue}(sk_{\mathrm{I}}, \mathsf{ver}, U)$ *by communicating with the adversary.*

$\mathsf{HJoinSystem}(U, \mathsf{ver}, pk_{\mathrm{I}}, V)$ - *this oracle first checks that* $U \notin \mathrm{HU} \cup \mathrm{CU}$ *and* $(\cdot, U) \notin \mathrm{VM}$. *It returns* $\perp$ *if both checks fails. Then it sets* $\mathsf{ver}[V, U] = 1$, *samples a fresh secret key* sk_U *and interactively executes the* $\mathsf{Join}(pk_{\mathrm{I}}, sk_U, U)$ *protocol with the adversary, receiving* $cred_U$. *It then adds* $(U, sk_U, cred_U, V)$ *to* USK *and* U *into* HU.

$\mathsf{CreateComment}(U, dom, m, pk_{\mathrm{I}})$ - *this oracle first checks that* $U \in \mathrm{HU}$ *and then computes* $(nym, \gamma) \xleftarrow{\$} \mathsf{Comment}(pk_{\mathrm{I}}, sk_U, cred_U, dom, m)$.
Finally, it adds (U, nym, dom, m, γ) *to* COMM *and outputs* γ *and the pseudonym* nym.

$\mathsf{Chall}_b(U, dom, m, sk_{\mathrm{I}}, pk_{\mathrm{I}})$ - *this oracle first checks that* $U \in \mathrm{HU}$ *and returns* $\perp$ *if not. If* $(U, dom, sk_{U,dom}, cred_{U,dom}) \notin CH$ *then the oracle executes the* $\mathsf{Join-Issue}$ *protocol to receive a new secret key* $sk_{U,dom}$ *and credential* $cred_{U,dom}$. *Then it adds* $(U, dom, sk_{U,dom}, cred_{U,dom})$ *to* CH *and computes:*

if $b = 0$**:** $(nym, \gamma) \xleftarrow{\$} \mathsf{Comment}(pk_{\mathrm{I}}, sk_U, cred_U, dom, m)$,

else if $b = 1$**:** $(nym, \gamma) \xleftarrow{\$} \mathsf{Comment}(pk_{\mathrm{I}}, sk_{U,dom}, cred_{U,dom}, dom, m)$

Finally, it outputs γ *and pseudonym* nym.

7.6.2 Protection against trolling

Definition 16. *We say that the system protects against trolling if for every adversary* $\mathcal{A}$, *every* 1^λ, *the probability* $\Pr[\mathsf{Exp}^{\mathsf{troll}}_{\mathcal{A}}(1^\lambda) = 1]$ *is negligible* 1^λ.

Experiment $\mathsf{Exp}^{\mathsf{troll}}_{\mathcal{A}}(1^\lambda)$**:**

$\mathrm{CU} \leftarrow \emptyset$; $\mathrm{HU} \leftarrow \emptyset$; $V \leftarrow \emptyset$;
$\mathrm{USK} \leftarrow \emptyset$; $\mathrm{COMM} \leftarrow \emptyset$; $\mathsf{ver} \leftarrow \emptyset$;
$(sk_{\mathrm{I}}, pk_{\mathrm{I}}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
$\mathcal{O} = \{\mathsf{CorruptUser}(\cdot), \mathsf{CreateHonestUser}(\cdot, \mathsf{ver}, sk_{\mathrm{I}}),$
$\mathsf{CorruptVer}(\cdot, \mathsf{ver}), \mathsf{JoinSystem}(\cdot, \mathsf{ver}, sk_{\mathrm{I}}), \mathsf{CreateComment}(\cdot, \cdot, \cdot, pk_{\mathrm{I}})\}$
$\mathrm{OUT} = \{(nym_i^*, dom_i^*, m_i^*, \gamma_i^*)\}_{i=1}^k \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, pk_{\mathrm{I}})$
$\mathrm{OUT} = \mathrm{OUT} \setminus \{(nym, dom, m, \gamma) : (\cdot, nym, dom, m, \gamma) \in \mathrm{COMM}\}$
Return 0 if $\mathsf{Verify}(pk_{\mathrm{I}}, nym_i^*, dom_i^*, m_i^*, \gamma_i^*) = 0$ for any $i \in \{1, \ldots, k\}$
$S = \{(nym, dom) : (nym, dom, \cdot, \cdot) \in \mathrm{OUT}\}$
$t = \max_{(\cdot, dom) \in S} \ |\{(nym) : (nym, dom) \in S\}|$
Return 1 if $t > |\mathrm{CU}| + |\mathrm{VM}|$

Figure 7.2: Trollthrottle protection against trolling

Theorem 8 (Protection against trolling). *The TrollThrottle protocol (see Def. 3) satisfies protection against trolling (see Def. 16) if the underlying DAA scheme provides*

user-controlled traceability (Def. 13) and is instantly linkable (Def. 5), the user hash function is collision-resistant and the proof system for relation $\mathcal{R}_{\mathsf{Join}}$ is a proof of knowledge.

Proof. We begin this proof by noting that the adversary can only win the trolling experiment in three ways.

1. by finding a comment with a DAA signature σ under basename dom_1 and message m that is also valid for basename dom_2 and message m, where $dom_1 \neq dom_2$.
2. forging a signature for an honest user for message m^* where there exists a tuple $(\cdot, \cdot, \cdot, m, \cdot) \in \text{COMM}$ for which $h(m^*) = h(m)$.
3. by creating a "fake" new user without interacting with the issuer or forging a signature for an honest user for message m^* where there exists no tuple $(\cdot, \cdot, \cdot, m, \cdot) \in \text{COMM}$ for which $h(m^*) = h(m)$.

We will now show that any adversary has only a negligible probability to actually perform any of the above attacks. To do so, we will create reductions that interact with the adversary and a DAA challenger for the user-controlled traceability. However, first we show how those reductions will answer the oracle queries of the adversary. In every case, the reduction will play the role of the adversary against the DAA scheme. Thus, it will receive the public key of the DAA issuer pk_I. Moreover, every reduction will keep its own local vector ver (initially zero) and initially empty lists $\text{HU}_{\mathcal{R}}$, $\text{CU}_{\mathcal{R}}$ (different than the ones used in the definitions) and COMM.

Generic way of answering oracle queries by the reduction

$\mathsf{CorruptVer}(U, \mathsf{ver})$ - on a corrupt verification query, the reduction just sets the local value $\mathsf{ver}[U]$ to 1.

$\mathsf{CreateHonestUser}(U, \mathsf{ver}, sk_{\text{I}})$ - on an honest user creation query, the reduction returns $cred_U$ if $(U, cred_U) \in \text{HU}_{\mathcal{R}}$. If such a tuple does not exist, it queries the DAA $\mathsf{HJoin}_{\mathsf{DAA}}(S)$ oracle, where $S = U$, receives a DAA credentials $cred_S$ and adds $(U, cred_U)$ into $\text{HU}_{\mathcal{R}}$ and returns $cred_U = cred_S$.

$\mathsf{CorruptUser}(U)$ - on a corrupt honest user query, the reduction returns $\perp$ if there exists no tuple $(U, cred_U) \in \text{HU}_{\mathcal{R}}$. Otherwise, it queries the DAA oracle $\mathsf{CorruptUser}_{\mathsf{DAA}}(S)$, where $S = U$ and receives the DAA secret key $sk_U = sk_S$ and credentials $cred_U = cred_S$. The reductions updates $\text{HU}_{\mathcal{R}} = \text{HU}_{\mathcal{R}} \setminus \{(U, cred_U)\}$, add $(U, sk_U, cred_U)$ to $\text{CU}_{\mathcal{R}}$ and returns $sk_U = sk_S$.

$\mathsf{JoinSystem}(U, \mathsf{ver}, sk_{\text{I}})$ - on a corrupt user joining query, the reduction returns $\perp$ if $\mathsf{ver}[U] = 0$ or $(U, \cdot) \in \text{HU}_{\mathcal{R}}$. The reduction then uses the extraction algorithm Extr to extract sk_S from Π_{Join}. It then uses its own $\mathsf{Join}_{\mathsf{DAA}}(U, sk_S)$ and obtains credentials $cred_S$. The reduction then adds $(U, sk_S, cred_S)$ to $\text{CU}_{\mathcal{R}}$ and returns $cred_U = cred_S$ to the adversary.

$\mathsf{CreateComment}(U, dom, m, pk_{\text{I}})$ - on a commenting query, the reduction first checks that the query is for an honest user, i.e. that $(U, \cdot) \in \text{HU}_{\mathcal{R}}$ and returns $\perp$ if this

is not the case. It then uses its own signing oracle to query $\mathsf{Sign}_{\mathsf{DAA}}(U, dom, m)$ receiving signature σ and computes $nym = \mathsf{NymExtract}(\sigma)$. Finally, it returns $\gamma = (\sigma, nym, m, dom)$ and it adds (U, nym, dom, m, γ) into COMM.

If the Extr fail, then the reduction also fails. Thus, it is easy to see that the probability of any reduction in simulating the real experiment without error depends heavily on this algorithm. Fortunately, we assumed that they fail only with negligible probability, so does our reduction.

Case 1 We will now discuss that there cannot exist any adversary that can use the first attack strategy. This basically follows from the instant linkability of the DAA scheme, i.e. because of the VerifyBsn, we ensure that signatures are linked to basenames and the probability that any $\mathcal{A}$ finds such a "collision" is negligible.

To show this more formally, let us assume that there exists an adversary $\mathcal{A}$ that wins by returning a valid comment $\gamma_1 = (\sigma^*, \cdot, dom^*, m^*)$ where $(\cdot, nym^*, dom^*, m^*, \gamma_1) \notin$ COMM but there exists a tuple $\gamma_2 = (\sigma^*, \cdot, dom, m^*)$ such that $(\cdot, nym^*, dom, m^*, \gamma_2) \in$ COMM, which is what we assumed in this case. However, because both commitments are valid we know that $\mathsf{VerifyBsn}(\sigma^*, dom^*) = 1$ and $\mathsf{VerifyBsn}(\sigma^*, dom) = 1$, and $\mathsf{Verify}_{\mathsf{DAA}}(pk_I, m, dom^*, \sigma^*, RL_\emptyset) = 1$ and $\mathsf{Verify}_{\mathsf{DAA}}(pk_I, m, dom, \sigma^*, RL_\emptyset) = 1$. Thus, we found a "collision" and broke the instant linkability property of the DAA scheme for which we assumed that there exists no PPT adversary with non-negligible probability.

Case 2 It is easy to see that by winning in this case the adversary $\mathcal{A}$ can be used to break collision-resistance of the hash function h. The reduction just returns (m, m^*) as a collision for h.

Case 3 We will now show that in case 2 if there exists an adversary $\mathcal{A}$ against the trolling experiment, then we can use it to construct a reduction $\mathcal{R}$ against the user-controlled-traceability experiment. In particular, we have shown above that how $\mathcal{R}$ can answer all possible queries of $\mathcal{A}$ using its own oracles for the user-controlled-traceability experiment. Thus, at some point $\mathcal{A}$ will conclude and return a list OUT. We assume without loss of generality that this list does not contain any of the signatures returned as part of the CreateComment oracle queries. Note that the experiment explicitly disallows such tuples. Since we assumed that $\mathcal{A}$ wins the experiment, thus there must exist a basename dom for which $\max_{(\cdot, dom) \in S} = t > |\text{CU}| + |\text{VM}|$, where $S = \{(nym, dom) : (\cdot, nym, dom, \cdot) \in \text{OUT}\}$. However, what this implies is that there must exist exactly t valid DAA signatures in OUT for the basename dom. Let us denote those signatures under respectively pseudonyms $nym_1, \ldots, nym_t$ and messages $m_1, \ldots, m_t$ as $\sigma_1, \ldots, \sigma_t$. We also know that all $nym_1, \ldots, nym_t$ are distinct. What's more, because of instant linkability we know that there exist secret keys $sk_1, \ldots, sk_t$ for which $nym_i = \mathsf{NymGen}(sk_i, dom)$, where there is at least one secret key sk_j which was not extracted by the reduction (and put in on the revocation list by the user-controlled-traceability experiment). It follows that for the revocation list $RL = \{sk_1, \ldots, sk_{j-1}, sk_{j+1}, \ldots, sk_t\}$ we have that $\mathsf{Verify}_{\mathsf{DAA}}(pk_I, m_j, dom, \sigma_j, RL) = 1$. Thus, since we know that all signatures in OUT are not an output of the CreateComment oracle and there exists at least one valid

signature despite using a revocation list with the secret keys of all corrupted users by returning (U, σ_j, m_j, dom) for some U of a signer, the reduction wins the user-controlled-traceability experiment. The U is chosen depending on the type of forgery. If sk_j does not correspond to a secret key of any honest user (the reduction can check this asking its oracle for a signature under a dummy message for all honest users in basename dom and comparing the corresponding pseudonym with nym_j) U is chosen as an identifier of a corrupted user and otherwise as the identifier of the honest user with secret key sk_j.

Thus, we have

$$\begin{aligned}\Pr[\mathsf{Exp}^{\mathsf{troll}}_{\mathcal{A}}1^\lambda) = 1] &= 3 \cdot (\mathbf{Adv}[\mathcal{R}^{trace}_{\mathsf{DAA}}] + \mathbf{Adv}[\mathcal{R}^{\mathsf{collision}}_{h}]+ \\ &\quad + \mathbf{Adv}[\mathcal{R}^{\mathsf{VerifyBsn}}_{\mathsf{DAA}}]) + \mathbf{Adv}[\mathcal{R}^{E_1}_{\Pi}] \\ &\leq \mathsf{negl}(\lambda).\end{aligned}$$

□

7.6.3 Non-frameability

Definition 17. *We say that the system is non-frameable if for every adversary $\mathcal{A}$, every 1^λ, the probability $\Pr[\mathsf{Exp}^{\mathsf{noframe}}_{\mathcal{A}}(1^\lambda) = 1]$ is negligible 1^λ.*

Experiment $\mathsf{Exp}^{\mathsf{noframe}}_{\mathcal{A}}(1^\lambda)$**:**

CU $\leftarrow \emptyset$; HU $\leftarrow \emptyset$; $V \leftarrow \emptyset$;
USK $\leftarrow \emptyset$; COMM $\leftarrow \emptyset$; $\mathsf{ver} \leftarrow \emptyset$;
$(sk_{\mathrm{I}}, pk_{\mathrm{I}}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
$\mathcal{O} = \{\mathsf{CorruptUser}(\cdot), \mathsf{CreateHonestUser}(\cdot, \mathsf{ver}, sk_{\mathrm{I}}),$
$\mathsf{CorruptVer}(\cdot, \mathsf{ver}), \mathsf{CreateComment}(\cdot, \cdot, \cdot, pk_{\mathrm{I}}), \mathsf{JoinSystem}(\cdot, \mathsf{ver}, sk_{\mathrm{I}})\}$
$(nym^*, dom^*, m^*\gamma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, pk_{\mathrm{I}})$
Return 0 if $\mathsf{Verify}(pk_{\mathrm{I}}, nym^*, dom^*, m^*, \gamma^*) = 0$
Return 1 if:
1) there exists no tuple $(\cdot, nym^*, dom^*, m^*, \gamma^*)$ in COMM, and
2) $\exists_{U^* \in \mathrm{HU}}\ nym^* = nym_{U^*}$, where
$(nym_{U^*}, \cdot) \xleftarrow{\$} \mathsf{Comment}(pk_{\mathrm{I}}, sk_{U^*}, cred_{U^*}, dom^*, m^*)$, and $(U^*, sk_{U^*}, cred_{U^*}, \cdot) \in$ USK

Figure 7.3: Trollthrottle non-frameability

Theorem 9 (Non-frameability). *The TrollThrottle protocol (see Def. 3) satisfies Non-frameability (see Def. 16) if the underlying DAA scheme provides user-controlled traceability (Def. 13) and is instantly linkable (Def. 5), the hash function h is collision-resistant and the proof system for relation $\mathcal{R}_{\mathsf{Join}}$ is a proof of knowledge.*

Proof. Assume there exists a PPT adversary $\mathcal{A}$ that can win the non-frameability game with non-negligible probability. Then by instant linkability there $\exists dom, U \in \mathrm{HU}, nym = \mathsf{NymGen}(sk_U, dom)$, s.t. $\mathsf{Verify}(pk_{\mathrm{I}}, nym, dom, m, \gamma) = 1$ and $\neg\exists(\cdot, nym, dom, m, \gamma) \in$

COMM. In other words, the adversary is able to create a valid $\gamma = (\sigma, nym, dom, m)$ tuple that corresponds to an honest user $U \in \mathrm{HU}$. We first exclude the case that the adversary wins by using a comment from COMM under a different message m'. It is easy to see that this basically corresponds to an attack against the collision-resistance of the hash function and we can use $\mathcal{A}$ to break the security of the hash function h. We will show that in any adversary winning in any other can be used to create reduction $\mathcal{R}$ that wins the user-controlled traceability experiment for the DAA scheme with non-negligible probability.

$\mathcal{R}$ simulates the non-frameability experiment for $\mathcal{A}$ according to 7.6.2. With non-negligible probability, $\mathcal{R}$ obtains $(nym^*, dom^*, m^*, \gamma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, pk_{\mathrm{I}})$. Let $out = (U^*, \sigma^*, m^*, dom^*)$, where $\gamma^* = (\sigma^*, nym^*, dom^*, m^*)$ and U^* corresponds to the honest user with $nym^* = \mathsf{NymGen}(sk_{U^*}, dom^*)$. According to winning conditions of the non-frameability experiment we have that $\mathsf{Verify}_{\mathsf{DAA}}(pk_{I,\mathsf{DAA}}, m^*, dom^*, \sigma^*, RL_\emptyset)$ will return 1 and σ^* is not the result of any signing query made by the reduction $\mathcal{R}$. What's more, the winning conditions require that the signature corresponds to an honest user, which by instant linkability means that σ^* is also valid for the revocation list containing the secret keys of all corrupted users in CU. Thus, by returning *out*, the reduction wins the user-controlled-traceability experiment with a non-negligible probability. It follows that we have

$$\begin{aligned}\Pr[\mathsf{Exp}^{\mathsf{noframe}}_{\mathcal{A}}(1^\lambda) = 1] =& \mathbf{Adv}[\mathcal{R}^{trace}_{\mathsf{DAA}}] + \mathbf{Adv}[\mathcal{R}^{\mathsf{collision}}_{h}] + \\ & \mathbf{Adv}[\mathcal{R}^{E_1}_{\Pi}] \leq \mathsf{negl}(\lambda).\end{aligned}$$

□

7.6.4 Anonymity

Definition 18. *We say that the system is anonymous if for every adversary $\mathcal{A}$, every 1^λ, the probability* $\Pr[\mathsf{Exp}^{\mathsf{anon}}_{\mathcal{A}}(1^\lambda) = 1] = \frac{1}{2} + \mathsf{negl}(\lambda)$.

Experiment $\mathsf{Exp}^{\mathsf{anon}}_{\mathcal{A}}(1^\lambda)$**:**

CU $\leftarrow \emptyset$; HU $\leftarrow \emptyset$; $V \leftarrow \emptyset$;
USK $\leftarrow \emptyset$; COMM $\leftarrow \emptyset$; $\mathsf{ver} \leftarrow \emptyset$;
$(sk_{\mathrm{I}}, pk_{\mathrm{I}}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
$\mathcal{O} = \{\mathsf{CorruptUser}(\cdot), \mathsf{CreateHonestUser}(\cdot, \mathsf{ver}, sk_{\mathrm{I}}),$
$\mathsf{HJoinSystem}(\cdot, \mathsf{ver}, pk_{\mathrm{I}}, V), \mathsf{CreateComment}(\cdot, \cdot, \cdot, pk_{\mathrm{I}})\}$
$(U^*, dom^*, m^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, sk_{\mathrm{I}})$
$b \xleftarrow{\$} \{0, 1\}$
$\mathsf{chall} = \mathsf{Chall}_b(U^*, dom^*, m^*, sk_{\mathrm{I}}, pk_{\mathrm{I}})$
$\mathcal{O}_2 = \{\mathsf{CorruptUser}(\cdot), \mathsf{CreateHonestUser}(\cdot, \mathsf{ver}, sk_{\mathrm{I}}),$
$\mathsf{HJoinSystem}(\cdot, \mathsf{ver}, pk_{\mathrm{I}}, V), \mathsf{CreateComment}(\cdot, \cdot, \cdot, pk_{\mathrm{I}})\}$
$b^* \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_2}(\rho, sk_{\mathrm{I}}, \mathsf{chall})$
Return 0 if $(U^*, \cdot, dom^*, \cdot, \cdot) \in COMM$
Return 1 iff $b = b^*$ and $U^* \in$ HU

Figure 7.4: Trollthrottle anonymity

Theorem 10 (Anonymity). *The TrollThrottle protocol (see Def. 3) satisfies Anonymity (see Def. 18), if the underlying DAA scheme provides user-controlled anonymity (Def. 12) and the proof system for relation* $\mathcal{R}_{\mathsf{Join}}$ *is a proof of knowledge.*

Proof. Assume there exists a PPT adversary $\mathcal{A}$ that can break the anonymity game with probability $\frac{1}{2} + \varepsilon(\lambda)$ where ϵ is non-negligible in λ. We construct a PPT adversary $\mathcal{R}$ against the user-controlled-anonymity game that wins with the same probability as $\mathcal{A}$.We construct $\mathcal{R}$ as follows:

1. Experiment simulation for $\mathcal{A}$:
   - The oracles $\mathsf{CorruptUser}(\cdot), \mathsf{CreateHonestUser}(\cdot, \mathsf{ver}, sk_{\mathrm{I}})$, $\mathsf{CreateComment}(\cdot, \cdot, \cdot, pk_{\mathrm{I}})$ are simulated according to 7.6.2.
   - To answer queries to the $\mathsf{HJoinSystem}$ oracle, the reduction uses its own $\mathsf{IHJoin}(S)$ oracle and the simulator Sim to generate the proof for relation $\mathcal{R}_{\mathsf{Join}}$ (here we require the zero-knowledge property).
   - The oracle $\mathsf{Chall}_b(U, dom, m, sk_{\mathrm{I}}, pk_{\mathrm{I}})$ is simulated as follows:
     - $\mathcal{R}$ chooses some $U' \neq U$ and $(U', \cdot, \cdot, \cdot) \notin$ USK uniformly at random.
     - $\mathcal{R}$ queries it's Join oracle on U' obtaining $cred_{U'}$.
     - $\mathcal{R}$ gives $S_0 = U$ and $S_1 = U'$, m and dom to its challenger and receives the challenge $\sigma'_b = \mathsf{Sign}_{\mathsf{DAA}}(sk_{S_b}, cred_{S_b}, m, dom)$.
     - $\mathcal{R}$ sends $(\mathsf{NymExtract}(\sigma'_b), (\sigma'_b, \mathsf{NymExtract}(\sigma'_b), dom, m))$ to $\mathcal{A}$. If the challenged bit is 0, the simulation of $\mathcal{R}$ is equivalent to $\mathsf{Chall}_0(\cdot)$. Otherwise, it the challenged bit is 1, the simulation is equivalent to $\mathsf{Chall}_1(\cdot)$.
   - With non-negligible probability, $\mathcal{R}$ obtains $b_A \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, sk_{\mathrm{I}})$.

2. $\mathcal{R}$ returns $out = b_A$ and wins the user-controlled anonymity experiment if $\mathcal{A}$ wins against it's anonymity game.

Thus, by user-controlled anonymity of the DAA scheme and the zero-knowledge of the proof system we have $\Pr[\mathsf{Exp}^{\mathsf{anon}}_{\mathcal{A}}(1^\lambda) = 1] = \frac{1}{2} + \mathbf{Adv}[\mathcal{R}^{anon}_{\mathsf{DAA}}] + \mathbf{Adv}[\mathcal{R}^{\mathsf{ZK}}_{\Pi}]$. □

7.6.5 Accountability

The system shall provide accountability against censorship by allowing a participant to claim and prove that a website censored its comment. The party provides evidence that can be used to prove that an entry in the public ledger belongs to a certain message and basename. Deciding when a message is acceptable is a matter of public opinion and not modelled here.

7.6.5.1 Soundness

Definition 19. *We say that the system's accountability mechanism is sound if for every adversary $\mathcal{A}$ and 1^λ, the probability $\Pr[\mathsf{Exp}^{\mathsf{accsound}}_{\mathcal{A}}(1^\lambda) = 1]$ is negligible 1^λ.*

Experiment $\mathsf{Exp}^{\mathsf{accsound}}_{\mathcal{A}}(1^\lambda)$**:**

$\mathrm{CU} \leftarrow \emptyset$; $\mathrm{HU} \leftarrow \emptyset$; $\mathrm{VM} \leftarrow \emptyset$;
$\mathrm{USK} \leftarrow \emptyset$; $\mathrm{COMM} \leftarrow \emptyset$; $\mathsf{ver} \leftarrow \emptyset$;
$(sk_{\mathrm{I}}, pk_{\mathrm{I}}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
$\mathcal{O} = \{\mathsf{CorruptUser}(\cdot), \mathsf{CreateHonestUser}(\cdot, \mathsf{ver}, sk_{\mathrm{I}}),$
$\mathsf{CorruptVer}(\cdot, \mathsf{ver}), \mathsf{CreateComment}(\cdot, \cdot, \cdot, pk_{\mathrm{I}}), \mathsf{JoinSystem}(\cdot, \mathsf{ver}, sk_{\mathrm{I}})\}$
$(dom^*, m^*, \gamma^*, evidence^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, pk_{\mathrm{I}})$
Return 1 if all of the following hold true

- COMM contains no tuple $(\cdot, dom^*, m^*, \cdot)$
- COMM contains a tuple $(\cdot, \cdot, \cdot, \cdot, \gamma^*)$
- $\mathsf{VerifyClaim}(pk_{\mathrm{I}}, dom^*, m^*, \gamma^*, evidence^*) = 1$

Figure 7.5: Trollthrottle accountability soundness

Theorem 11 (Sound accountability). *The TrollThrottle protocol (see Def. 3) has a sound accountability mechanism (see Def. 19) if h is collision resistant.*

Proof. Assume there exists a PPT adversary $\mathcal{A}$ that can win the sound accountability game with non-negligible probability. For the case where the experiment returns 1, the adversary returns $(dom^*, m^*, \gamma^*, evidence^*)$ such that, $(\cdot, \cdot, dom^*, m^*, \cdot) \notin \mathrm{COMM}$, but, for some U', nym', dom' and m', $(U', nym', dom', m', \gamma^*) \in \mathrm{COMM}$, and (by definition of $\mathsf{VerifyClaim}$), $\gamma^* = (\sigma^*, nym^*, h(m^*), dom^*)$ for some σ^* and nym^*, as well as $\mathsf{Verify}(pk_{\mathrm{I}}, nym^*, dom^*, m^*, \gamma^*) = 1$. By definition of $\mathsf{CreateComment}$, $(nym', \gamma^*) \xleftarrow{\$} \mathsf{Comment}(pk_{\mathrm{I}}, sk_{U'}, cred_{U'}, dom', m')$, and thus by definition of $\mathsf{Comment}$, $\gamma^* = (\sigma', nym', h(m'), dom') = (\sigma^*, nym^*, h(m^*), dom^*)$. Hence, $(U', nym^*, dom^*, m', \gamma^*) \in \mathrm{COMM}$. As we assume h to be collision-resistant, $m' \neq m$ while $h(m') = h(m)$ would constitute an attack. Hence, if $\mathcal{A}'$ does not find a collision this way, then

$(U', nym^*, dom^*, m^*, \gamma^*) \in$ COMM, contradicting the assumption that no such tuple is in COMM. □

7.6.5.2 Completeness

Definition 20. *We say that the system's accountability mechanism is complete if for every adversary $\mathcal{A}$, every 1^λ, the probability* $\Pr[\mathsf{Exp}_{\mathcal{A}}^{\mathsf{acccompl}}(1^\lambda) = 1]$ *is negligible* 1^λ.

Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{acccompl}}(1^\lambda)$**:**

CU $\leftarrow \emptyset$; HU $\leftarrow \emptyset$; $V \leftarrow \emptyset$;
USK $\leftarrow \emptyset$; COMM $\leftarrow \emptyset$; $\mathsf{ver} \leftarrow \emptyset$;
$(sk_{\mathrm{I}}, pk_{\mathrm{I}}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
$\mathcal{O} = \{\mathsf{CorruptUser}(\cdot), \mathsf{CreateHonestUser}(\cdot, \mathsf{ver}, sk_{\mathrm{I}}),$
$\mathsf{CorruptVer}(\cdot, \mathsf{ver}), \mathsf{CreateComment}(\cdot, \cdot, \cdot, pk_{\mathrm{I}}), \mathsf{JoinSystem}(\cdot, \mathsf{ver}, sk_{\mathrm{I}})\}$
$(U^*, nym^*, dom^*, m^*, \gamma^*, evidence^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}}(\rho, pk_{\mathrm{I}})$
Return 1 if there exist tuples $(U^*, nym^*, dom^*, m^*, \gamma^*) \in$ COMM,
and $(U^*, sk_U, cred_U, \cdot) \in$ USK such that
$\mathsf{VerifyClaim}(nym^*, dom_i^*, m^*, \gamma^*, x) = 0$
for $x = \mathsf{Claim}(pk_{\mathrm{I}}, sk_U, cred_U, dom^*, m^*, \gamma^*, nym^*)$.

Figure 7.6: Trollthrottle accountability completeness

Theorem 12 (Completeness of accountability mechanism). *The TrollThrottle protocol (see Def. 3) has a complete accountability mechanism (see Def. 18), if correctness holds.*

Proof. This follows immediately from the definition of CreateComment (Def. 15), and the correctness of the DAA scheme (Def. 14). □

8

Accountable Javascript - Appendix

8.1 Verification of Security Properties

By default, Tamarin assumes that the adversary controls the network. Our model allows the adversary to impersonate the untrusted parties in the protocol and thereby access their secrets. This is logged with a $Corrupted(p)$ event in the trace with p an identifier for the corrupted principal.

We model the principals in the following structure using applied-π calculus.

```
in($p);
(
    ( event Corrupted($p);
      out(sk($p)) )
    | out(pk(sk($p)))
    | (
        /* process goes here */
      )
)
```

The shortcut $\$p$ denotes that the term p is a public value. The attacker, by inputting the public value $\$p$ can pick some identifier for the party. Then, if the attacker corrupts the party, a corruption event is emitted and the attacker gets access to the secret key. The public (verification) key is emitted so that other parties can use it to verify the signed messages of p. We exemplify the process with the example of $P_{Developer}$ as follows:

```
in($D);
(
    [..]
    | (
        in(<$manifest, $url, $v>);
        event DUploads($D, $url, φ);
        out(<'update', $D, $manifest, $url, $v, φ>)
        ...
      )
)
```

This code snippet includes the interaction with the network via `in` and `out`, which is represented by the attacker. The attacker hence inputs the public values *manifest*, *url* and version number v, then the developer process computes a signature φ from these values and sends an update message to the log including all that information. Events are annotations associated with the parts of the processes that enable to define restrictions and security properties. In this example, before sending the update message to the log, the developer logs a *DUploads* event in the trace, annotating the developer's new code update request to the transparency logs.

The process P_{Log} represents the transparency log as a protocol party that can receive and send messages, and in addition apply insert and lookup operations to an append-only global store. The applied-π calculus provides constructs for modelling the manipulation of a global store. The code snippet below includes an insert and a lookup operation.

```
    insert <$D, $L, 'version', $url>, $v;
```

```
...
lookup <$D, $L, 'manifest', $url> as $manifest
    in P else Q
```

The insert construct associates the value $\$v$ to the key which is a tuple $< \$D, \$L,$ '*version*', $\$url >$ and successive inserts overwrite the old values. The lookup construct retrieves the value associated with the key $< \$D, \$L,$ '*manifest*', $\$url >$ and assigns it to $\$manifest$ variable. If the lookup was successful, it proceeds with process P, otherwise with Q. $\$D$ stands for the developer's identity, whereas $\$L$ stands for the log's identity. Since there are unbounded number of developers and logs; we associate the values that are stored in the global store with the URL and the identities of the related developer and log for uniqueness.

Our model also includes lock and unlock, which the stateful applied-π calculus defines for exclusive access to the global store in the concurrent setting. The code snippet below shows an example of lock and unlock operations used in our protocol.

```
lock ($url);
insert <..., $url>, ...;
...
unlock ($url);
```

When a $\$url$ is locked, any subsequent attempt to lock the same $\$url$ will be blocked until it is unlocked. We provide exclusive accesses based on the $\$url$, when the log attempts to insert a new value to the global store. This is an over approximation: if a lock requires exclusive access independent for every write (independent of the URL) our model correctly captures this behaviour too. We do not require locks for other reads, which also increases generality.

Security properties and restrictions are first-order formulas over the annotated events and time points. Universal quantification (meaning: for all) and existential quantification (meaning: there exists) are used to check if the security property formula (lemma) holds for all examples in the domain or there exists at least one example that satisfies the formula respectively. If the lemma holds for the former case then the Tamarin Prover shows that it is proven, whereas for the latter case a satisfying example is presented to the user. The time points enable to account for event order in the trace, where e.g. $E@i$ means that event E was emitted at index i in the trace. We prove that the following security properties hold in our protocol:

Theorem 13 (Authentication of origin). *Intuitively, the client will only execute active content code (signified by the event CExec with url and manifest φ) if the code was uploaded by the honest developer D (logged the event DUploads), or the developer was corrupted. The KU event is emitted whenever the attacker (who is acting on behalf of the corrupted party D) constructs a message. We simplify the formula as follows:*

$$CExec(\$D, \$url, \varphi) \implies DUploads(\$D, \$url, \varphi) \vee (Corrupted(\$D) \wedge KU(\$url) \wedge KU(\varphi))$$

Formally, the lemma is: for all CExec events there exists either an earlier DUploads event or there exists a Corrupted($D) event and KU events before CExec event.

Theorem 14 (Transparency). *If the client executes JS code c for url with timestamp ts (CExec′), then there is a corresponding log entry (Log) and it was deemed recent (CRecent) by the client. The session identifier sid binds the moment when the client checks the timestamp is recent (CRecent) to the moment it executes (CExec′) the code.*

$$CExec'(\$url, sid, c, ts) \quad \Rightarrow \quad Log(\$url, c, ts) \quad \wedge \quad CRecent(sid, ts)$$

Authentication of origin and transparency describe the proactive behaviour of the extension. The following theorems cover the reactive behaviour. We first establish that a claim that a client submits to the public is non-repudiable, i.e. that a corrupted client cannot forge false evidence to implicate honest parties.

Theorem 15 (Accountability). *When the public accepts a claim (identified with server id, url, manifest, client nonce and log timestamp) then, even if the client was corrupted, the code must exist in the logs (Log′), and the server must have sent that data, either honestly, or dishonestly via the adversary.*

$$\begin{aligned} PAccept(\$W, \$url, \varphi, n, ts) \implies & Log'(\$url, \varphi, ts) \wedge \\ & (WSend(\$W, \$url, \varphi, n) \vee \\ & (Corrupted(\$W) \wedge KU(\$W, \$url, \varphi, n)) \end{aligned}$$

Here, the event WSend is emitted by W (who is honest) right before it sends the signed tuple sig_W to C in Fig. 4.3.

Theorem 16 (End to end guarantee). *When the client executes a malicious code, then a corrupted developer is necessary to distribute it.*

$$CExec(\$D, \$url, \text{`}malicious\text{`}) \quad \Longrightarrow \quad Corrupted(\$D)$$

Theorem 17 (End to end non-guarantee). *When the client executes a malicious code, then a corrupted developer is sufficient to distribute it.*

$$Ex.\ CExec(\$D, \$url, \text{`}malicious\text{`}) \implies (All\ x.\ Corrupted(x) \implies (x = \$D))$$

Tamarin reports these results within 3 hours on a 16-core computer with 2.6 GHz Intel Core i5 processors and 64 GB of RAM. The proof is fully automatic, but relies on a so-called 'sources' lemma to specify were certain messages can originate from. We specified this lemma by hand, but it is verified automatically. The full protocol can be found in the supplementary material [58].

8.2 Claim Verification

The public runs a procedure to verify the claim generated by a client that was allegedly targeted by a website. As detailed in the Accountable JavaScript Appendix 8.1 Theorem

15, a claim is identified with server name, URL, manifest, request nonce and the timestamp that was set for the manifest by the ledger. The signatures on the request and the response data are verified, and the request nonce is asserted with the server nonce for authenticity. Next, the delivered content behaviours are checked against the manifest using the measurement procedure. Then, the public evaluates if the manifest is the latest version on the ledger using the timestamp. If the evaluation fails in any of these steps, then the claim is accepted.

8.3 Formal model of Accountable JavaScript

```
theory AccountableJavaScript
begin

builtins: multiset

functions:
    pk/1, sk/1[private],
    sign/2, verify/3[destructor],
    true/0, eq/2,
    extractmsg/1[destructor]

equations:
    verify(pk(sk(i)), sign(sk(i), m), m ) = true,
    extractmsg(sign(skey,m))=m // only for attacker


predicates: LessThan(x,y) <=> (Ex z. y = x + z)
            , HonestRun() <=> not (Ex p #j. Corrupted(p)@j)


export queries:
"
set preciseActions = true.
"

//for the same url, the W and D must be the same
restriction webserver_dev_unique:
    "All w1 d1 w2 d2 u #i #j. WebserverDevUnique(w1,d1,u)@i &↷
     WebserverDevUnique(w2,d2,u)@j
        ==> w1=w2 & d1=d2"

//it must be the same webserver
restriction webserver_url_unique:
    "All w1 w2 u #i #j. WebserverUrlUnique(w1,u)@i & ↷
    WebserverUrlUnique(w2,u)@j
        ==> w1=w2"

//it must be the same developer
restriction dev_url_unique:
    "All d1 d2 u #i #j. DevUrlUnique(d1,u)@i & DevUrlUnique(↷
    d2,u)@j
        ==> d1=d2"
```

```
//it must be the same content if it is the same url and version
restriction content_ver_unique:
    "All u v c1 c2 #i #j. ContentVersionUnique(u,v,c1)@i & ContentVersionUnique(u,v,c2)@j
        ==> c1=c2 & #i = #j"

// ledger always honest
restriction ledger_cannot_corrupt:
    "not (Ex x #i #j. Corrupted(x)@i & LedgerE(x)@j)"


restriction log_stamp_unique:
    "All u v c d l #i #j. LogsUnique(u,v,c,d,l)@i & LogsUnique(u,v,c,d,l)@j ==> #i = #j"


restriction new_log_mono:
    "All url v1 v2 #i #j. NewLogInserted(url,v1)@i & NewLogInserted(url,v2)@j & #i < #j
        ==> LessThan(v1,v2)"



restriction parties_unique:
    "All p #i #j. Party(p)@i & Party(p)@j
        ==> #i=#j"



let Developer =
(
    in($D);
    event Party($D);
    ((
        event Corrupted($D);
        out(sk('D',$D))
    )
    |
    out(pk(sk('D',$D)))
    | (
        in(<$content,$url,$v>);
        if $content = 'malicious' then 0
        else (

            event DUploads($D,$url, $content);
            event ContentVersionUnique($url, $v, $content);
            event DevUrlUnique($D, $url);

            let mUpdate=<'update', $D, $content, $url, $v> in

            out(<mUpdate, sign(sk('D',$D), mUpdate)>);
            in(<'updateNote', $L, =mUpdate, logstamp, $ts>);
            if verify(pk(sk('L',$L)), logstamp, <$ts, sign(sk('D',$D), mUpdate)>)=true() then
```

```
                out(<'updateCode', $W, $L, mUpdate, sign(sk(↷
                'D',$D), mUpdate), logstamp, $ts>)
        )

    ))
)



let Ledger =
(
    in($L);
    event Party($L);
    event LedgerE($L);
    (
    out(pk(sk('L',$L)))
    | (
        in(<mUpdate, sigmUpdate>);
        let <'update', $D, $content, $url, $v>=mUpdate in
        if verify(pk(sk('D',$D)), sigmUpdate, mUpdate)=true()↷
         then
            in($ts);
            let logstamp = sign(sk('L',$L), <$ts, sigmUpdate>)↷
             in
            event LogStamp($url,$content,$ts);
            event LogsUnique($url, $v, $content, sigmUpdate, ↷
            logstamp);
            event NewLogInserted( $url, $v);
            //event LogStampNonEquivocation(<$url, $D, $L, ↷
            $content, $v, sigmUpdate, logstamp, $ts >);
            insert <$D, $L, 'lversion', $url>, $v;
            insert <$D, $L, 'lcontent', $url>, $content;
            insert <$D, $L, 'lsignD', $url>, sigmUpdate;
            insert <$D, $L, 'lts', $url>, $ts;
            insert <$D, $L, 'lsignL', $url>, logstamp;

            out(<'updateNote', $L, mUpdate, logstamp, $ts>)
    ))
)

let Webserver =
(
    in($W);
    event Party($W);
    (

        event Corrupted($W);
        out(pk(sk('W',$W)))
    |
    (
        (
                in(<'updateCode', =$W, $L, mUpdate, ↷
                sigmUpdate, logstamp, $ts>);
                let <'update', $D, $content, $url, $v>=mUpdate ↷
                in
```

```
                if (verify(pk(sk('L',$L)), logstamp, <$ts, sigmUpdate> )=true() &
                        verify(pk(sk('D',$D)), sigmUpdate, mUpdate )=true()) then

                    lock $url;
                    event WebserverDevUnique($W, $D, $url);
                    event WebserverUpdated( $url, $v);
                    event WCodeUpdated( $url, sigmUpdate, logstamp);
                    insert < $W, 'developer', $url>, $D;
                    insert < $W, 'ledger', $url>, $L;
                    insert < $D, $L, 'version', $url>, $v;
                    insert < $D, $L, 'content', $url>, $content;
                    insert < $D, $L, 'signD', $url>, sigmUpdate;//the signature from D
                    insert < $D, $L, 'timestamp', $url>, $ts;
                    insert < $D, $L, 'signL', $url>, logstamp;//the signature from L
                    //TODOlater the logs must be accesible by the attacker
                    unlock $url
            )
            ||
            (
                in(<'request', newReq, signewReq>);

                let <$C, =$W, $url, ~n>=newReq in

                if verify(pk(sk('C',$C)), signewReq, newReq)=true() then
                    event WebserverUrlUnique($W, $url);
                    lookup < $W, 'developer', $url> as $D in
                    lookup < $W, 'ledger', $url> as $L in
                    lookup < $D, $L, 'version', $url> as $v in
                    lookup < $D, $L, 'content', $url> as $content in
                    lookup < $D, $L, 'signD', $url> as sigmUpdate in
                    lookup < $D, $L, 'timestamp', $url> as $ts in
                    lookup < $D, $L, 'signL', $url> as logstamp in
                    //in($x_content); // superflous, but to make clear: adversary could send *any* content
                    //transparency_developer lemma will fail if webserver sends x_content instead
                    event WResponseSent( $W, $D, $url, ~n, $content);
                    event Wsent(<$W, $url, ~n, sigmUpdate, logstamp>);
                    event LearnedFromW(<$W, $url, $content, ~n>);
```

```
                        let resp = <~n, $url, $v, $content, $C, $W↷
                        , $D, $L, sigmUpdate, logstamp> in

                        out(<'response', resp, sign(sk('W',$W), ↷
                        resp), $ts>)


            )
        ))
    )

    let Client =
    (
        in($C);
        event Party($C);
        ((
            event Corrupted($C);
            out(sk('C',$C))
        )
        |
        out(pk(sk('C',$C)))
        |
        (
            new ~n; in($url);

            let newReq=<$C, $W, $url, ~n> in

            out(<'request', newReq, sign(sk('C',$C), newReq)>);


            in(<'response', resp, sigResp, $ts>);
            let < =~n, =$url, $v, $content, =$C, =$W, $D, $L, ↷
            sigmUpdate, logstamp>=resp in


            if (verify(pk(sk('W',$W)), sigResp, resp)=true() &
                verify(pk(sk('D',$D)), sigmUpdate, <'update', $D↷
                , $content, $url, $v> )=true() &
                verify(pk(sk('L',$L)), logstamp, <$ts, sigmUpdate↷
                >)=true()) then

                        event CRecent(~n, $ts);
                        event RecentResponseReceived($W, $url, ~n↷
                        , $content, $ts);
                        event HonestResponseReceived($C, $url, ~n↷
                        );
                        event CExecutes($D, $url, $content);
                        event CExecuteTime($url, ~n, $content, $ts);
                        //event End2EndGuarantee($D, $content);

                        let evidence = <~n, $url, $v, $content, $C,↷
                         $W, $D, $L, sigmUpdate, logstamp, ↷
                        sigResp> in

                        out($url);
```

```
                    out(<'claim', evidence, sign(sk('C',$C), ↷
                    evidence)>)

    ))
)


let Pub =
    (
        in($url);
        lookup < $D, $L, 'lversion', $url> as $v in
        lookup < $D, $L, 'lcontent', $url> as $content in
        lookup < $D, $L, 'lsignD', $url> as sigmUpdate in
        lookup < $D, $L, 'lts', $url> as $ts in
        lookup < $D, $L, 'lsignL', $url> as logstamp in
        in(<'claim', <~n, =$url, =$v, =$content, $C, $W, $D, ↷
        $L, =sigmUpdate, =logstamp, sigResp>, sigEvidence>);


        if (verify(pk(sk('C',$C)), sigEvidence, <~n, $url, $v,↷
         $content, $C, $W, $D, $L, sigmUpdate, logstamp, ↷
        sigResp>)=true() &
            verify(pk(sk('W',$W)), sigResp, <~n, $url, $v, ↷
            $content, $C, $W, $D, $L, sigmUpdate, logstamp> )↷
            =true() &
             verify(pk(sk('D',$D)), sigmUpdate, <'update', ↷
             $D, $content, $url, $v>)=true() &
              verify(pk(sk('L',$L)), logstamp, <$ts, ↷
              sigmUpdate>)=true() )

                then
                    //event PAccepts(<p_sigevidence, $C, $W,↷
                     $D, $L, $url,~n, p_dsig, p_lsig, p_wsig↷
                    >)
                    event PAccepts(<$W, $url, $content, ~n, $ts↷
                    >)

    )

//auto ; ~3 minutes
lemma dev_update_or_adv_provide [sources]:
    "
    All w u s1 s2 n #i. Wsent(<w,u,n,s1,s2>)@i ==>
        ((Ex #j. KU(<s1,s2>)@j & j<i ) | (Ex #j. WCodeUpdated↷
        (u,s1,s2)@j & j<i  ))
    "

//auto ; ~3 minutes
// it is possible to receive a verified response without any ↷
corrupted party
lemma sanity[output=[spthy]]:
     exists-trace
      "Ex c url n #i. HonestResponseReceived(c,url,n)@i & ↷
      HonestRun()"

//auto ; takes ~10sec
```

```
//nonrepudiation lemma
//Intuition: The client will only execute JS
//code if the code was uploaded by the honest developer ðİŘů,
//or the developer was corrupted
lemma auth_of_origin:
    " All d url content #i. CExecutes(d,url,content)@i
        ==>  (
                (Ex #j. DUploads(d,url,content)@j) |
                (Ex #k #l #m. Corrupted(d)@k & KU(url)@l & KU↷
                (content)@m )
            )
        "

//auto ; takes ~3 minutes
//Intuition: if the client executes the code it must be ↷
recent
lemma code_recent:
    " All u sid ts content #i. CExecuteTime(u, sid, content, ↷
    ts)@i
        ==> Ex #j #k.
                LogStamp(u, content, ts)@j &  j < i
                & CRecent(sid,ts)@k
     "
//auto ; takes ~5 minutes
//Intuition: If the webser sends a JS response
//(either the developer D has uploaded it or D is corrupted) ↷
and
//the JS is in the logs.
//Note also, it will fail if the webserver sends arbitrary ↷
content
lemma transparency_developer:
    " All w d u nonce content #i. WResponseSent(w,d,u,nonce,↷
    content)@i
        ==> ((Ex #j . DUploads(d,u,content)@j)
            | (Ex #k. Corrupted(d)@k) & (Ex ts #l. LogStamp(u↷
            ,content,ts)@l)

            )
        "
//auto ; takes ~3 minutes
//Intuition: if Client receives a legitimate response from W
//that means (webserver has sent it or webserver is corrupted↷
)
//and the Js is in the logs
lemma transparency_webserver:
    " All w u nonce content ts #i. RecentResponseReceived(w,u↷
    ,nonce,content,ts)@i
        ==> ((Ex d  #j. WResponseSent(w,d,u,nonce,content)@j)↷
         |
                (Ex #k. Corrupted(w)@k )) & (Ex #l. LogStamp(↷
                u,content,ts)@l & l<i)

        "

//auto ; takes ~15 seconds
//stronger non-repudiation:
```

```
// Intuition: if public accepts a Js code reception,
// (either the code was learned from W or W is corrupted and adversary computed w,u,c,n by itself)
// and the Js is in the logs
lemma public_accept_claim_only_if_W_sent:
    "
        All w u content n ts #i . PAccepts(<w,u,content,n,ts>)@i

                ==>  (( Ex #j. LearnedFromW(<w,u,content,n>)@j & j <i ) |
                    ((Ex #m. Corrupted(w)@m & m < i)
                    & (Ex #k1. KU(<w,u,content,n>)@k1 & k1<i ) ) ) & (Ex #k2. LogStamp(u,content,ts)@k2 & k2<i  )
    "


lemma e2eguarantee:
    " All d u #i. CExecutes(d,u, 'malicious')@i
        ==> (Ex #k. Corrupted(d)@k)
        "

// verified
// corrupting D is sufficient to distribute malicious code
lemma e2eNonguarantee:
    exists-trace
    " Ex d u #i. CExecutes( d, u, 'malicious')@i
      & (All #k x. Corrupted(x)@k ==> (x = d ))
        "



process:
    !Developer | !Webserver | !Client | Pub | !Ledger

end



end
```

Listing 8.1: Formal model of Accountable JavaScript protocol

8.4 Formal model of Code Verify

```
theory CodeVerify
begin

builtins: multiset, hashing

functions:
    h/1, true/0, eq/2,
    pk/1, sk/1[private],
```

```
    sign/2, verify/3[destructor],
    chan/2[private]

equations:
    verify(pk(sk(i)), sign(sk(i), m), m ) = true

predicates: HonestRun() <=> not (Ex p #j. Corrupted(p)@j)

options: asynchronous-channels

export queries:
"
set preciseActions = true.
"

restriction parties_unique:
    "All p #i #j. Party(p)@i & Party(p)@j
        ==> #i=#j"


let Developer = (

    ((
        event Corrupted('D');
        out(sk('D'))
    )
    |
    out(pk(sk('D')))
    | (
            in(<$content, $v>);
            if $content = 'malicious' then 0
            else (
                event DUploads(< $content, $v, h($content)>);
                let mUpdate = <'update', h($content), $v > in

                out(<mUpdate, sign(sk('D'), mUpdate)>);
                in(<'updateNote', =mUpdate, cf_sign>);
                if verify(pk(sk('CF')), cf_sign, sign(sk('D')↷
                , mUpdate) )=true() then
                    let codeUpdate = <'updateCode', $content, ↷
                    $v > in
                    out(<codeUpdate, sign(sk('D'), codeUpdate↷
                    )> )
            )
       )
    )

)

let CloudFlare =
(

    ((
        event Corrupted('CF');
        out(sk('CF'))
    )
```

```
    |
    out(pk(sk('CF')))
    | (
            in(<mUpdate, sigmUpdate>);
            let <'update', hash_content, $v>=mUpdate in
            if verify(pk(sk('D')), sigmUpdate, mUpdate)=true
            () then
                event HashUpdated(<$v, hash_content>);
                let cf_sign = sign(sk('CF'), sigmUpdate ) in
                out(<'updateNote', mUpdate, cf_sign >);
                !(
                    in(chan($C,'CF'),<'hashRequest', $C, =$v
                    >);
                    event HashResponse(<$C, $v, hash_content
                    >);
                    out(chan($C,'CF'),<'hashResponse', $C,
                    hash_content, $v>)
                )
    ))
)

let Webserver =
(

    ((
        event Corrupted('W');
        out(sk('W'));
        !(in($C); out(chan($C,'W')))
    )
    |
    out(pk(sk('W')))
    | (
            in(<codeUpdate, sigCodeUpdate>);
            let <'updateCode', $content, $v>= codeUpdate in
            if verify(pk(sk('D')), sigCodeUpdate, codeUpdate
            )=true() then
                event CodeUpdated(<$content, $v>);
                !(
                    in(chan($C,'W'),<'httpRequest', $C>);
                    event WResponseSent(< $C, $content, $v >);
                    out(chan($C,'W'),<'httpResponse', $content
                    , $v>)
                )

    ))
)

let Client  =
(
    in($C);
    event Party($C);
    ((
        event Corrupted($C);
        out(sk('C',$C));
        out(chan($C,'W'))
    )
```

```
    |
    out(pk(sk('C',$C)))
    | (
        out(chan($C,'W'),<'httpRequest', $C>);
        in(chan($C,'W'),<'httpResponse', $content, $v >);

        out(chan($C,'CF'),<'hashRequest', $C, $v >);
        in(chan($C,'CF'),<'hashResponse', =$C, hash_content,↷
         =$v>);

        if  hash_content = h($content) then
            event CodeVerified(< $C, $content, $v, ↷
            hash_content>);
            let evidence = <$C, $content, $v, hash_content > ↷
            in

            out(<evidence, sign(sk('C', $C), evidence)>)


    ))
)

let Pub =
(

    in(<evidence, sigEvidence>);
    let < $C, $content, $v, hash_content >= evidence in
    if ( verify(pk(sk('C', $C)), sigEvidence, evidence )=true↷
    () &
                hash_content = h($content) ) then

        event PAccepts(<$C, $content, $v, hash_content >)
)


// Reactivated: we need this because otherwise chan(..) ↷
forces partial reconstructions
// verified
lemma hash_resp_provide [sources]:
    "
    All c v hsh #i. HashResponse(<c,v,hsh>)@i ==>
        ((Ex #j. KU(hsh)@j & j<i ) | (Ex #j. HashUpdated(<v,↷
        hsh>)@j & j<i  ))
    "

//auto
lemma sanity[output=[spthy]]:
     exists-trace
      "Ex c  #i. CodeVerified(c)@i & HonestRun()"

// verifies ~1min
lemma auth_of_origin:
    " All c v con hsh #i. CodeVerified(<c, con, v, hsh>)@i
        ==>  (
                (Ex #j. DUploads(<con, v, hsh>)@j) |
```

```
                (Ex #k #l #m. Corrupted('D')@k & KU(v)@l & KU↷
                (con)@m )  |
                (Ex #k . Corrupted('CF')@k  )
            )
        "



// corrupting D is necessary to distribute malicious code, (W↷
 and CF restricted to corrupt together)
// auto <1min
lemma e2eguarantee:
    " All c v hsh #i. CodeVerified(<c, 'malicious', v, hsh>)↷
    @i
        ==> (Ex #k. Corrupted('D')@k)
            | (Ex #k . Corrupted('CF')@k  )
        "

// auto, quick
// corrupting D is sufficient to distribute malicious code
lemma e2eNonguarantee:
    exists-trace
    " Ex c v hsh #i. CodeVerified(< c, 'malicious', v, hsh>)↷
    @i
      & (All #k x. Corrupted(x)@k ==> (x = 'D' ))
        "

// auto, quick
// non-accountability.
// The client can fake the public's information, as the TLS ↷
channels does not
// provide non-repudiation
lemma public_accept_claim_only_if_W_sent:
    exists-trace
    "
    not(
        All c v con hsh #i . PAccepts(<c,con,v,hsh>)@i

                ==>  (( Ex #j. WResponseSent(<c, con, v>)@j &↷
                 j <i )
                | (Ex #m. Corrupted('W')@m & m < i)
                | (Ex #m. Corrupted('CF')@m & m < i)
                      ))
    "

process:
    !Developer | !Webserver | !CloudFlare |  !Client | Pub

end


end

  /* hash_resp_provide (all-traces): verified (49 steps) */
  /* sanity (exists-trace): verified (9 steps) */
  /* auth_of_origin (all-traces): verified (774 steps) */
```

```
/* e2eguarantee (all-traces): verified (630 steps) */
/* e2eNonguarantee (exists-trace): verified (7 steps) */
/* public_accept_claim_only_if_W_sent (exists-trace): ↷
verified (6 steps) */
```

Listing 8.2: Formal model of Code Verify protocol

8.5 Evaluation Details

We provide details about evaluation of each case study as follows.

8.5.1 'Hello World' Application Scenario

We evaluate the sample web application (Listing 4.3) using our browser extension. The manifest file in Listing 4.4 is produced automatically. The extension fetches and verifies the signed manifest linked in the response headers in parallel to collecting the metadata on the *inline* script. Signature validation, meta data collection and the final compliance check succeed with imperceptible impact on performance with Accountable JS. We evaluated the same scenario by enabling CSP and disabling Accountable JS. We firstly deployed a detailed CSP header that is close to Accountable JS manifests, namely includes the valid sources for scripts and provides the hashes for the script and event handler resources.

```
add_header content-security-policy "default-src 'self'; ↷
script-src 'self' 'sha256-AfuyZ600rkX8AD+xANHUProHJm+22↷
Tp0bMnvPFk/vas='; object-src 'none'";
```

We compared the CSP results with the Accountable JS results and the difference is very small.

8.5.2 Self-Contained Web Application Scenario

This case study shows that our browser extension is compatible with WhatsApp's web client. All active components are hosted on `web.whatsapp.com`, it is thus easy to generate and maintain the manifest file. We observed that URLs for some external scripts include parts of their content's hash, likely related to caching optimisations. As any change to the content of an external script requires a new manifest file anyway, this is not a concern for our proposal. Moreover, `generate_manifest` can automatically generate the manifest file.

For measurement, we firstly mirrored the HTML page WhatsApp provides, locally to add integrity attributes for scripts. However we observed that some dynamic behaviours (e.g. script add and delete dynamically) did not exactly take place in local. Hence, we evaluated the public website with integrity attributes for scripts stored hardcoded in the extension's content script (that collects metadata) and a hardcoded response header `x-acc-js-link` directed to a local URL that provides the signed manifest file. The

WhatsApp Web application consists of nine *external* and four *inline* scripts present in the initial HTML. No more active content is appended after the window's load event, however, some external scripts are removed later. These removals occur during the measurement process, hence the automatically generated manifest successfully marks these external scripts with the *persistent* attribute and the others without it. The WhatsApp website is incompatible with Lighthouse tool, it shows a banner during the evaluation and the active contents are not delivered. Therefore, we use Puppeteer's Page metrics for evaluation.

The traffic requirements are modest (about 1.17 bytes) and incur only modest blocking time. The performance overhead for combined duration of all tasks performed by the browser for baseline is 204ms, whereas it is 220ms for Code Verify and 244ms for Accountable JS. The difference between Code Verify and Accountable JS is very small. This is remarkable, because Code Verify only applies SRI checks on external scripts but not event handlers or iframes. In contrast to Accountable JS, the order of active elements is ignored, attributes are not checked (e.g. `load='async'` for scripts) and a short hash value is downloaded from Cloudflare, rather than a signature.

8.5.3 Trusted Third-party Code Scenario

This case study shows that our prototype is compatible with third party active contents and dynamic modification to the DOM. Again, the extension captures the jQuery code and the inline JS, and automatically generates a manifest. Note that the jQuery library removes the effectuating active content from the DOM, after interpreting the code and displaying the message in the window. It removes the script element in Listing 4.1 and leaves only a non-executable message:

```
<html>
    <head>
        <script src="https://googleapis../jquery-3.6.1.js">
        </script>
    </head>
    <body>Hello World</body>
</html>
```

The measurement captures the inline script although it is removed and successfully compares the active content list against the manifest. The additional network traffic is almost identical to the simpler 'Hello World' case. Likewise, the performance overhead is imperceptible. The difference between CSP overheads and Accountable JS overheads is very small again.

8.5.4 Delegate Trust to Third Parties Scenario

Nimiq's Wallet application behaves differently when the user does not have the Nimiq credentials stored in the browser and when the user has credentials. It contains Hub in an iframe and Hub contains Keyguard in an iframe if the user does not have an account yet. If the user has an account, the Hub does not embed the Keyguard. We generated a manifest for Wallet that covers both cases by declaring the Keyguard with $dynamic = true$. In this way, the Keyguard may or may not be delivered. The browser

extension will take the Keyguard into account if it is delivered; otherwise it will ignore it. For this case study, we created a user account using Nimiq Hub prior the evaluation; hence the Keyguard is not embedded inside the Hub.

We created a simple shopping cart application that uses Nimiq's Wallet in an iframe. The first-party inline scripts communicate with the third party code in the iframe via postMessages. As shown in Listing 4.2, the inline script transmits a transaction record to the Nimiq Wallet. In the main window's manifest (Listing 4.5), the inline elements have *trust* = *assert*, while trust for the third party iframe (Wallet) is delegated to Nimiq. Hence Nimiq's server is in charge of delivering a separate signed manifest for its content. The main window's manifest was automatically generated, but we changed the *trust* attribute for Wallet to *delegate*.

The manifest for Nimiq's Wallet (Listing 4.7) has seven *external* scripts and one iframe that is Hub. We chose to assert trust for the external scripts via cryptographic hashes. For the Hub iframe, the manifest section has a nested manifest attribute that includes the manifest sections for the active contents inside the Hub (six *external* scripts, one *inline* script and the iframe for the Keyguard). For the KeyGuard iframe, being the most critical component, the manifest section has also a nested manifest attribute that includes the manifest sections for the active contents inside the KeyGuard document.[1] Our prototype successfully evaluates the active content list against the manifests. For efficiency, the prototype measures all elements simultaneously and combines their results once computed. The browser extension panel displays result instantaneously, while the evaluation is in progress, although the evaluation is usually too quick for the user to notice. The traffic requirements are about 9.9 kB for the additional signature. In terms of performance, the impact of Accountable JS is again imperceptible.

In the CSP evaluation for this case study, we defined CSP headers for the main website, Wallet, Hub and Keyguard. Accountable JS performance results are again close to CSP except the total blocking time is slightly higher than the CSP. Besides, the reaction time unexpectedly decreases more with Accountable JS. However, the difference in reaction time is minimal and could possibly be explained by a) network latency, (b) side effects of the browser's just-in-time compilation or scheduling or (c) a side effect of the former two on how Lighthouse evaluates the reactive metric. Nimiq is a complex web application heavily dependent on external data, in particular the remote blockchain it connects.

8.5.5 Untrusted Third-Party Code

For Google AdSense, we tested with the active content of the framework, but no ads imported, as we could not obtain an account with the provider.

As in the previous case, we integrate the high-security code (Nimiq's Wallet) as a sandbox and include third party code AdSense with *trust* = *blind* − *trust*. The web application, including AdSense, is functional and our prototype shows compliance with the manifest. The network overhead is comparable to Nimiq A, albeit slightly higher.

[1] We used both, for demonstrations in Section 4.4. `generate_manifest` produces all variants, but the nested manifest has precedence during the compliance checks.

This is due to the larger size of the manifest, which now includes AdSense as well. Nevertheless, the performance overhead is imperceptible with Accountable JS.

8.5.6 Compartmentalisation of Code and Development Process

As we stated before in Accountable JavaScript Appendix Section 8.5.4, Nimiq's Wallet behaves differently when the user does not have an account stored in the browser yet. Furthermore, it redirects to Nimiq Hub from Wallet, when there is no account. In this case study, we didn't create an account and since we have the local copy of Nimiq framework, we slightly updated the Nimiq's code to prevent the redirection, to have the compartmentalisation case study (Wallet includes Hub in iframe and Hub further includes Keyguard in an iframe).

We evaluate the performance impact of compartmentalisation slightly differently. We consider Nimiq's Hub application, which includes the KeyGuard application with *trust* = *delegate* and thus requires a separate manifest for the KeyGuard. This time, a separate signing key is used for the KeyGuard manifest. For the baseline performance, we inline the KeyGuardâĂŹs manifest as an entry for its iframe in the Hub's manifest, thus having one manifest and one signing key. In contrast to the other case studies, the extension is activated in the baseline measurement, too.

We observe that there are two more round trips and noticeably higher traffic overhead (about the overhead of Accountable JS, not the overall page traffic of 4,6 MB). This is due to downloading the extra SXG certificate and manifest. The effect on the rendering metrics is small. However there is a barely noticeable increase in reaction time. Nevertheless, this can again be explained with network latency and side effects described above in Accountable JavaScript Appendix Section 8.5.4.

Bibliography

Other references

[1] *A URL for Blob and MediaSource reference*. W3C Working Draft. 2023. URL: https://www.w3.org/TR/FileAPI/#url.

[2] *AdSense Program Policies*. 2021. URL: https://support.google.com/adsense/answer/48182?amp;stc=aspe-1pp-en (visited on 07/19/2021).

[3] Alexa Internet, Inc. *Top Sites in Germany*. Online. 2019. URL: https://www.alexa.com/topsites/countries/DE.

[4] *Alexa News*. 2019. URL: https://www.alexa.com/topsites/category/Top/News.

[5] Alexandre, A. *MEGA Chrome Extension Compromised to Steal Users' Monero*. https://cointelegraph.com/news/mega-chrome-extension-compromised-to-steal-users-monero. Sept. 2019.

[6] *Alloy 6.0.0*. 2021. URL: https://github.com/AlloyTools/org.alloytools.alloy/releases/tag/v6.0.0.

[7] *Alloy Analyser*. URL: https://alloytools.org/.

[8] *Amazon EC2 on-demand pricing*. URL: https://aws.amazon.com/ec2/pricing/on-demand/.

[9] Asokan, N., Shoup, V., and Waidner, M. Asynchronous protocols for optimistic fair exchange. In: *S&P'98*. IEEE Comp. Soc., 1998, 86–99.

[10] Azimpourkivi, M., Topkara, U., and Carbunar, B. A secure mobile authentication alternative to biometrics. In: *Proc. of the 33rd ACSAC*. ACSAC 2017. ACM, Orlando, FL, USA, 2017, 28–41. URL: http://doi.acm.org/10.1145/3134600.3134619.

[11] Baldimtsi, F. and Lysyanskaya, A. Anonymous credentials light. In: *(CCS'13)*. ACM, 2013, 1087–1098.

[12] Barreto, P. S. L. M. and Naehrig, M. Pairing-friendly elliptic curves of prime order. In: *Selected Areas in Cryptography*. Ed. by Preneel, B. and Tavares, S. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, 319–331.

[13] Barth, A. *The Web Origin Concept*. RFC 6454. http://www.rfc-editor.org/rfc/rfc6454.txt. RFC Editor, Dec. 2011. URL: http://www.rfc-editor.org/rfc/rfc6454.txt.

[14] Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., and Tian, Y. Run-time monitoring and formal analysis of information flows in chromium. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. URL: https:

//www.ndss-symposium.org/ndss2015/run-time-monitoring-and-formal-analysis-information-flows-chromium.

[15] Benndorf, V. and Normann, H.-T. The willingness to sell personal data. *The Scandinavian Journal of Economics* 120, 4 (2018), 1260–1278.

[16] Berners-Lee, T., Fielding, R. T., and Masinter, L. *Absolute URI*. STD 66. http://www.rfc-editor.org/rfc/rfc3986.txt. RFC Editor, Jan. 2005. URL: https://datatracker.ietf.org/doc/html/rfc3986#section-4.3.

[17] Berners-Lee, T., Fielding, R. T., and Masinter, L. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. http://www.rfc-editor.org/rfc/rfc3986.txt. RFC Editor, Jan. 2005. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[18] Bernhard, D., Fuchsbauer, G., Ghadafi, E., Smart, N. P., and Warinschi, B. Anonymous attestation with user-controlled linkability. *Int. J. Inf. Secur.* 12, 3 (2013), 219–249.

[19] Bhargavan, K., Delignat-Lavaud, A., and Maffeis, S. Language-based defenses against untrusted browser origins. In: *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., Aug. 2013, 653–670. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/bhargavan.

[20] Bielova, N., Devriese, D., Massacci, F., and Piessens, F. Reactive non-interference for a browser model. In: Cited by: 38; All Open Access, Green Open Access. 2011, 97–104. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-81055139598&doi=10.1109%2fICNSS.2011.6059965&partnerID=40&md5=24982a29cf90e1f78fb3944f4f0e607f.

[21] *Big query reddit dataset*. URL: https://bigquery.cloud.google.com/dataset/fh-bigquery:reddit_comments.

[22] Bohannon, A. Foundations of webscript security. PhD thesis. University of Pennsylvania, 2012. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=699295d8e46892582ece122ceb0328ee52a3b493.

[23] Bohannon, A. and Pierce, B. C. Featherweight firefox: formalizing the core of a web browser. In: *USENIX Conference on Web Application Development (WebApps 10)*. USENIX Association, June 2010. URL: https://www.usenix.org/conference/webapps-10/featherweight-firefox-formalizing-core-web-browser.

[24] Brickell, E. F., Camenisch, J., and Chen, L. Direct anonymous attestation. In: *CCS'04*. ACM, 2004, 132–145.

[25] Brickell, E., Chen, L., and Li, J. A new direct anonymous attestation scheme from bilinear maps. In: *Trusted Computing-Challenges and Applications*. Springer, 2008, 166–178.

[26] Brickell, E., Chen, L., and Li, J. Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *Int. J. Inf. Secur.* 8, 5 (2009), 315–330.

[27] Brickell, E. and Li, J. A pairing-based daa scheme further reducing tpm resources. In: *Trust and Trustworthy Computing*. Ed. by Acquisti, A., Smith, S. W., and Sadeghi, A.-R. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, 181–195.

[28] *BrowserStack*. URL: https://www.browserstack.com/.

[29] BSI. *Technical Guideline TR-03110 v2.21 – Advanced Security Mechanisms for Machine Readable Travel Documents and eIDAS Token*. 2016. URL: https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html.

[30] Bugliesi, M., Calzavara, S., and Focardi, R. Formal methods for web security. *Journal of Logical and Algebraic Methods in Programming* 87 (2017), 110–126. URL: https://www.sciencedirect.com/science/article/pii/S2352220816301055.

[31] Bugliesi, M., Calzavara, S., Focardi, R., and Khan, W. Cookiext: patching the browser against session hijacking attacks. *Journal of Computer Security* 23, 4 (2015). Cited by: 33, 509–537. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84944273601&doi=10.3233%2fJCS-150529&partnerID=40&md5=450e08f72972da184fd6c6f1523b8ae5.

[32] Camenisch, J., Drijvers, M., Dzurenda, P., and Hajny, J. Fast keyed-verification anonymous credentials on standard smart cards. In: *ICT Systems Security and Privacy Protection (SEC 2019)*. Springer, 2019, 286–298.

[33] Camenisch, J., Hohenberger, S., Kohlweiss, M., Lysyanskaya, A., and Meyerovich, M. How to win the clonewars: efficient periodic n-times anonymous authentication. In: *CCS'06*. ACM, 2006, 201–210.

[34] Carlin, D., Burgess, J., O'Kane, P., and Sezer, S. You could be mine(d): the rise of cryptojacking. *IEEE Secur. Priv.* 18, 2 (2020), 16–22. URL: https://doi.org/10.1109/MSEC.2019.2920585.

[35] Carroll, O. *St.Petersburg troll farm to influence US election campaign*. 2017. URL: https://www.independent.co.uk/news/world/europe/russia-us-election-donald-trump-st-petersburg-troll-farm-hillary-clinton-a8005276.html.

[36] Castro, M., Liskov, B., et al. Practical byzantine fault tolerance. In: *OSDI*. Vol. 99. 1999, 173–186.

[37] Chang, B., Kesselman, J., and rahman, r. *Document Object Model (DOM) Level 3 Validation Specification*. Editors, W3C Recommendation. 2004. URL: http://www.w3.org/TR/DOM-Level-3-Val/ (visited on 04/23/2021).

[38] Chen, L., Morrissey, P., and Smart, N. *DAA: Fixing the pairing based protocols*. Cryptology ePrint Archive, Report 2009/198, Withdrawn. 2009.

[39] Chen, L. *A DAA Scheme Requiring Less TPM Resources*. Cryptology ePrint Archive, Report 2010/008. 2010.

[40] Chen, S., Meseguer, J., Sasse, R., Wang, H. J., and Wang, Y.-M. A systematic approach to uncover security flaws in gui logic. In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. 2007, 71–85.

[41] Chen, X. and Feng, D. A new direct anonymous attestation scheme from bilinear maps. In: *Young Computer Scientists, 2008. ICYCS 2008*. IEEE. 2008, 2308–2313.

[42] Cimpanu, C. *Chrome extension caught hijacking users' search engine results*. 2019. URL: https://www.zdnet.com/article/chrome-extension-caught-hijacking-users-search-engine-results/ (visited on 11/04/2021).

[43] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In: *Computer Aided Verification*. Ed. by Emerson, E. A. and Sistla, A. P. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, 154–169.

[44] Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. Model checking and the state explosion problem. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Meyer, B. and Nordio, M. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, 1–30. URL: https://doi.org/10.1007/978-3-642-35746-6_1.

[45] *Content Scripts*. 2021. URL: https://developer.chrome.com/docs/extensions/mv3/content_scripts/ (visited on 04/23/2021).

[46] Corrigan-Gibbs, H. and Ford, B. Dissent: accountable anonymous group messaging. In: *CCS'10*. ACM. 2010, 340–350.

[47] *Crypto-js documentation*. URL: https://www.npmjs.com/package/crypto-js.

[48] *Department of Justice : Charges in case 1:18-cr-00032-DLF*. URL: https://www.justice.gov/file/1035477/download.

[49] DigiCert. *Get your Signed HTTP Exchanges certificate*. 2021. URL: https://docs.digicert.com/manage-certificates/certificate-profile-options/get-your-signed-http-exchange-certificate/ (visited on 11/05/2021).

[50] Dingledine, R., Mathewson, N., and Syverson, P. Reputation in p2p anonymity systems. In: *Workshop on economics of peer-to-peer systems*. Vol. 92. 2003.

[51] *Document write*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document/write.

[52] *Domain names - implementation and specification*. RFC 1035. Nov. 1987. URL: https://www.rfc-editor.org/info/rfc1035.

[53] Eastlake, D. et al. *Transport layer security (TLS) extensions: Extension definitions*. RFC 6066. 2011. URL: https://www.rfc-editor.org/rfc/rfc6066.txt.

[54] Eén, N. and Sörensson, N. An extensible sat-solver. In: *International Conference on Theory and Applications of Satisfiability Testing*. 2003. URL: https://api.semanticscholar.org/CorpusID:9774288.

[55] Eicholz, A., Moon, S., Danilo, A., Leithead, T., and Faulkner, S. *Sandboxing*. W3C Recommendation. Jan. 2021. URL: https://www.w3.org/TR/2021/SPSD-html52-20210128/browsers.html.

[56] Eijdenberg, A., Laurie, B., and Cutter, A. *Verifiable Data Structures*. 2015. URL: https://github.com/google/trillian/blob/b7ea8d2ca870e5b8ae1c05e9d2a33c4fdcca4580/docs/papers/VerifiableDataStructures.pdf (visited on 11/05/2021).

[57] Esiyok, I., Berrang, P., Cohn-Gordon, K., and Künnemann, R. Accountable javascript code delivery. In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3,*

2023. The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/accountable-javascript-code-delivery/.

[58] Esiyok, I., Berrang, P., Gordon, K.-C., and Kuennemann, R. *Accountable Javascript Supplementary Material*. 2023. URL: https://github.com/iesiyok/accountable-js.

[59] Esiyok, I., Hanzlik, L., Künnemann, R., Budde, L. M., and Backes, M. TrollThrottle —Raising the Cost of Astroturfing. In: *Applied Cryptography and Network Security*. Ed. by Conti, M., Zhou, J., Casalicchio, E., and Spognardi, A. Vol. 12147. Lecture Notes in Computer Science. Springer International Publishing, Cham, 2020, 456–476. URL: https://link.springer.com/10.1007/978-3-030-57878-7_22 (visited on 05/19/2021).

[60] Esiyok, I., Nemati, H., and Kuennemann, R. *Formal Browser model for Security Analysis Supplementary Material*. Github. 2023. URL: https://github.com/model-evaluator/model-evaluator/tree/maven.

[61] Etim, B. *Why No Comments? It's a Matter of Resources*. 2017. URL: https://www.nytimes.com/2017/09/27/reader-center/comments-moderation.html.

[62] *Facebook Messenger*. https://www.messenger.com/. Accessed on: [2023-08-23].

[63] Ferrara, E., Varol, O., Davis, C., Menczer, F., and Flammini, A. The rise of social bots. *Commun. ACM* 59, 7 (June 2016), 96–104. URL: http://doi.acm.org/10.1145/2818717.

[64] Fielding, R., Notthingham, M., and Reschke, J. *HTTP Semantics*. URL: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-15.txt#section-8 (visited on 07/14/2021).

[65] Franz, M. E unibus pluram: massive-scale software diversity as a defense mechanism. In: *Proceedings of the 2010 New Security Paradigms Workshop*. NSPW '10. Association for Computing Machinery, New York, NY, USA, Sept. 21, 2010, 7–16. URL: https://doi.org/10.1145/1900546.1900550 (visited on 07/22/2021).

[66] Gasarch, W. Private information Retrieval Survey (), 31. URL: https://crypto.stanford.edu/~dabo/courses/cs355_fall07/pir.pdf.

[67] Gemalto. *The electronic passport in 2018 and beyond*. June 2018. URL: https://www.gemalto.com/govt/travel/electronic-passport-trends.

[68] Goldreich, O. and Ostrovsky, R. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (May 1, 1996), 431–473. URL: https://doi.org/10.1145/233551.233553 (visited on 07/20/2021).

[69] Grier, C., Tang, S., and King, S. T. Designing and implementing the op and op2 web browsers. *ACM Transactions on the Web* 5, 2 (2011). Cited by: 18. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-80052072472&doi=10.1145%2f1961659.1961665&partnerID=40&md5=592428561b181d34e988300a22c94a6b.

[70] Groß, T., Pfitzmann, B., and Sadeghi, A.-R. Browser model for security analysis of browser-based protocols. In: *Computer Security – ESORICS 2005*. Ed. by

Vimercati, S. d. C. di, Syverson, P., and Gollmann, D. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, 489–508.

[71] Groth, J. and Kohlweiss, M. One-out-of-many proofs: or how to leak a secret and spend a coin. In: *EUROCRYPT 2015*. Springer, 2015, 253–280.

[72] Groth, J., Ostrovsky, R., and Sahai, A. Perfect non-interactive zero knowledge for NP. In: *EUROCRYPT 2006*. Ed. by Vaudenay, S. Vol. 4004. LNCS. Springer, Heidelberg, 2006, 339–358.

[73] Group, T. C. *Main Specification Version 1.1b*. 2001. URL: https://trustedcomputinggroup.org/tpm-main-specification/.

[74] Gu, L., Ding, X., Deng, R. H., Xie, B., and Mei, H. Remote attestation on program execution. In: *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*. STC '08. Association for Computing Machinery, New York, NY, USA, Oct. 31, 2008, 11–20. URL: https://doi.org/10.1145/1456455.1456458 (visited on 07/21/2021).

[75] Gupta, A. *The Real Twitter Files: The Algorithm*. URL: https://www.news.aakashg.com/p/the-real-twitter-files-the-algorithm (visited on 04/02/2023).

[76] Hansen, R. and Silveira, V. *Code Verify : An open source browser extension for verifying code authenticity on the web*. 2022. URL: https://engineering.fb.com/2022/03/10/security/code-verify/ (visited on 03/10/2022).

[77] Hegelich, S. and Janetzko, D. Are social bots on twitter political actors? empirical evidence from a ukrainian social botnet. In: *Tenth Int. AAAI*. 2016. URL: https://ojs.aaai.org/index.php/ICWSM/article/view/14764/14613.

[78] Herlihy, M. and Moir, M. Enhancing accountability and trust in distributed ledgers (June 2016).

[79] Holt, J. E. and Seamons, K. E. Nym: practical pseudonymity for anonymous networks. *Internet Security Research Lab Technical Report* 4 (2006), 1–12.

[80] Howard, P., Kollanyi, B., and Woolley, S. C. *Bots and automation over Twitter during the second US presidential debate*. Tech. rep. Political Bots, 2016.

[81] *HTML Living Standard: 8.1.8.2 Event handlers on elements, Document objects, and Window objects*. URL: https://html.spec.whatwg.org/#event-handlers-on-elements,-document-objects,-and-window-objects.

[82] *HTML Living Standard: Dynamic Markup Insertion: document.write()*. Online. Accessed on: [2023-07-23]. URL: https://html.spec.whatwg.org/multipage/dynamic-markup-insertion.html#dom-document-write-dev.

[83] Huang, L., Joseph, A. D., Nelson, B., Rubinstein, B. I., and Tygar, J. Adversarial machine learning. In: *Proc. of the 4th ACM workshop AISec*. ACM. 2011, 43–58.

[84] ICAO. *Machine Readable Travel Documents - Part 11: Security Mechanism for MRTDs*. Doc 9303. 2015.

[85] *Is it allowed to use Iframe*. 2020. URL: https://support.google.com/adsense/thread/24384322/is-it-allowed-to-use-iframe?hl=en (visited on 11/05/2021).

[86] *Javers v7.3.2*. 2023. URL: https://javers.org/.

[87] Johnson, P. C., Kapadia, A., Tsang, P. P., and Smith, S. W. Nymble: anonymous ip-address blocking. In: *Intern. Workshop on PETS*. Springer. 2007, 113–133.

[88] Kobeissi, N. *An Analysis of the ProtonMail Cryptographic Architecture*. Cryptology ePrint Archive, Report 2018/1121. 2018. URL: https://eprint.iacr.org/2018/1121.

[89] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: exploiting speculative execution. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, 1–19.

[90] Krawczyk, H. Cryptographic extraction and key derivation: the hkdf scheme. In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Rabin, T. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, 631–648.

[91] Kreil, M. *Social Bots, Fake News und Filterblasen*. 2017. URL: https://en.wikipedia.org/wiki/List_of_newspapers_by_circulation.

[92] Kremer, S. and Künnemann, R. Automated analysis of security protocols with global state. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. IEEE Computer Society, Washington, DC, USA, 2014, 163–178. URL: https://doi.org/10.1109/SP.2014.18.

[93] Kremer, S. and Künnemann, R. Automated analysis of security protocols with global state. In: *S&P'14*. IEEE Comp. Soc., 2014, 163–178.

[94] Kruisselbrink, M. *The Blob Interface and Binary Data*. W3C Working Draft. 2023. URL: https://w3c.github.io/FileAPI/#blob-section.

[95] Kumar, S., Cheng, J., Leskovec, J., and Subrahmanian, V. An army of me: sockpuppets in online discussion communities. In: *Proc. of the 26th Int. Conf. on WWW*. 2017, 857–866.

[96] Kuran, T. and Sunstein, C. R. Availability cascades and risk regulation. *Stan. L. Rev.* 51 (1998), 683.

[97] *LastPass: About Password Iterations*. URL: https://support.logmeininc.com/lastpass/help/about-password-iterations-lp030027.

[98] Laurie, B. and Kasper, E. *Revocation Transparency*. 2014. URL: https://www.links.org/files/RevocationTransparency.pdf (visited on 09/20/2021).

[99] Leiser, M. Astroturfing, cyberturfing and other online persuasion campaigns. *EJLT* 7, 1 (2016). URL: http://ejlt.org/article/view/501.

[100] Li, Y. and Ye, J. Learning adversarial networks for semi-supervised text classification via policy gradient. In: *Proc. of the 24th ACM SIGKDD*. KDD '18. ACM, London, United Kingdom, 2018, 1715–1723.

[101] *Libsodium documentation*. URL: https://libsodium.gitbook.io/doc/.

[102] Long, K. *Keeping The Times Civil, 16 Million Comments and Counting*. 2017. URL: https://www.nytimes.com/2017/07/01/insider/times-comments.html.

[103] Louw, M. T., Ganesh, K. T., and Venkatakrishnan, V. AdJail: practical enforcement of confidentiality and integrity policies on web advertisements. In: *19th USENIX Security Symposium (USENIX Security 10)*. USENIX Association, Washington, DC, Aug. 2010. URL: https://www.usenix.org/

conference/usenixsecurity10/adjail-practical-enforcement-confidentiality-and-integrity-policies-web.

[104] Macedo, N., Brunel, J., Chemouil, D., Cunha, A., and Kuperberg, D. Lightweight specification and analysis of dynamic systems with rich configurations. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE'16: 24nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering. ACM, Seattle WA USA, Nov. 2016, 373–383. URL: https://dl.acm.org/doi/10.1145/2950290.2950318 (visited on 07/27/2023).

[105] Marcos Cáceres, Kenneth Rohde Christiansen, Mounir Lamouri, Anssi Kostiainen, Matt Giuca, and Aaron Gustafson. *Web App Manifest*. URL: https://www.w3.org/TR/appmanifest/.

[106] Masinter, L. *RFC2397: The "Data" URL Scheme*. USA, 1998. URL: https://www.rfc-editor.org/rfc/rfc2397.

[107] *Matrix's Hydrogen client*. https://hydrogen.element.io/. Accessed on: [2023-08-23].

[108] *Measure performance with the RAIL model*. June 10, 2020. URL: https://web.dev/rail/ (visited on 10/24/2022).

[109] *Media Capture and Streams: 9.2 Media Devices*. World Wide Web Consortium (W3C). 2023. URL: https://w3c.github.io/mediacapture-main/#mediadevices.

[110] Medien, H. 2018. URL: https://www.heise.de/newsticker/meldung/Der-Hass-in-den-Kommentarspalten-macht-vielen-Medien-Sorgen-4140143.html.

[111] *MEGA*. https://mega.io/. Accessed on: [2023-08-23].

[112] Meier, S., Schmidt, B., Cremers, C., and Basin, D. The tamarin prover for the symbolic analysis of security protocols. In: *Computer Aided Verification*. Ed. by Sharygina, N. and Veith, H. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, 696–701.

[113] Meiklejohn, S., DeBlasio, J., O'Brien, D., Thompson, C., Yeo, K., and Stark, E. SoK: SCT Auditing in Certificate Transparency. *PoPETs* 2022, 3 (July 2022), 336–353. URL: https://petsymposium.org/popets/2022/popets-2022-0075.php (visited on 11/21/2022).

[114] Melara, M. S., Blankstein, A., Bonneau, J., Felten, E. W., and Freedman, M. J. CONIKS: bringing key transparency to end users. In: *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., Aug. 2015, 383–398. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara.

[115] Michael Bristow BBC News, B. *China's internet 'spin doctors'*. 2008. URL: http://news.bbc.co.uk/2/hi/7783640.stm.

[116] Mihaylov, T., Georgiev, G., and Nakov, P. Finding opinion manipulation trolls in news community forums. In: *Proc. of the Nineteenth CoNLL*. 2015, 310–314.

[117] Mike, W. *Post-Spectre Web Development*. W3C Working Draft. 2021. URL: https://www.w3.org/TR/post-spectre-webdev/.

[118] *MIRACL with DAA API for Javascript*. URL: https://github.com/iesiyok/MIRACL.

[119] *Modifications of AdSense ad code*. 2021. URL: https://support.google.com/adsense/answer/1354736 (visited on 11/05/2021).

[120] Moonesamy, S. *RFC6694: The "about" URI Scheme*. USA, 2012. URL: https://www.rfc-editor.org/rfc/rfc6694.

[121] *Multiprecision Integer and Rational Arithmetic Cryptographic Library (MIRACL)*. URL: https://github.com/miracl/MIRACL.

[122] Nielsen, J. *Response Times: The 3 Important Limits*. Jan. 1, 1993. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (visited on 10/24/2022).

[123] *Nimiq*. https://www.nimiq.com. Accessed on: [2023-08-23].

[124] Ortega, F. J., Troyano, J. A., Cruz, F. L., Vallejo, C. G., and Enríquez, F. Propagation of trust and distrust for the detection of trolls in a social network. *Computer Networks* 56, 12 (2012), 2884–2895. URL: http://www.sciencedirect.com/science/article/pii/S138912861200179X.

[125] *Page.metrics method*. URL: https://pptr.dev/api/puppeteer.page.metrics/ (visited on 12/03/2022).

[126] Park, D., Stefănescu, A., and Roşu, G. KJS: a complete formal semantics of JavaScript. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15: ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, Portland OR USA, June 3, 2015, 346–356. URL: https://dl.acm.org/doi/10.1145/2737924.2737991 (visited on 07/20/2021).

[127] Peng, J., Choo, R. K., and Ashman, H. Astroturfing detection in social media: using binary n-gram analysis for authorship attribution. In: *2016 IEEE Trustcom/BigDataSE/ISPA*. Aug. 2016, 121–128.

[128] Pettersen, Y. N. *The Transport Layer Security (TLS) Multiple Certificate Status Request Extension*. RFC 6961. June 2013. URL: https://rfc-editor.org/rfc/rfc6961.txt.

[129] Pickren, R. *Webcam Hacking*. 2019. URL: https://www.ryanpickren.com/webcam-hacking.

[130] Pinkas, B., Rosulek, M., Trieu, N., and Yanai, A. SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension. In: *Advances in Cryptology – CRYPTO 2019*. Ed. by Boldyreva, A. and Micciancio, D. Lecture Notes in Computer Science. Springer International Publishing, Cham, 2019, 401–431.

[131] Pinkas, B., Schneider, T., and Zohner, M. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.* 21, 2 (Jan. 2018), 7:1–7:35. URL: http://doi.acm.org/10.1145/3154794.

[132] Reis, C., Dunagan, J., Wang, H. J., Dubrovsky, O., and Esmeir, S. Browsershield: vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web* 1, 3 (2007), 11. URL: https://doi.org/10.1145/1281480.1281481.

[133] Reuter, M. and Dachwitz, I. *Moderation bleibt Handarbeit: Wie große Online-Medien Leserkommentare moderieren*. URL: https://netzpolitik.org/

2016/moderation-bleibt-handarbeit-wie-tageszeitungen-leserkommentare-moderieren.

[134] Laurie, B., Langley, A., and Kasper, E. *Certificate Transparency*. RFC 6962 (Experimental). Fremont, CA, USA: RFC Editor, June 2013. URL: https://www.rfc-editor.org/rfc/rfc6962.txt.

[135] Fielding, R. and Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. URL: https://www.rfc-editor.org/rfc/rfc7231.txt.

[136] Ritzdorf, H., Wust, K., Gervais, A., Felley, G., and Capkun, S. TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing. In: *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. Internet Society, San Diego, CA, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-4_Ritzdorf_paper.pdf (visited on 06/17/2021).

[137] Ross, D. and Gondrom, T. *HTTP Header Field X-Frame-Options*. RFC 7034. Oct. 2013. URL: https://www.rfc-editor.org/info/rfc7034.

[138] Russell, A., Song, J., Archibald, J., and Kruisselbrink, M. *Service Workers 1*. URL: https://www.w3.org/TR/service-workers/ (visited on 07/23/2021).

[139] Santos, J. F., Maksimović, P., Grohens, T., Dolby, J., and Gardner, P. Symbolic Execution for JavaScript. In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. PPDP '18: The 20th International Symposium on Principles and Practice of Declarative Programming. ACM, Frankfurt am Main Germany, Sept. 3, 2018, 1–14. URL: https://dl.acm.org/doi/10.1145/3236950.3236956 (visited on 07/20/2021).

[140] Sasse, R., King, S. T., Meseguer, J., and Tang, S. Ibos: a correct-by-construction modular browser. In: *Formal Aspects of Component Software*. Ed. by Păsăreanu, C. S. and Salaün, G. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, 224–241.

[141] Schmidt, A. L., Zollo, F., Scala, A., Betsch, C., and Quattrociocchi, W. Polarization of the Vaccination Debate on Facebook. *arXiv e-prints*, arXiv:1801.02903 (Jan. 2018), arXiv:1801.02903. arXiv: 1801.02903 [cs.SI].

[142] Schmidt, B., Meier, S., Cremers, C., and Basin, D. The tamarin prover for the symbolic analysis of security protocols. In: *CAV'13*. Vol. 8044. LNCS. Springer, 2013, 696–701.

[143] Schwarz, M., Lipp, M., and Gruss, D. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In: *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. Internet Society, San Diego, CA, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07A-3_Schwarz_paper.pdf (visited on 05/19/2021).

[144] Seah, C. W., Chieu, H. L., Chai, K. M. A., Teow, L., and Yeong, L. W. Troll detection by domain-adapting sentiment analysis. In: *2015 18th Int. Conf. on Inf. Fusion (Fusion)*. July 2015, 792–799.

[145] *Selenium*. URL: https://www.selenium.dev/.

[146] *SharedArrayBuffer Objects*. ECMAScript Language Specification. 2023. URL: https://tc39.es/ecma262/multipage/structured-data.html#sec-sharedarraybuffer-objects.

[147] Shearer, E. and Matsa, K. E. *News User Acrosss Social Media Platforms 2018*. Pew Research Center. 2018. URL: https://www.pewresearch.org/journalism/2018/09/10/news-use-across-social-media-platforms-2018/.

[148] *Signed exchange generation*. 2021. URL: https://github.com/WICG/webpackage/tree/master/go/signedexchange (visited on 11/05/2021).

[149] Sluganovic, I., Roeschlin, M., Rasmussen, K. B., and Martinovic, I. Using reflexive eye movements for fast challenge-response authentication. In: *Proc. of the 2016 ACM SIGSAC CCS*. ACM. 2016, 1056–1067.

[150] *Source code for Twitter's Recommendation Algorithm*. URL: https://github.com/twitter/the-algorithm (visited on 06/11/2023).

[151] *SpiderOak*. https://spideroak.com/. Accessed on: [2023-08-23].

[152] *Square*. https://squareup.com. Accessed on: [2023-08-23].

[153] Steffens, M., Musch, M., Johns, M., and Stock, B. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In: *Proceedings 2021 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. Internet Society, Virtual, 2021. URL: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_3A-3_24028_paper.pdf (visited on 09/20/2021).

[154] *Stripe*. https://stripe.com. Accessed on: [2023-08-23].

[155] Team, C. D. *The Coq Proof Assistant Reference Manual V8.4pl2*. 2013. URL: https://flint.cs.yale.edu/cs430/coq/pdf/Reference-Manual.pdf.

[156] *Tendermint avl performance and benchmarks*. URL: https://github.com/tendermint/iavl/blob/master/PERFORMANCE.md.

[157] *The Web Platform Tests project*. https://web-platform-tests.org/. Accessed on: [2023-08-23].

[158] Thies, J., Zollhofer, M., Stamminger, M., Theobalt, C., and Nießner, M. Face2face: real-time face capture and reenactment of rgb videos. In: *Proc. of the IEEE CVPR*. 2016, 2387–2395.

[159] Tony Romm, W. P. *Senate investigators want answers from Reddit and Tumblr on Russia meddling*. 2018.

[160] Torlak, E. and Jackson, D. Kodkod: a relational model finder. In: *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Mar. 2007, 632–647.

[161] *Trillian*. 2021. URL: https://github.com/google/trillian (visited on 09/20/2021).

[162] *Trillian – Experimental Beam Map Generation*. 2021. URL: https://github.com/google/trillian/tree/b7ea8d2ca870e5b8ae1c05e9d2a33c4fdcca4580/experimental/batchmap (visited on 11/05/2021).

[163] *Trollthrottle Big Query*. URL: https://github.com/iesiyok/trollthrottle/tree/master/big_query.

[164] *Trollthrottle Browser Extension*. URL: https://github.com/iesiyok/trollthrottle_chrome.

[165] *Trollthrottle Simulation*. URL: https://github.com/iesiyok/trollthrottle.

[166] Twitter, I. *Update on Twitter's review of the 2016 US election*. 2018. URL: https://blog.twitter.com/official/en_us/topics/company/2018/2016-election-update.html.

[167] *Using Subresource Integrity*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity#using_subresource_integrity (visited on 11/03/2021).

[168] Uzun, E., Chung, S. P. H., Essa, I., and Lee, W. Rtcaptcha: a real-time captcha based liveness detection system. In: *NDSS*. Georgia Institute of Technology, 2018.

[169] VanToll, T. *What Exactly Is..... The 300ms Click Delay*. Nov. 21, 2013. URL: https://www.telerik.com/blogs/what-exactly-is.....-the-300ms-click-delay (visited on 10/24/2022).

[170] Veronese, L., Farinier, B., Bernardo, P., Tempesta, M., Squarcina, M., and Maffei, M. Webspec: towards machine-checked analysis of browser security mechanisms. In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, May 2023, 2761–2779. URL: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179465.

[171] Wang, G., Mohanlal, M., Wilson, C., Wang, X., Metzger, M., Zheng, H., and Zhao, B. Y. Social turing tests: crowdsourcing sybil detection. *arXiv preprint arXiv:1205.3856* (2012).

[172] West, M. *Mikewest/Signature-Based-Sri*. July 13, 2020. URL: https://github.com/mikewest/signature-based-sri (visited on 07/20/2021).

[173] West, M. and Sartori, A. *Content Security Policy Level 3*. W3C Working Draft. 2023. URL: https://www.w3.org/TR/CSP3/.

[174] *What is a web server?* https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server. Accessed on: [2023-08-23].

[175] *WhatsApp Web*. https://web.whatsapp.com/. Accessed on: [2023-08-23].

[176] WHATWG. *Cross-Origin Opener Policies*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/browsers.html#cross-origin-opener-policies.

[177] WHATWG. *DOM API*. Accessed: 2023-09-01. 2023. URL: https://dom.spec.whatwg.org/.

[178] WHATWG. *DOM Standard*. Accessed on: [2023-09-01]. 2023. URL: https://dom.spec.whatwg.org/#ref-for-dom-document-createelement%E2%91%A0.

[179] WHATWG. *Fetch: CORS protocol*. Accessed on: [2023-09-01]. 2023. URL: https://fetch.spec.whatwg.org/#http-cors-protocol.

[180] WHATWG. *Fetch: Cross-Origin Embedder Policies*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/browsers.html#coep.

[181] WHATWG. *Fetch: Cross-Origin Resource Policy Header*. Accessed on: [2023-09-01]. 2023. URL: https://fetch.spec.whatwg.org/#cross-origin-resource-policy-header.

[182] WHATWG. *HTML Standard: 4.8.5 Iframe Element*. Accessed: 2023-09-01. 2023. URL: https://html.spec.whatwg.org/multipage/iframe-embed-object.html#the-iframe-element.

[183] WHATWG. *HTML Standard: A cross-origin isolated capability*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/webappapis.html#concept-settings-object-cross-origin-isolated-capability.

[184] WHATWG. *HTML Standard: Document.write*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/dynamic-markup-insertion.html#dom-document-write-dev.

[185] WHATWG. *HTML Standard: The History Interface*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/nav-history-apis.html#the-history-interface.

[186] WHATWG. *HTML Standard: The History Interface*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/nav-history-apis.html#dom-history-pushstate-dev.

[187] WHATWG. *HTML Standard: The Location Interface*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/nav-history-apis.html#dom-location-replace-dev.

[188] WHATWG. *HTML Standard: The Window Object*. Accessed on: [2023-09-01]. 2023. URL: https://html.spec.whatwg.org/multipage/nav-history-apis.html#the-window-object.

[189] WHATWG. *Http Responses: Access-Control-Allow-Origin*. Accessed on: [2023-09-01]. 2023. URL: https://fetch.spec.whatwg.org/#http-responses.

[190] World Wide Web Consortium (W3C). *HTML Standard*. Online. Accessed on: [2023-07-23]. URL: https://www.w3.org/html/wg/spec/browsers.html#browsing-contexts.

[191] World Wide Web Consortium (W3C). *Interface Document*. Online. Accessed on: [2023-07-23]. URL: https://dom.spec.whatwg.org/#interface-document.

[192] World Wide Web Consortium (W3C). *Origin*. Online. Accessed on: [2023-07-23]. URL: https://www.w3.org/html/wg/spec/origin-0.html.

[193] World Wide Web Consortium (W3C). *Secure Contexts*. https://w3c.github.io/webappsec-secure-contexts/. Accessed on: [2023-07-23].

[194] World Wide Web Consortium (W3C). *URLs*. Online. Accessed on: [2023-07-23]. URL: https://www.w3.org/TR/2011/WD-html5-20110525/urls.html#absolute-url.

[195] Wullner, D. *Lassen Sie uns diskutieren*. 2015. URL: https://www.sueddeutsche.de/kolumne/ihre-sz-lassen-sie-uns-diskutieren-1.2095271.

[196] Yasskin, J. *Signed HTTP Exchanges*. 2021. URL: https://wicg.github.io/webpackage/draft-yasskin-http-origin-signed-responses.html (visited on 04/13/2021).

[197] Yasskin, J. *Use Cases and Requirements for Web Packages*. URL: https://datatracker.ietf.org/doc/draft-yasskin-wpack-use-cases/ (visited on 11/02/2020).

[198] Yasskin, J. *Web Bundles*. URL: https://wicg.github.io/webpackage/draft-yasskin-dispatch-web-packaging.html.

[199] *Z3 Theorem Prover*. URL: https://github.com/Z3Prover/z3/wiki#background.

[200] Zeit, D. 2016. URL: https://www.zeit.de/zeit-magazin/2016/31/kommentare-internet-medien-community-redakteur.

[201] Zeit, D. 2018. URL: https://blog.zeit.de/glashaus/2018/03/02/wie-wir-leserkommentare-moderieren/.