A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification

Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, Nils Wenzler, and Tim Würtele
University of Stuttgart, Stuttgart, Germany
{quoc-huy.do, pedram.hosseyni, ralf.kuesters, guido.schmitz, tim.wuertele}@sec.uni-stuttgart.de, nils.wenzler@gmail.com

***Abstract*—Payment is an essential part of e-commerce. Merchants usually rely on third-parties, so-called payment processors, who take care of transferring the payment from the customer to the merchant. How a payment processor interacts with the customer and the merchant varies a lot. Each payment processor typically invents its own protocol that has to be integrated into the merchant's application and provides the user with a new, potentially unknown and confusing user experience.**

Pushed by major companies, including Apple, Google, Mastercard, and Visa, the W3C is currently developing a new set of standards to unify the online checkout process and "streamline the user's payment experience". The main idea is to integrate payment as a native functionality into web browsers, referred to as the *Web Payment APIs*. While this new checkout process will indeed be simple and convenient from an end-user perspective, the technical realization requires rather significant changes to browsers.

Many major browsers, such as Chrome, Firefox, Edge, Safari, and Opera, already implement these new standards, and many payment processors, such as Google Pay, Apple Pay, or Stripe, support the use of Web Payment APIs for payments. The ecosystem is constantly growing, meaning that the Web Payment APIs will likely be used by millions of people worldwide.

So far, there has been no in-depth security analysis of these new standards. In this paper, we present the first such analysis of the Web Payment APIs standards, a rigorous formal analysis. It is based on the *Web Infrastructure Model (WIM)*, the most comprehensive model of the web infrastructure to date, which, among others, we extend to integrate the new payment functionality into the generic browser model.

Our analysis reveals two new critical vulnerabilities that allow a malicious merchant to over-charge an unsuspecting customer. We have verified our attacks using the Chrome implementation and reported these problems to the W3C as well as the Chrome developers, who have acknowledged these problems. Moreover, we propose fixes to the standard, which by now have been adopted by the W3C and Chrome, and prove that the fixed Web Payment APIs indeed satisfy strong security properties.

I. Introduction

Today, it is impossible to imagine everyday life without e-commerce. Consumers order goods or other services online and make the necessary payments directly online as well. Many different variants have emerged on the web to carry out an order and the associated payment. Basically, every merchant or store application performs this process slightly differently. Merchants, instead of processing a payment by themselves (e.g., by collecting credit card information and charging the cardholder), often outsource the actual payment process to a third party, a payment processor, such as Stripe, Google Pay, or PayPal.

In such a heterogeneous environment every participant is at risk of making mistakes: The information flow between merchant and payment processor could be manipulated by a malicious customer [57], payment processor schemes could be flawed [61], or users could be confused by the many different user interfaces and be more susceptible to phishing attacks [18], just to name a few. Also, customers are likely to abort a checkout process when facing a bad user experience [27].

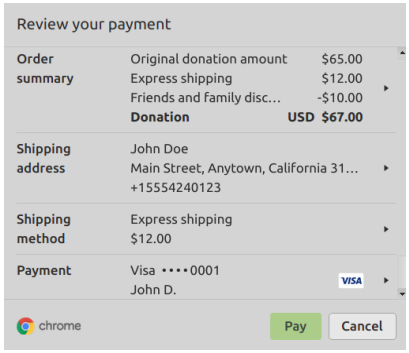

Figure 1 Web Payment User Interface in Google Chrome

Pushed by a long list of major companies in the technology and banking sector (e.g., Amazon, American Express, Apple, Barclays Bank, Facebook, Google, Huawei, Klarna, Mastercard, Netflix, Samsung, Stripe, and Visa), the W3C is currently developing the *Web Payment APIs (WPA)* [50], an approach to simplify and standardize payment and the checkout process in the browser. To this end, the W3C extends browsers with a new, native functionality that provides a way to negotiate all necessary checkout information among the customer, the merchant's website, and an (external) payment processor (including the selection of the payment processor). The main idea is that the merchant, instead of providing web pages or JavaScript by itself for checkout, hands-off this process to the WPA in the user's web browser. The browser then presents the user with the new payment user interface (see Figure 1) that is outside of the web context. In this user interface, the user can review the order, select a (previously stored) shipping address, a payment method, and a payment processor (e.g., pay by credit card using Stripe). As this dialog is always the same, regardless of what merchant or payment processor is involved, the user experience will also be always the same. As a result, the checkout process is unified, fast, and more convenient.

The feature set introduced by the WPA forms a quite complex protocol and requires several modifications to browsers. For example, payment processors install a so-called payment handler in the user's browser that is based on the recently introduced service worker infrastructure [47]. Also, several new means for intra-browser communication are introduced to facilitate the exchange of user and payment information

across the entities involved in the checkout process. The WPA define several interfaces, each of which is crafted for the communication with one of the entities involved in the process.

Major browsers already support the current state of the specifications and well-known payment providers like Google Pay [29], Apple Pay [2], and Stripe [49] are also already supporting the standard. Hence, it is likely to see widespread adoption by merchants in the near future, and hence, it is important to perform rigorous analysis now before problems are harder to fix. As the WPA are used to handle sensitive personal and payment data and even to initiate payments, users, merchants and payment providers have to rely on their security. At the same time, the handling of personal and payment data makes the protocols and APIs a lucrative attack target, making the WPA an interesting subject for rigorous security analysis.

In this paper, we present the first in-depth formal security analysis of the WPA. We base our analysis on the most comprehensive formal model of the web infrastructure to date, the *Web Infrastructure Model (WIM)* [21]. The WPA inherently uses many features of the web, including scripts, the notion of origins (and hence, the notion of schemes and domains), the window and document structure of browsers, cross-document messages, HTTPS, and XMLHttpRequests. Therefore, basing the analysis of the WPA on a model that supports these features is crucial in order to be able to model the WPA in a natural, detailed, and faithful way.

As the WPA extend web browsers significantly, we have to extend the WIM itself to be able to formally analyze the WPA. These extensions make the WIM more expressive and add a new attack surface, which can be relevant for future analyses of web applications and standards even those unrelated to payment as attackers can make use of the new APIs in unexpected ways, making this extension of independent interest.

Our formal analysis of the WPA based on the extended WIM reveals two new critical vulnerabilities which allow malicious merchants to over-charge unsuspecting customers. We verified these security problems with the Google Chrome browser and reported them to the Chrome developers as well as the W3C, who both acknowledged the flaws. We also propose fixes, which by now have been incorporated in the standard as well as the Chrome implementation. The Chrome developers even released a hot-fix for their current stable version.

To show that the fixes we propose are indeed sufficient, we use our formal model of the WPA to prove that the standard indeed satisfies strong security properties with our fixes in place. We note that in order for this analysis to be meaningful it is important that this analysis is based on a detailed model of the web infrastructure, as provided in this work with the extended WIM: first, as mentioned, to obtain a faithful model of the WPA in the first place, and second, to cover a large attack surface.

Related Work. As mentioned above, the WPA are the first proposal to standardize web payment [46]. Until now, there has been no in-depth security analysis of the WPA, and to the best of our knowledge, there has been no in-depth formal analysis of any web payment protocol, the main reason probably being the previous lack of a standard for web payment.

Besides formal treatment of crypto currencies and the underlying blockchain protocols (see, e.g., [4, 28, 43]) or e-cash (see, e.g., [19, 38]) there is surprisingly little literature on formal analyses of real-world payment systems in general, although security is paramount for such systems. Recently, Basin et al. have formally analyzed the EMV standard (the protocol used by bank cards to perform in-person payment at point-of-sale terminals) using Tamarin [6]. Although the EMV protocol had already been extensively studied before (see, e.g., [10, 41, 44, 45]), Basin et al. discovered two new severe attacks. Using formal methods, they also formally proved that the protocol is secure in certain configurations. This example also illustrates the benefit of applying formal methods.

Formal methods are also applied for analyzing APIs outside of the context of payments, e.g., for analyzing the security of an API for social networks [3], the API defined in PKCS#11 [17], the W3C Web Authentication API [31], and the W3C Web Cryptography API [58]. However, none of these analyses were conducted within a model of the web infrastructure or using a detailed browser model, and thus, do not account for attacks that arise from the web infrastructure.

As mentioned above, we base our formal analysis on the WIM. The WIM has successfully been used to analyze web standards, so far standards for authorization and authentication, such as OAuth 2.0 [24], OpenID Connect 1.0 [25], Mozilla's BrowserID [21, 22], and the OpenID Financial-Grade API[1] [20]. These analyses all uncovered several severe, previously unknown attacks and illustrate the power of the WIM. The WIM has not been used to analyze or even specify security properties of payment systems yet. Also, in contrast to previous analyses, in this paper we focus our analysis on a native web feature itself rather than protocols that run on top of the web infrastructure.

The WIM is in fact, by far, the most comprehensive model of the web infrastructure to date. Other models of the web infrastructure, such as work by Pai et al. [42], by Kumar [35, 36], or Bansal et al. [5] are far more abstract and limited by the tools upon which they are based. Other approaches, such as [9] and [11], model only single components of the web and do not take the overall infrastructure into account.

Contributions. In summary, our main contributions presented in this paper are as follows:

- We conduct the first in-depth security analysis of the WPA, and in particular, the first formal analysis. As mentioned, the WPA is likely to be used for payments on the web by millions of people.
- We significantly extend the WIM to support the native web features of the WPA. This extension also includes DOM event handling and even a framework for so-called service workers [47], a standard independent of WPA. These extensions are useful also beyond the analysis of the WPA as they constitute an immanent extension of the web attack surface.

[1] A standard which extends OAuth 2.0 and is targeted at authorization of access to protected resources in high-risk environments, but not directly towards payment.

- During our analysis, we uncovered two previously unknown critical flaws which would allow a malicious merchant to over-charge unsuspecting customers.
- We construct practical attacks from these findings and verify them using the Google Chrome implementation. We notified the respective working group at the W3C as well as the Google Chrome developers, who acknowledged the problems.
- We propose fixes that close the discovered vulnerabilities. The fixes have been incorporated into the WPA standard as well as Google Chrome.
- Using the developed extended WIM and the detailed formal model of the WPA based on the extended WIM, we formally prove that our fixes are indeed sufficient and that the fixed WPA standard satisfies strong security properties.

Structure of This Paper. In Section II, we present the WPA. Section III describes the vulnerabilities discovered in our formal security analysis, the fixes we propose, and the reactions to our findings from the W3C Web Payments Working Group and the Google Chrome team. In Section IV, we briefly recall the WIM, with our extensions as well as the model of WPA presented in Section V. We formulate the security of the WPA standard in Section V, with a proof sketch given in Section VII. We conclude in Section VIII. The full detailed model and proofs are given in the appendix.

II. The W3C Web Payment APIs

In the following section, we give an overview of the WPA, describe the different entities that are involved in the checkout process, followed by an overview of the specifications of the WPA. We then explain a typical checkout flow and some additional sub-flows in detail.

A. Overview of the WPA

The WPA are a set of standards [12, 13, 14, 32, 37] published by W3C's Web Payments Working Group. The standards define different parts of the checkout process, centered around the browser. On a high level, the checkout process is as follows: The customer (which is also referred to as *payer*) decides to pay, e.g., by clicking on a "checkout" button. The merchant (*payee*) website then creates what is called a *Payment Request*, a JavaScript object containing checkout data like a list of items, accepted *Payment Methods* – e.g., credit card – and a total amount. The Payment Request is then handed to the Payment Request API [14], implemented by the browser. The payment request triggers the browser to show a special user interface, called the *payment UI* (which is provided by the browser itself, not the website, see Figure 1). This payment UI allows the user to select one of the shipping and payment data sets stored in the browser, e.g., credit card information, or enter new data sets. After selecting or entering the required data sets, the user confirms the checkout in the payment UI. Depending on the selected payment method, the browser might ask the customer to perform additional steps to authorize the payment before executing it, e.g., log in to the *payment provider*. Payment providers (which the specifications call *Payment Method Providers*) take care of the actual financial transaction. The interaction between customer, merchant, and payment provider is orchestrated from within the browser by a *payment handler* (see Section II-B).

Compared to common practice where each merchant presents the user a web form, which might be different at each merchant, this approach has a number of benefits as pointed out by, e.g., Mozilla, Microsoft, Google, and several blog posts on the topic [30, 33, 34, 39, 40, 55]: 1) The user can easily update her payment and shipping information for all websites as it is stored in one central place in the browser and the data there can be "re-used" over several websites. 2) There is no need to store any payment or even shipping data at the merchant anymore, lowering the impact and attackers' incentive for data breaches at the merchant's end (e.g., [26, 48, 56, 59, 60] to name a few). Note that this does not necessarily make the browser more interesting to attack: all this data is entered through the browser anyway and modern browsers' autofill features for web forms already require them to store address and payment data. 3) As the payment UI is a part of the browser, the user experience and checkout flow are always the same, avoiding confusion and errors. 4) Adopting this new, easier, and faster checkout procedure is also expected to increase conversion rates for the merchants, especially with mobile users.

B. Components of the WPA

As mentioned, the WPA consist of a set of specifications, different W3C standards, which we sketch here before describing the protocol flows defined by them.

Payment Request API [14]. The Payment Request API can be used by the merchant to interact with the customer and the payment method provider through the customer's browser. It is implemented by the browser and handles communication between the merchant and the WPA in the browser.

Payment Handler API [32]. The W3C WPA use so-called *web service workers* [47], another W3C standard which exists independently of the WPA. In a nutshell, web service workers are event-driven JavaScript programs that can be installed in a browser by some website. These workers are running outside the context of a specific window or tab, but still at the origin that installed them. They are intended to extend web applications with some offline functionality, but in the WPA this is not the main intent: the Payment Handler API specification defines how special web service workers called *payment handlers* can interact with the browser to handle payments on behalf of a customer. Similarly to the Payment Request API, the Payment Handler API is also implemented by the customer's browser.

Payment Method Identifiers [12]. In general, a payment method is identified by either a standardized string, like `basic-card` (see below), or a URL (pointing to a payment method manifest, see below). This document specifies details of these identifiers, e.g., that a URL's scheme must be `https` to be a valid payment method identifier.

Payment Method Basic Card [13]. This specification defines the `basic-card` payment method, a mechanism to provide payment card details, such as a credit card number, directly to the merchant. The merchant would then process

this information by itself without using any other features of the WPA, such as payment handlers.

Payment Method Manifest [37]. This specification describes how a payment method provider, like Google Pay, can specify a default payment handler and a set of permissible payment handler origins for "its" payment method in a machine-readable format. This manifest has to be accessible under the URL identifying the payment method, so browsers can retrieve the manifest.

C. Protocol Flow

In the following, we illustrate the core workings of the WPA by describing the steps of a simple, successful protocol execution, shown in Figure 2.[2] Note that the specifications define additional (sub-)flows, some of which we discuss in the following subsection. Our formal model captures these additional flows as well as several ways to abort a flow.

We assume that there is at least one payment handler already installed in the customer's browser – typically by visiting some payment method provider's website and agreeing to the installation of that provider's payment handler.

The flow starts with the creation of a payment request by the merchant in Step 1, typically after the customer expressed the wish to checkout, e.g., by clicking a "Checkout" button (the payment request is created via JavaScript code contained in the merchant's website). This payment request contains the following fields:

- `id`: A unique identifier for the payment (chosen by the merchant). If omitted by the merchant, the browser generates the payment id. This id is included in all messages and events to uniquely identify this specific payment process.
- `methodData`: A list of payment methods accepted by the merchant, denoted by their payment method identifiers (see Section II-B). Each element can also have an associated payment method data set with additional information for the respective payment method (more on that later).
- `details`: Total cost, and optional details, e.g., a list of items with their respective costs, and shipping options.
- `options`: Settings indicating which data the customer has to provide. For example, whether an email address, shipping address or phone number are required.

After creating the payment request, the merchant's website hands this data to the browser by means of the Payment Request API 2. Upon receiving a payment request `PR`, the browser compares the list of payment methods mentioned in `PR.methodData` (i.e., those supported by the merchant) with its list of registered payment handlers and selects all payment handlers registered for at least one of the payment methods from `PR.methodData` 3.[3] After this initial selection, the browser triggers each of the selected payment handlers with a `CanMakePaymentEvent` 4. This event contains some basic information about the requested payment: The origin of the merchant's top level page, the origin at which the payment request was initiated and the applicable payment method data sets (from `PR.methodData`).

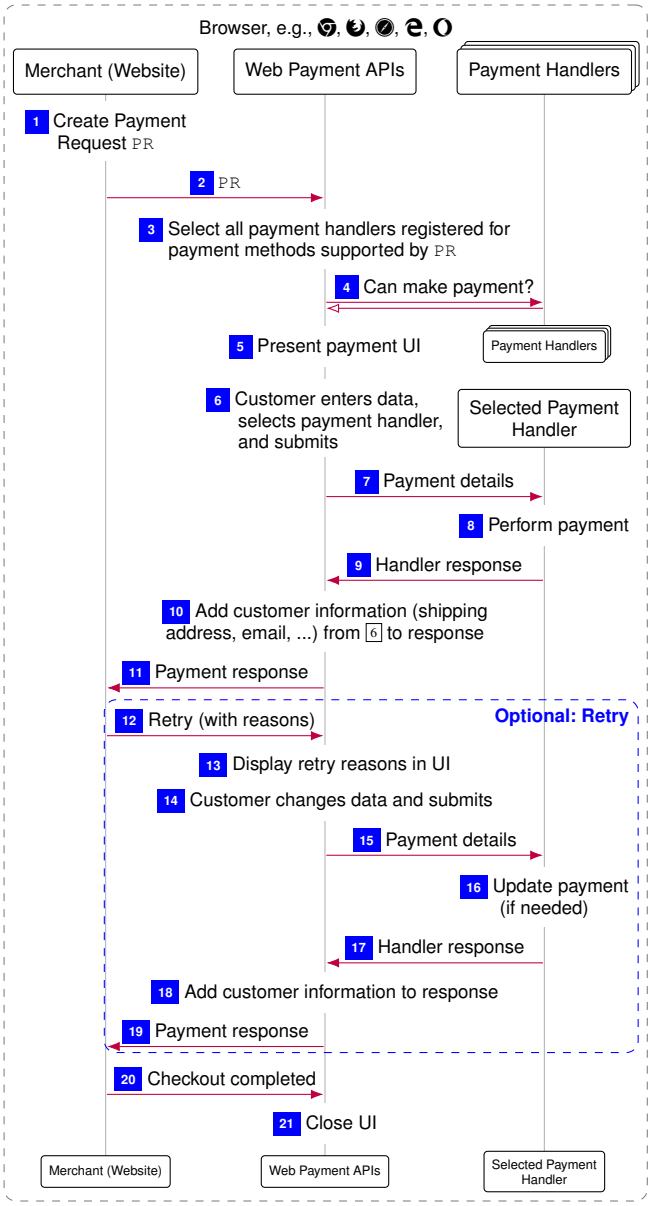

Figure 2 General flow of a payment with the WPA. Note that payment handlers are not part of the WPA implemented by the browser. They are web service workers – typically provided by the payment provider – which use the WPA to engage in the payment process. They may also send requests, e.g., to the payment provider and – with some restrictions – open windows.

Upon receiving a `CanMakePaymentEvent`, the payment handler determines if it can process the payment request. This decision process is specific to the payment method and may depend, for example, on legal requirements. The inner workings of this decision are outside of the scope of the WPA. Note that a payment handler may communicate with the payment provider to make this decision.

After receiving responses from all triggered payment handlers, the browser shows a special dialog (that is *not* part of

[2]All flow figures in this paper are generated with Annex: https://github.com/danielfett/annexlang

[3]If no matching payment handler is installed in the browser, the checkout is aborted. In particular, the browser does not try to install missing payment handlers at this stage.

a website, but of the browser itself) called the *payment UI* [5] (see also Figure 1). The payment UI allows the customer to enter (or select stored) requested information like shipping address, email address, and of course the payment details [6]. The latter includes the selection of one of the available payment handlers and methods.[4] If the customer enters (or selects a stored or changes) the shipping address, an additional step is required: As this might change the shipping costs, the merchant website is informed and receives a partially anonymized address to recalculate shipping costs. The details of this procedure are omitted from Figure 2 for brevity of presentation (see Section II-D for details).

Once the customer submits her choices, the browser assembles a `PaymentRequestEvent` and hands it to the payment handler selected by the customer [7]. This `PaymentRequestEvent` contains the same data as the `CanMakePaymentEvent` (see above) plus the payment data entered by the user, e.g., a credit card number, expiration date and verification number.

When receiving the `PaymentRequestEvent`, the payment handler takes the necessary steps to perform or at least facilitate a payment [8]. This can be as simple as returning some information from the `PaymentRequestEvent` (e.g., credit card data) in the handler response, but it can also be a complex process including communication with the payment provider or opening new windows (e.g., opening a Google Pay window in which the customer authenticates herself and authorizes the payment). The exact process is specific to the payment method and thus not within the scope of the WPA specifications.

After performing these steps, the payment handler finishes its work by creating a `PaymentHandlerResponse` [9]. This response contains (once again) the payment method identifier and a `details` field, whose exact contents depend on the payment method. The contents of `details` can, for example, be some signed payment confirmation by the payment provider or be as simple as just "reflecting" credit card data.

Upon receiving the handler response, the browser creates a payment response [10]. This response consists of the `details` from the handler response, the selected payment method, and additional data requested by the merchant in `PR.options`, like a shipping address. The payment response is then handed back to the merchant's website [11].

The merchant can now inspect the response (e.g., verify the validity of the credit card data) and either finish the flow by indicating completion of the checkout process [20] (in this case, the browser closes the payment UI [21]) or by indicating that there is a problem [12]. In the latter case, the merchant can provide a list of reasons for the error. These reasons are freely chosen by the merchant and can be given as a mix of general errors and problems tied to specific parts of the customer data, e.g., the shipping address.

In the error case, the browser prompts the user to retry. To this end, the browser displays the reasons given by the merchant [13] and allows the user to revise the entered data [14].[5] Once the customer has done so and submits again, the browser triggers another `PaymentRequestEvent` to the then selected payment handler with the same payment id, but possibly different payment details (e.g., a different payment method selected by the user). The payment handler then processes and possibly updates the payment (if necessary) and once again answers with a handler response, triggering the browser to create a payment response and hand it to the merchant. At this point, the merchant can again either accept the payment response or initiate another retry.

D. Extended Flows

As already mentioned, the general flow presented above does not include all possible error modes and optional flows. For example, either party can abort the flow at any point, the browser can restrict payment handler execution time, and there are several sub-flows that are or can be initiated during certain steps of the general flow. The most important of these sub-flows are described in the following. Note that our formal model subsumes these flows as well.

1) Update Payment Details: As briefly mentioned in Section II-C, an additional communication step may occur while the customer enters her data into the payment UI: When a shipping address is selected or updated, the merchant is given the opportunity to update certain details of the payment, e.g., the shipping costs. Analogous sub-flows are defined for when the customer selects or changes her selection of: shipping option, payment method, and payer details, i.e., name, email address and phone number. All of these sub-flows are very similar, the only difference is what data the merchant receives. For example, when the customer selects or changes the shipping option, the merchant only gets the ID of the new shipping option, but not the name or phone number.

Due to these similarities, we describe only the sub-flow for a changed shipping option which is depicted in Figure 3.

Obviously, this sub-flow can only be initiated once the customer interacts with the payment UI (cf. Steps [6] and [14] of Figure 2). When the customer selects a shipping option [2], the browser locks the payment UI [3] to prevent the customer from submitting during the update process. In the next step, the browser triggers an event containing a reference to the original payment request and the new shipping option selection [4]. If the merchant has registered an event listener for these events, she can now reply with a new set of payment details, e.g., change the shipping costs (otherwise, the browser will just assume an empty update). The browser will then update the payment request and payment UI accordingly and unlock the payment UI again.

In our formal model (see Section V), we subsume all update flows by allowing the merchant to send updated payment details at any time, but, according to the specifications, the browser accepts updates only until the customer submits. This

[4]A payment handler might support multiple payment methods. The supported payment methods of a payment handler are the so-called *instruments* of that handler; the payment UI shows a list of available instruments.

[5]Not limited to the parts rejected by the merchant, e.g., the customer is also allowed to change the payment handler. This was also modeled in our original formal analysis of the WPA, which led to one of the attacks presented later.

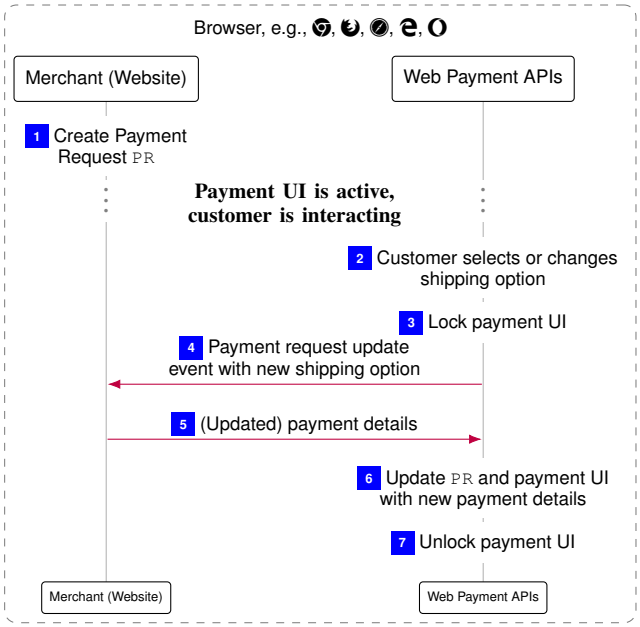

Figure 3 Sub-flow to update payment details after customer data changes (by example of a changed shipping option).

is a safe over-approximation of the merchant's capabilities that only strengthens our security properties.

2) Customer Authentication and Authorization of Payments: When a payment handler handles a payment, it only receives a subset of the payment details from the WPA, e.g., the total amount. These details however do not include information on the customer's identity, i.e., neither the address nor payer details, like name or phone number. This of course raises the question how the payment handler or at least the payment provider gets to know the customer's identity, i.e., how to authenticate the customer. In some jurisdictions, it is also necessary for certain payment providers to collect explicit consent for each transaction.

To meet these requirements, the Payment Handler API specification [32] defines a way for payment handlers to open a window. Such a window could for example contain a login form or ask for additional information like an account number or a one-time authentication code. Following what the specification suggests, browsers usually embed such a window into the existing payment UI. To prevent phishing and similar attacks, a payment handler can only open a window with the same origin that installed the payment handler. In most cases, this window will be a website of the payment provider.

The mechanism to open a window is explicitly modeled in our browser and used by our generic payment handler to authenticate the customer (see Section V-B).

III. Attacks and Vulnerabilities

During our formal analysis of the WPA (see the following sections for details), we found two critical vulnerabilities which we describe in the following, along with suggestions on how to fix them. We also describe our disclosure process and discuss the responses by the W3C Web Payments Working Group and

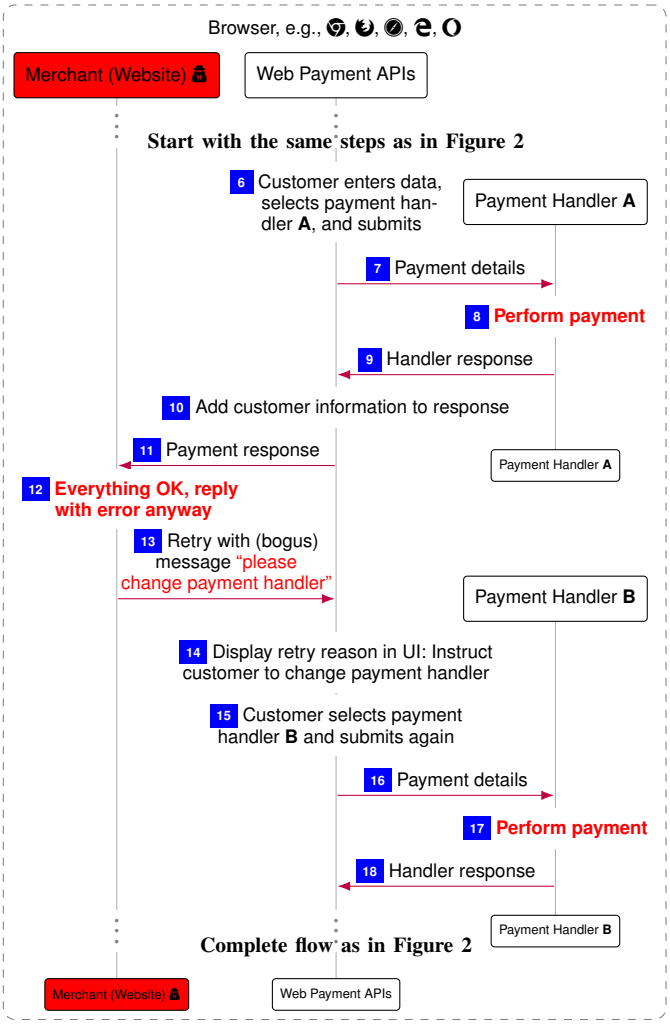

Figure 4 Protocol flow of a retry-based double charging attack: The merchant receives two payments while the customer thinks she only sent a single payment (because the first one allegedly failed).

the Google Chrome developers. Besides being relevant on their own, the attacks we found during our formal analysis are also a good indication of the faithfulness and usefulness of the (extended) WIM.

A. Double Charging with Retry

In Section II-C, we described the retry mechanism built into the WPA. This mechanism is intended to enable merchants to reject a payment response, but allow the customer to change something in order to make the payment response acceptable for the merchant. Also, recall that a *payment id* is used to identify a payment request throughout the different protocol steps. This payment id can be used by payment handlers (and/or payment method providers) to detect duplicates, e.g., when executing a retry. Strictly speaking, the exact workings of such a duplicate detection are outside the scope of the specifications, but it has to rely on the payment id as everything else might change during a retry (including the total costs, e.g., due to a shipping address change).

For our attack, we assume that an honest customer wants to check out at a malicious merchant; the attack flow is depicted in Figure 4: After the customer has initiated the checkout, the merchant creates a (regular) payment request and hands it to the browser, which guides the customer through steps 6 to 11 as described in Section II-C.

Upon receiving the payment response, however, the merchant triggers a retry 12. In this retry request, the merchant includes an error message which instructs the customer to switch to a different payment handler and provider, because the first one allegedly did not work 13. The browser informs the customer by displaying the error message in its payment UI 14. From the customer's perspective, this looks like some legitimate problem with the first payment handler/provider, so she selects a different one and submits again 15. The browser subsequently triggers the (new) payment handler 16, which in turn performs the (second) payment 17. Note that this attack originates from the behavior of the browser and not from the implementation of the payment handlers, as the second payment handler has no way of detecting that it is called in a retry context: the second handler has not seen the payment id of the overall transaction before. Afterward, the flow finishes as usual (though we note that the merchant could of course repeat this attack again).

The result is that the customer paid twice without any indication that she did so. Furthermore, none of the involved payment handlers and payment providers could prevent this, as their views of the attack flow are indistinguishable from a legitimate flow. In particular, the first payment handler and provider are not informed that "their" response was rejected and the second payment handler and provider have no way of knowing that "their" transaction belongs to a retry flow.

Implementation and Verification of the Attack. To test our attack, we implemented a dummy payment handler and merchant. At the time when we did the analysis and implementation, Google Chrome was the only browser available to us that already had full support for the WPA, thus we tested our attack against the Chrome implementation, which turned out to be vulnerable to our attack. Now that the specifications have adopted our fix, the wide range of browsers implementing the WPA specifications today are secured against this attack.

Uncovering the Attack. We discovered the above attack when trying to formally prove the main security theorem (see Section VI). In particular, to prove this theorem, we have to show that a payment transaction is never performed twice.[6] While it is easy to prove this property if the payment is handled by a single payment provider multiple times (who can easily detect and reject duplicated instructions), it is impossible to prove this property if the same transaction can be carried out by two different payment providers (who do not know which payments have been processed by the respective other provider). We therefore have to prove that the same payment request is never sent to two different payment providers. The (unfixed) WPA standard, however, does not restrict that browsers (during a retry) hand the same payment request to different payment handlers (and hence, possibly to multiple payment providers). Thus, the proof fails at this point.[7]

Fix. There are several ways to address this vulnerability, e.g., one could inform a payment handler when "its" transaction is repeated with a different handler, so it can revoke a payment and wait for successful revocation before triggering the second payment handler. Or the payment handler could include a status in its response, indicating whether a payment has already been made and prevent changing the payment handler if so. During our discussion with the W3C Web Payments Working Group, they opted for a very simple fix by disallowing a change of payment handler altogether, even though this might force the customer to abort the whole flow if the payment handler selected in the first place is actually not working.

As this is what the specifications adopted in the end, we updated our formal model to reflect this change, with our analysis results with this fix presented in Sections V to VII.

B. Ambiguous Payment Method Data

As mentioned in Section II-C, each payment request (PR) contains a field called `methodData`, which holds a list of accepted payment methods, along with some additional data for each of these payment methods. This additional data is intended for payment method specific details like a destination account number (to send money to), but can also specify additional fees, e.g., "if you pay with a card belonging to network X, it incurs a US$3.00 processing fee".

Now, for our second attack we again assume that the merchant is malicious, while everybody else is honest. Figure 5 depicts the course of the attack: The merchant creates a payment request `PR` where the list in `methodData` contains (exactly) two entries for method data, both *for the same payment method*, i.e., with identical payment method identifiers; this is perfectly valid according to the W3C specifications. One might not contain fees, the other might contain very high fees. The merchant then hands `PR` to the WPA (as usual). This triggers the browser to search its list of installed payment handlers for matches with the payment method identifiers given in `PR` and query the matching handlers 4. Afterward, the browser shows the payment UI.

We assume that our user will always select the first entry, which in this case contains no additional fees; she enters the required data and submits 6. Her browser now assembles a `PaymentRequestEvent` with payment details, including the *full* list of `methodData`, i.e., both entries, and hands it to the selected payment handler 7. The `PaymentRequestEvent`, however, does not include any information on which exact entry the user has selected from `methodData`. This handler now has to decide which payment method data entry to use. As the specifications do not contain guidance on that either, we assume that the handler always selects the last method data entry (which in our attack incurs a huge additional fee) and performs the payment 8. The

[6] We capture this property for the fixed model in Lemma 18 of Appendix E.

[7] For the fixed WPA model (reflecting the fix in Line 152 of Algorithm 21), we show that a browser never hands the same payment request to two different payment handlers, see Lemma 15 of Appendix E .

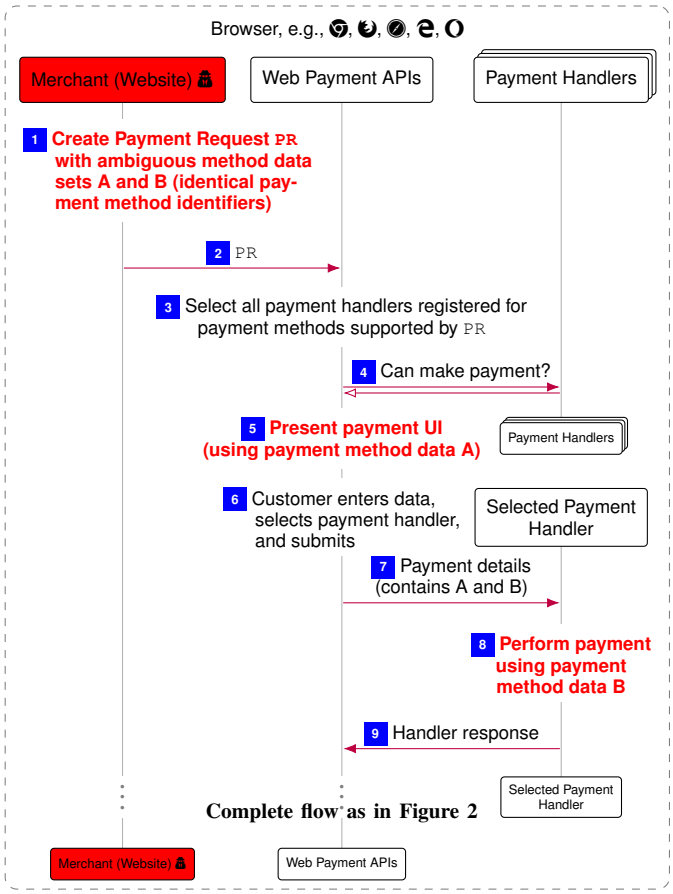

Figure 5 Attack with ambiguous payment method data: The customer sees a payment method data set without additional fees (5) while the payment handler "sees" one with a huge additional fee (8). In the end, the customer is charged a processing fee she never agreed to.

remaining steps are as usual. The result is that the customer is charged with a fee she never agreed to, without any indication. Note that this attack does not at all rely on how exactly the browser and payment handler break ties in case of multiple applicable method data entries, as long as there is a possibility that they choose different entries.

Implementation and Verification of the Attack. Using a similar setup as for the first attack, we verified that Google Chrome passes ambiguous method data to payment handlers. Hence, it is up to the payment handler to guess which entry the user selected leading to the problem as sketched above.

Uncovering the Attack. Similar to the previous attack, we found this attack while trying to prove the main security theorem (see Section VI) of the WPA. In particular, we have to show that if a payment provider performs a transaction, then there previously was a payment request event created by the respective browser with the corresponding values, i.e., the method data selected by the user.[8] As the browser does not tell the payment handler which method data entry the user selected, the payment handler has to infer this information based on the payment method identifier and the list of method data. As this list stems from an untrusted source (a potentially malicious merchant), there could be multiple entries for the same payment method identifier, making this information ambiguous. Hence, a payment handler can infer different data than the user selected and thus, the security theorem cannot hold true.

Fix. There are two ways to mitigate this attack: the browser could either reject ambiguous entries or propagate the user's choice to the payment handler. We recommended the first option to keep the API interface stable. This fix was adopted by the W3C Web Payments Working Group and we incorporated it into our model.

Together with the fix from Section III-A, we were able to prove the WPA secure in our model (see Sections V to VII).

C. Responsible Disclosure

We first notified the W3C Web Payments Working Group of the first attack on Nov. 2nd, 2019 [51] and (after checking their implementation) the Chrome developers on Nov. 25th, 2019 [15]. While the W3C Working Group at first did not acknowledge the problem, the Chrome developers fixed this problem on Jan. 25th, 2020 (and even released a hot-fix for their current stable version).

When we presented our full results to the W3C Working Group on Apr. 1st, 2020 [52, 53, 54], the W3C Working Group acknowledged all of our findings and updated their specification on May 11th and May 25th, 2020. The Chrome developers disallowed ambiguous method data in their implementation following the updated specification on May 27th, 2020 [16].

IV. The Web Infrastructure Model

Our formal security analysis of the WPA is based on the WIM, a generic Dolev-Yao style web model proposed by Fett et al. in [21]. Here, we only briefly recall this model following the description in [24] (see also [20, 21, 22, 23] for comparison with other models and discussion of its scope). Our extension of this model required for the analysis of the WPA, including aspects of the service worker standard [47], is presented in Section V-A.

The WIM is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The WIM defines a general communication model, and, based on it, web systems consisting of web browsers, DNS servers, and web servers as well as web and network attackers.

Communication Model. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes communicate via *events*, which consist of a message as well as a receiver and a sender address. In every step of a run of a system (see below), one event is chosen non-deterministically from a "pool" of waiting events and is delivered to one of the processes that listen to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events.

[8] For the fixed model (Lines 94 to 97 of Algorithm 1), we capture this property in Lemma 10 of Appendix E.

As usual in Dolev-Yao models (see, e.g., [1]), messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures). For example, in the web model, an HTTP request is represented as a term r containing a nonce, an HTTP method, a domain name, a path, URI parameters, request headers, and a message body. For instance, an HTTP request for the URI http://ex.com/show?p=1 is represented as $r := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{ex.com}, /\mathtt{show}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$ where the body and the list of request headers is empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}}))$, where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example, $\mathsf{dec_a}(\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}})), k_{\text{ex.com}}) \equiv \langle r, k'\rangle$, i.e., these two terms are equivalent w.r.t. the equational theory.

A *Dolev-Yao process* (*DY process*, in short) consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

The so-called *attacker process* is a DY process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any DY process could possibly perform. Attackers can corrupt other parties.

A *script* models JavaScript running in a browser. Scripts are defined similarly to DY processes, i.e., a script is a relation, typically specified as a non-deterministic algorithm. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below). Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

A *system* is a set of processes. A *configuration* of this system consists of the states of all processes in the system, the pool of waiting events, and a sequence of unused nonces. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation step. The transition from one configuration to the next configuration in a run is called a *processing step*. We write, for example, $Q = (\mathrm{S}, \mathrm{E}, \mathrm{N}) \to (\mathrm{S}', \mathrm{E}', \mathrm{N}')$ to denote the transition from the configuration (S, E, N) to the configuration $(\mathrm{S}', \mathrm{E}', \mathrm{N}')$, where S and S' are the states of the processes in the system, E and E' are pools of waiting events, and N and N' are sequences of unused nonces.

A *web system* formalizes the web infrastructure and web applications, and is defined as a tuple $(\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$. $\mathcal{W}$ denotes a set of DY processes and is partitioned into the sets Hon, Net, and Web. Hon is a set of honest processes, e.g., web browsers, web servers, or DNS servers (they might be corrupted by an attacker during the run of a system, though). Web and Net are the sets of *web attackers* (who can listen to and send messages from their own addresses only) and *network attackers* (who may listen to and spoof all addresses and therefore are the most powerful attackers). A web system further contains a set of scripts $\mathcal{S}$ (comprising honest scripts and the attacker script), and a mapping script from scripts to their string representation. E^0 is defined as an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \mathtt{TRIGGER}\rangle$ for every $a \in \cup_{p \in \mathcal{W}} I^p$, where I^p denotes the set of addresses of the process p.

Web Browsers. An honest browser formally is modeled as a DY process and thought to be used by one honest user, who is modeled as part of the browser. User actions, such as following a link, are modeled as non-deterministic actions of the web browser. User credentials are stored in the initial state of the browser and are given to selected web pages when needed. Besides user credentials, the state of a web browser contains (among others) a tree of windows and documents, cookies, and web storage data (localStorage and sessionStorage).

A *window* inside a browser contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window. Each represents one loaded web page and contains (among others) a script and a list of subwindows (modeling iframes). As sketched above, when triggered by the browser, a script is provided with all data it has access to, such as a (limited) view on other documents and windows, certain cookies, and web storage data. Scripts then output a command and a new state. This way, scripts can navigate or create windows, send XMLHttpRequests and postMessages, submit forms, set/change cookies and web storage data, and create iframes. Note that scripts—besides modeling JavaScript behavior—also model some aspects of user behavior. For example, if a web page provides some link on which in reality the user can click on, the script may non-deterministically instruct the browser to navigate the window to the URL of the link. Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the relevant web standards.

A browser interacts with the network using messages of different types: The browser sends DNS and HTTP(S) requests (including XMLHttpRequests), and it processes the responses. The model includes handling of relevant HTTP(S) headers, including, for example, cookie, location, strict transport security (STS), and origin headers. A browser, at any time, can also receive a so-called trigger message. This kind of message is used to invoke actions that model browser-related user behavior (e.g., navigate a window or entering some URL) as well as running a script (which includes some aspects of user behavior as well, see above). When receiving such a trigger message, the

browser non-deterministically chooses which action it takes. For instance, if the browser decides to trigger a script, the script is run and outputs a command as described above, which is then further processed by the browser. Browsers can also become corrupted, i.e., be taken over by web and network attackers. Once corrupted, a browser behaves like an attacker process.

V. Formal Model of the Web Payment APIs

We formalize the WPA based on the WIM as a web system that contains one network attacker (which is powerful enough to subsume an arbitrary number of web attackers and other network attackers), a finite but arbitrary number of browsers, as well as of merchants and payment providers. While the definition of attackers and browsers (outlined above and extended in Section V-A below) are part of the (generic) WIM, merchants and payment providers (modeled as web servers) as well as scripts that exercise the feature set of the WPA are defined specifically for the analysis of the WPA.

We highlight that, in order to create a detailed model of the WPA, a detailed model of the web infrastructure such as the WIM is needed. Otherwise, it is not straightforward to even describe the WPA, as the WPA define an extension to web browsers and a protocol that involves browsers and web servers. So to model the WPA faithfully, many web features are needed that, as mentioned before, the WIM already contains, including scripts, the notion of origins (and hence, the notion of schemes and domains), the window and document structure of the browser, post messages, HTTPS, and XMLHttpRequests. Besides these standard web features, the WPA require even more web features, namely DOM events, service workers and service workers registry, and an extension to the script API.

With the WIM extended with these features, the WPA can be modeled in a natural and direct way, which also helps to avoid modeling errors since one does not have to translate the WPA into some other (more artificial) modeling context.

In addition, in order for the security proof of the fixed WPA to be meaningful, the underlying model of the web infrastructure and the model of WPA itself should be as detailed as possible, because by this, we provably exclude large classes of attacks (see also the last paragraph in Section VI-C.)

In the following, we present these generic extensions of the WIM (see Section V-A). On top of this extended WIM, we then model payment handlers, payment method providers, and merchant web servers (see Section V-B).

A. Generic Extensions of the WIM

As sketched in the previous section, the WIM already has a very detailed model of browser behavior. This has paved the way for detailed analyses of several web-based authentication and authorization protocols [20, 21, 22, 23, 24, 25] which make heavy use of different HTTP/HTML features and JavaScript APIs, all of which are explicitly modeled in the WIM. Following this approach of explicitly modeling the relevant browser behavior—instead of using a very abstract, and hence, less meaningful, model—we significantly extended the WIM's browser model to incorporate the WPA. More specifically, our core extensions to the browser model are: We add an extensible mechanism to trigger and process DOM events, introduce service workers and their execution, and add the WPA and extend script execution to incorporate these APIs. Note that the first two extensions are largely independent of the WPA: They model separate standards of independent interest and are used by, but not limited to the WPA.

Extensible DOM Event Processing. The WPA make heavy use of DOM events,[9] e.g., to trigger payment handlers. Such DOM events allow for signaling that something has occurred to registered event listeners.

We extended the WIM's browser with a set of pending events and an extensible function to process such events. This function models the registered event listeners. Whenever the modeled browser is triggered to process a DOM event, one of the pending events is chosen non-deterministically, removed from the set of pending events and handed to the processing function. We do not restrict the format of DOM events in any way except for the first member, which must state the type of the event. This information is used to dispatch events to the respective event listeners and resembles the `type` member of events in the DOM standard. We provide the model of DOM event processing in Appendix D.

Service Workers. Service workers are event-driven JavaScript programs that run inside a browser in the background and can, for example, be used to provide some offline functionality for web applications. In the context of the WPA, payment handlers are instances of service workers. Hence, we extended the WIM's browser with a set of registered service workers. Similar to the event processing, a service worker can be chosen non-deterministically whenever a browser is triggered. That service worker is then executed similarly to a script. As a service worker has a slightly different view on the browser's state (e.g., it cannot access the window structure) and slightly different capabilities than a script, we reflect these differences in the input of the service worker and the possible commands it can output to the browser. The possible commands which a service worker can output include triggering certain events, sending XMLHttpRequests and postMessages as well as opening new browser windows. In addition to these general commands, we also model the (de)registration of payment instruments, i.e., supported payment methods, by payment handlers.

Script API Extensions. The WPA themselves are a notable extension to the WIM's browser model. As mentioned, all major browsers have implemented the WPA. As a consequence, web applications and scripts, honest and dishonest, might (on purpose or accidentally) interfere with one of the various parts of the WPA, changing the properties of such applications and scripts in a real-world execution context, thereby widening the attack surface.

In our WIM extension, the actual API functions defined by the WPA specifications are modeled as script commands, i.e., a script can output a command instructing the browser to call one of the API functions with a given list of arguments. For example, a script on some merchant's website can call

[9] See https://dom.spec.whatwg.org/#event for more details on DOM events.

an API function to indicate completion of a payment. The formal definitions of these functions closely follow the WPA standards, which we described informally in Section II. In the following, we describe the main parts of the script API extension of the WIM, which is shown in Algorithm 1. The complete browser model, including the full script API with all extensions is provided in Appendix D.

Algorithm 1 specifies the function RUNSCRIPT which is part of the browser definition and is responsible for executing a script. Roughly speaking, in the WIM, a browser can run any script of any (active) document at any time. To this end, the browser (from its state s) non-deterministically selects a document (identified by a pointer $\overline{d}$) that is the active document of some window (identified by a pointer $\overline{w}$) and executes the function RUNSCRIPT with this data. This function then assembles all (state) information that a script is allowed to access, runs that script, and finally processes the output of the script (a command and new state information).

The first part of RUNSCRIPT (Lines 2–7) assembles the information passed to the script depending on the position of the script's document in the window tree and the origin of that document. This data includes a limited view on the window tree (produced by the function Clean), cookies accessible to the script, web storage (local and session storage), as well as secrets that a user would potentially enter into that document (recall that user behavior is modeled as part of the browser).

Next, the script is executed (Lines 8–10). Formally, the function treats a script as a relation (identified by a string) and non-deterministically selects one possible outcome of that script (see Line 10, where $\mathcal{T}_{\mathcal{N}}(V)$ denotes the set of all terms with nonces in $\mathcal{N}$ and variables in V). The function then updates the browser's state accordingly taking care of freshly chosen nonces,[10] cookies, and web storage.

Finally, RUNSCRIPT processes the so-called *command* emitted by the script. The command models an API call and can, for example, instruct the browser to open, close, or navigate a window, start an XHR, or send a postMessage (defined in Lines 16-91, not shown here). To model the respective functionalities introduced by the WPA, we extend RUNSCRIPT by adding cases for all API functions exposed to a script (Lines 91ff.), where here we only show an excerpt of these functions (leaving out Lines 126ff.). We emphasize that our analysis, of course, covers the full definition of browsers, which is contained in Appendix D.

If the command created by the script is a PR_CREATE command (Lines 91ff.) with the arguments $methodData$, $details$, and $options$, the browser creates a payment request with these values (see Steps 1–2 in Section II-C and Section 3.1 of [14]). In this case, the browser verifies if $methodData$ is well-formed (see Definition 55 in Appendix C) and not ambiguous (our fix for the second attack, see Section III-B).

The browser then assembles a term $paymentReq$ that is (1) used to internally track this payment request in the browser's state (Line 100) and (2) used as a return value for the script

[10]Nonces chosen by the script are denoted by the placeholders λ_i and then mapped to fresh nonces chosen by the browser relation, which are denoted by ν_j (see Line 10).

Algorithm 1 Web Browser Model: Execute a script.

1: **function** RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
2: **let** $tree$:= Clean($s', s'.\overline{d}$)
3: **let** $cookies$:= $\langle \{ \langle c.\texttt{name}, c.\texttt{content.value} \rangle |$
↪ $c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right] \wedge$
↪ $c.\texttt{content.httpOnly} = \bot \wedge$
↪ $\big(c.\texttt{content.secure} \implies$
↪ $(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S})\big)\}\rangle$
4: **let** $tlw \leftarrow s'.\texttt{windows}$ **such that**
↪ tlw is the top-level window containing $\overline{d}$
5: **let** $sessionStorage$:=
↪ $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$
6: **let** $localStorage$:= $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
7: **let** $secrets$:= $s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
8: **let** $R \leftarrow \texttt{script}^{-1}(s'.\overline{d}.\texttt{script})$
9: **let** in := $\langle tree, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}$,
↪ $s'.\overline{d}.\texttt{scriptinputs}$, $cookies$,
↪ $localStorage$, $sessionStorage$, $s'.\texttt{ids}$, $secrets\rangle$
10: **let** $state'$ $\leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
↪ $cookies' \leftarrow \texttt{Cookies}^{\nu}$, $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
↪ $sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
↪ $out := \langle state', cookies', localStorage'$,
↪ $sessionStorage', command\rangle$
↪ **such that** $out = out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$
↪ **with** $(in, out^{\lambda}) \in R$
11: **let** $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$:=
↪ CookieMerge($s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$,
↪ $cookies'$)
12: **let** $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$:= $localStorage'$
13: **let** $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$:=
↪ $sessionStorage'$
14: **let** $s'.\overline{d}.\texttt{scriptstate}$:= $state'$
15: **switch** $command$ **do**

→ In this excerpt, we focus on the extension for the WPA and refer to Appendix D for the full model. We use the same line numbers as in Appendix D.

Extension with Payment APIs

91: **case** $\langle \texttt{PR\_CREATE}, methodData, details, options\rangle$
92: **if** $\neg(methodData \in \text{MethodDatas})$ **then stop** $\langle\rangle$, s'
93: **if** $methodData = \langle\rangle$ **then stop** $\langle\rangle$, s'
94: **let** $seenPMIs := \langle\rangle$
95: **for each** $\langle pmi, recv, paymentId\rangle \in methodData$ **do**
96: **if** $pmi \in seenPMIs$ **then stop** $\langle\rangle$, s'
→ Fix for second attack (ambiguous method data)
97: **let** $seenPMIs := seenPMIs +^{\langle\rangle} pmi$
98: **let** $paymentReq := \langle \texttt{PAYMENTREQUEST}, \nu_{14}, s'.\overline{d}.\texttt{nonce}$,
↪ $methodData, details, options, \langle\rangle, CR, \bot, \langle\rangle\rangle$
99: **let** $transactionId := \nu_{18}$
100: **let** $s'.\texttt{paymentStorage}[\nu_{14}]$:=
↪ $\langle paymentReq, \langle\rangle, \langle\rangle, transactionId\rangle$
101: **let** $s'.\overline{d}.\texttt{scriptinputs}$
↪ := $s'.\overline{d}.\texttt{scriptinputs} +^{\langle\rangle} paymentRequest$
102: **stop** $\langle\rangle$, s'
103: **case** $\langle \texttt{PR\_SHOW}, PRN, detailsUpdate\rangle$
104: **let** $paymentReq$:=
↪ $s'.\texttt{paymentStorage}[PRN].\texttt{paymentReq}$
105: **if** $paymentReq.\texttt{state} \neq CR$ **then stop** $\langle\rangle$, s'
106: **if** $s'.\overline{w}.paymentRequestShowing = \top$ **then**
107: **let** $s'.\texttt{paymentStorage}[PRN].\texttt{paymentReq}$
↪ $.\texttt{state} := CL$
108: **stop** $\langle\rangle$, s'
109: **let** $s'.\texttt{paymentStorage}[PRN].\texttt{paymentReq}$
↪ $.\texttt{state} := IN$

```
110:  let s'.w̄.paymentRequestShowing := ⊤
111:  let handlers := ⟨⟩
112:  for each mds := ⟨pmi, receiver, paymentIdentifier⟩ ∈
        ↪ paymentReq.methodData do
113:    let ph := GET_PAYMENT_HANDLERS(pmi, s')
114:    let handlers := handlers +^⟨⟩ ph
115:    for each handler ∈ ph do
116:      let s'.events := s'.events+^⟨⟩
            ↪ ⟨CANMAKEPAYMENT, handler.nonce,
            ↪ tlw.origin, s'.d̄.origin, mds⟩
117:  let handler ← handlers
118:  let s'.paymentStorage[PRN].handlerNonce :=
        ↪ handler.nonce
119:  if detailsUpdate ≠ ⟨⟩ then
120:    let s'.paymentStorage[PRN].paymentReq
          ↪ .updating := ⊤
121:    let s'.events := s'.events+^⟨⟩
          ↪ ⟨PR_UPDATE_DETAILS, PRN, detailsUpdate⟩
122:  let s'.events := s'.events+^⟨⟩
        ↪ ⟨SUBMITPAYMENT, PRN, handler.nonce⟩
123:  stop ⟨⟩, s'
```

that issued this command (Line 101). This term contains a fresh nonce ν_{14} as the payment request nonce (used as a unique identifier of a payment request within the browser), the browser's internal identifier for the document in which the script was executed ($s'.\overline{d}$.nonce), the arguments of the command, as well as other state information of the payment request (see below and also Definition 61 in Appendix C). The browser further chooses what we call the transaction identifier (a fresh nonce ν_{18}) that we use in our model to identify the corresponding monetary transaction. We note that our model at no place uses this value for any kind of decision and only passes this value along with the data specified by the WPA.

If the command created by the script is a PR_SHOW command (Lines 103ff.) with a payment request nonce PRN and a $detailsUpdate$ entry, the browser starts the payment process (subsumed in Step 2 in Section II-C and described in Section 3.3 of [14]).

First, the browser looks up the information about the payment request in its state (using the payment request nonce) and checks whether the payment request has not been processed yet (indicated with the string CR for *created*). If the payment request has already been processed or if the browser window is currently processing another payment request, the browser aborts and, in the latter case, changes the state of the payment request to CL for *closed* (see Lines 105–108).

Next, the browser stores that it is now processing this payment request by setting the paymentRequestShowing flag of the current window to true and setting the state of the payment request to IN for *interactive*. Then, the browser determines a payment handler for this payment (Lines 111–117). To this end, the browser selects the payment handlers that support the payment method identifiers given in the payment request using the GET_PAYMENT_HANDLERS function (see Algorithm 25 for the definition of this function). For each of the selected handlers, the browser creates a CanMakePaymentEvent event (see also Section II-C). Instead of waiting for the responses of the payment handlers, the browser immediately non-deterministically chooses one of the previously selected payment handlers (Line 117); a safe over-approximation for a user choosing a payment handler that is willing to process the payment.

Further, the merchant script can provide updates to the payment request using the argument $detailsUpdate$.

Finally, the browser creates a SubmitPaymentEvent in Line 122, indicating (within the model) that it accepts the payment. This event will be processed in a future processing step of the browser.

B. Instantiating Relevant Service Workers and Servers

Based on the extended WIM, we can now specify generic payment providers and merchants as servers, and payment handlers as service workers. We note that the exact behavior of these parties is out of the scope of the specifications, so we modeled them with minimal assumptions.

Payment Provider. We modeled the payment provider as an HTTPS server with three endpoints that reflect typical interactions with a payment provider: /index, /authenticate, and /pay. When receiving a request to the /index endpoint, the payment provider returns a script that, when executed in the browser, sends a request with the customer's credentials to the /authenticate endpoint, modeling the user entering her password in order to authenticate.

Upon receiving a request at the /authenticate endpoint, the payment provider checks whether the request contains an identity (i.e., user name) and a secret (i.e., password) and compares those with known login credentials. If this check succeeds, the payment provider creates a fresh nonce, called token, and stores this token in its state, associated with the identity of the authenticated user. Afterward, the payment provider responds with a message containing the token. This message is eventually delivered to the aforementioned script that sent the authentication request. Once that script receives the token message, it forwards it to the browsing context that originally opened the /index page via postMessage – this will usually be a payment handler.

At the /pay endpoint, the payment provider expects one of the tokens it has issued via the /authenticate endpoint, and information about the transaction that it should perform. In particular, the receiver and total amount of the payment—the sender is determined by looking up the identity associated with the received token. In our model, the transaction is performed by storing the payment information, i.e., sender, receiver, amount, and the payment id assigned to the original payment request, in the transactions map of the payment provider's state.

Note that during a retry, the payment provider will receive a second request to the /pay endpoint with updated payment information. In that case, the request's payment identifier will be the same as in the original /pay request and the payment provider updates the corresponding entry in its transactions map. This models a payment provider canceling the original transaction and replacing it with a new one.

Payment Handler. As described in Section II, payment handlers are service workers and we model them as such. We implement a generic payment handler in our model. In order to also model a simple (and probably typical) payment method manifest (see Section II-B), we require that a payment handler is installed under the respective payment provider's origin. This models a payment method manifest that only allows payment handlers to be installed by the payment provider itself. Note that this restriction does not limit the number of payment methods provided by a payment handler: one payment provider can offer multiple payment methods.

When our generic payment handler receives payment details to initiate a payment on behalf of the customer (see 7 in Figure 2), it opens a window with the `/index` page at its own, i.e., the payment provider's, origin.

As explained before, our generic payment provider responds with a script that acquires a token from the payment provider for the customer's identity – modeling a login page at which the customer authenticates herself – which is then forwarded to the payment handler.

After receiving such a token, the payment handler sends a request to the `/pay` endpoint of the payment provider containing the token and the relevant payment information, i.e., the receiver and total amount (the sender is determined by examining the token). Finally, the payment handler creates a payment handler response and hands it back to the WPA in the browser (see 9 in Figure 2).

We note that our model of payment handlers does not send responses to `CanMakePaymentEvents`. This is a safe over-approximation as discussed in Section V.

Merchant. We model a generic merchant as a HTTPS server that, upon receiving a request at its `/index` endpoint, serves a script script_merchant to the browser which uses the WPA as sketched in Section II-C.

C. The WPA Web System

In the following, we model the WPA protocol in the extended WIM as a class $\mathcal{WPAPI}$ of web systems (see Section IV for the definition of web systems). In the following sections, we show that *all* web systems in that class satisfy certain desired security properties, which implies that they hold true for arbitrary numbers of browsers, payment providers, etc., running concurrently. We refer to $\mathcal{WPAPI}$ also as our *WPA model*.

Definition 1. A web system $(\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ belongs to $\mathcal{WPAPI}$ iff the following conditions are satisfied:

- $\mathcal{W}$ is partitioned into the sets Hon and Net. Net includes a network attacker process and Hon consists of a finite set of web browsers B, a finite set of web servers for the merchants C and a finite set of payment provider servers PP with $\mathsf{Hon} := B \cup C \cup PP$.

  The browsers are those introduced in Section V-A. We highlight that each browser of a $\mathcal{WPAPI}$ web system can have an arbitrary number of payment handlers. Honest payment provider servers, merchant servers, and payment handlers behave as sketched in Section V-B, with formal definitions given in Appendix D. DNS servers are subsumed by the adversary, i.e., they are all dishonest, but we assume a PKI. Initially, all participants (except the network attacker) are honest, but can be dynamically corrupted by the network attacker during the run of a web system.
- $\mathcal{S} = \{\text{script\_merchant}, \text{script\_payment\_provider\_index}, \text{script\_default\_payment\_handler}\}$ contains the scripts of the model as sketched in Section V-B and formally defined in Appendix D.
- script is a mapping from scripts to their string representation, where
  - $\mathsf{script}(\text{script\_merchant}) = \texttt{script\_merchant}$
  - $\mathsf{script}(\text{script\_payment\_provider\_index}) = \texttt{script\_payment\_provider\_index}$
  - $\mathsf{script}(\text{script\_default\_payment\_handler}) = \texttt{script\_default\_payment\_handler}$.

We call a web system in $\mathcal{WPAPI}$ a *WPA system*.

VI. Security Properties

In the following, we describe the security properties that the WPA should fulfill. These properties reflect natural integrity properties that one would expect from every payment system. In a nutshell, these properties state the following: (1) Whenever some (honest) payment provider performs a financial transaction on behalf of a customer, the customer has expressed the wish to do so. This property is called *Intended Payment* and described in Section VI-A. (2) Each payment that the customer authorizes is performed at most once, and if it is performed, the relevant aspects of the transaction are exactly what the customer authorized. This property is called *Uniqueness of Payments* and described in Section VI-B.

In the formalization of these properties, we make use of two important data structures, which record the user's intent to pay and transactions performed by the payment provider: A browser stores (in its state) all payments for which the browser's user expressed intent (i.e., she selected a payment handler and submitted the payment in the payment UI) in a dictionary called $paymentIntents$. Each payment provider in turn stores (in its state) all performed transactions in a map called $transactions$. In order to be able to relate transaction intents in a browser to transactions in a payment provider's state, we identify each transaction by a $transactionId$ chosen uniquely by the browser.

We note that there clearly cannot be a one-to-one mapping between the payment intents stored in the browser and the transactions stored by the payment provider as for some intents there may not be a corresponding transaction: a network attacker can hold back requests indefinitely.

Also, since the WPA do not specify how a merchant can check that a financial transaction was successfully initiated by the payment provider, we do not consider properties that would give the merchant such guarantees.

In the following, we give formal definitions for the aforementioned properties. We highlight that both properties do not require the involved merchant to be honest, thus, hold true even if a customer interacts with a malicious merchant. However, the payment provider and handler involved in the corresponding transaction need to be honest; all other parties

not directly involved in the transaction, such as other browsers, handlers, and payment providers, may have been corrupted by the adversary at any point in time.

A. Intended Payments

From the customer's viewpoint, it is important that a payment performed by the payment provider in their name corresponds to a payment the customer authorized. In particular, this means that the sender, receiver, and total amount of the transaction coincide with what the customer confirmed in the payment UI of her browser. This implies that no malicious party can initiate a payment on behalf of an (honest) customer.

For the formal definition of this *Intended Payments* property we define a helper function that maps a transaction identifier $txId$ to the corresponding payment intent in a browser state (stored under that transaction identifier). Note that there might be multiple payment intents in the case of retries.

Definition 2 (Intents of a browser given a transaction identifier). For a configuration (S, E, N) of a run ρ of a WPA system in $\mathcal{WPAPI}$, a browser b in B, and a nonce $txId$, we define $\mathsf{intents}(\cdot,\cdot,\cdot)$ as follows: $\mathsf{intents}(\mathrm{S}, b, txId) := \mathrm{S}(b).\texttt{paymentIntents}[txId]$

We can now formalize the *Intended Payments* property (with the explanation following the definition); π denotes projections on list entries.

Definition 3 (Intended Payments). A WPA system in $\mathcal{WPAPI}$ fulfills *Intended Payments* iff for every run ρ of this system, every configuration (S, E, N) in ρ, every payment provider server $pp \in PP$ honest in S, and every $t \in \mathrm{S}(pp).\texttt{transactions}$ it holds true that:
If $b := \texttt{ownerOfID}(t.\texttt{sender}) \in B$ is a browser honest in S, then $\exists i \in \mathbb{N}$ such that $t.\texttt{total} = \pi_i(\mathsf{intents}(\mathrm{S}, b, t.\texttt{txId})).\texttt{details.total} \wedge t.\texttt{receiver} = \pi_1(\pi_i(\mathsf{intents}(\mathrm{S}, b, t.\texttt{txId})).\texttt{methodData}).\texttt{receiver}$.

The *Intended Payments* property ensures that for each transaction stored at an honest payment provider for some honest sender's account, the sender's browser holds a corresponding intent (identified by the transaction identifier) with the same receiver and total. As already mentioned, the behavior of the customer is subsumed by the browser, with the mapping from customer to browser defined by the mapping $\mathsf{ownerOfID}$.

We require that both the involved browser of the customer as well as the payment method provider are honest—otherwise, all is lost anyway w.r.t. *Intended Payments*. However, as already mentioned above, we do not require that merchants are honest, i.e., this property holds true even for malicious merchants involved in the payment process.

B. Uniqueness of Payments

As mentioned, intuitively, *Uniqueness of Payments* ensures that for each payment that the customer authorizes, there is at most one transaction executed by any honest payment provider, and that this transaction has the correct values.

Definition 4 (Uniqueness of Payments). A WPA system in $\mathcal{WPAPI}$ fulfills *Uniqueness of Payments* iff for every run ρ of this system, every configuration (S, E, N) in ρ, every browser $b \in B$ honest in S, every $(txId, intents) \in \mathrm{S}(b).\texttt{paymentIntents}$, and with $PP_h = \{pp \in PP : \mathrm{S}(pp).\texttt{isCorrupted} = \bot\}$ being the set of payment providers that are honest in S, then $|\cup_{pp \in PP_h} \{t \in \mathrm{S}(pp).\texttt{transactions} \mid txId = t.\texttt{txId} \wedge b = \mathsf{ownerOfID}(t.\texttt{sender})\}| \leq 1$. If such a t exists, there exists a $pi \in intents$ such that $t.\texttt{total} = pi.\texttt{details.total}$ and $t.\texttt{receiver} = \pi_1(pi.\texttt{methodData}).\texttt{receiver}$.

This property requires that for every payment intent stored in an honest browser (identified by $txId$), there is at most one corresponding transaction with that browser's user as the sender (payer) in the combined state of all honest payment providers. If such a transaction exists, then we require that the receiver and total of the transaction correspond to the values of one of the intents stored by the browser for $txId$.

C. Security Theorem

The following theorem states that the WPA protocol is secure, i.e., fulfills both the Intended Payments property and the Uniqueness of Payments property. We refer to Appendix E-D for the full proof of this theorem; we provide a proof sketch in Section VII.

Theorem 1. All web systems in $\mathcal{WPAPI}$ fulfill the Intended Payments and the Uniqueness of Payments properties.

We emphasize that our modeling and analysis takes into account a powerful attacker with capabilities way beyond the usual network attacker used in protocol analyses as the WIM and its extension presented here is a detailed model of the web infrastructure, in fact the most comprehensive one to date: Our attacker not only completely controls the network but, as mentioned, also the whole DNS system. He can also make use of the various headers supported by the model. Furthermore, an honest browser may visit malicious websites with malicious scripts which run in the browser—in parallel to (possibly multiple) WPA sessions. This, of course, also allows the attacker to use the whole set of supported web features in the browser, including sending postMessages to honest browsing contexts, making requests to (honest) servers from an honest browser (cross site request forgery attacks), trigger arbitrary events, access data stored in the browser (complying with access restrictions defined in the various web standards, e.g., related to the same-origin policy), etc. This detailed view makes it possible to exclude subtle attacks emerging from delicate details in how different web technologies inter-operate.

We note that our analysis does not cover privacy properties, as privacy is not a central concern of the WPA. In particular, the WPA intentionally "leaks" privacy-related information. For example, the user's address is (partially) provided to the merchant even before the user approves the final payment (see also Section II-C). Also, payment details may be released to the merchant, e.g., credit card details when using the `basic-card` payment method (see Section II-B). Moreover, the protocol does not intend to hide the identity of the merchant or the kind of goods paid for from the payment provider.

VII. Proof Sketch

The proof of Theorem 1 is split into 14 lemmas. To give an impression of the proof, we show the proof of the

Intended Payments property (see Lemma 2 below), which in turn is proven using two key lemmas (with one based on further lemmas). We here give the proof of one of these key lemmas (Lemma 1) and state (and prove) the second lemma (Lemma 10) in Appendix E. To give an impression of the proof of Uniqueness of Payments, we provide a high-level proof sketch in Appendix A-B. Full proofs of all lemmas are provided in Appendix E.

A. Relation between Payment Request Events and Payment Intents

The following lemma relates payment request events of a browser to the payment intents it stores. As introduced in Section VI, the user's intent to submit a payment is modeled by adding the corresponding payment request event to the `paymentIntents` entry of the browser state. Additionally, the browser model has a state entry `events`, where it stores events that it may process at any time.

More precisely, Lemma 1 states that for every payment request event stored in the `events` entry of the browser state of an honest browser, there is a payment intent stored by the browser with the same total and the same receiver (in the first element of the `methodData` entry). Furthermore, the payment intent is stored using the transaction identifier of the payment request event as a key.

Lemma 1 (Payment Request Event Implies Payment Intent). For every WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every browser $b \in B$ that is honest in S, every payment request event (as defined in Definition 64) $preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$, it holds true that $\exists pi$.

$$pi \in^{\langle\rangle} \mathrm{S}(b).\texttt{paymentIntents}[preqEvent.\texttt{txId}] \wedge$$
$$pi.\texttt{details.total} = preqEvent.\texttt{total} \wedge$$
$$\pi_1(pi.\texttt{methodData}).\texttt{receiver} =$$
$$\pi_1(preqEvent.\texttt{methodData}).\texttt{receiver}.$$

Proof: Let $preqEvent$ be a payment request event such that $preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$, and b a browser honest in S. Initially, the `events` state entry of the browser is empty (per Definition 69 of Appendix D). An honest browser adds payment requests events to its `events` state entry only in Line 22 of Algorithm 22. After this line is executed, the browser will execute Line 25 of the same algorithm (as there is no `stop` between these lines), in which the browser stores a payment intent pi. The dictionary key used for storing the intent is the transaction identifier stored in $preqEvent$, i.e., $preqEvent.\texttt{txId}$ (see Line 20 and Line 25 of Algorithm 22).

Moreover, there exists a payment request $paymentReq$ such that $pi.\texttt{details.total} = paymentReq.\texttt{details.total}$ (Line 23 of Algorithm 22). The value of the total of $preqEvent$ is set to the value stored in $paymentReq$, i.e., $preqEvent.\texttt{total} = paymentReq.\texttt{details.total}$ (Line 10 and Line 20 of Algorithm 22). Therefore, we conclude that $pi.\texttt{details.total} = preqEvent.\texttt{total}$.

The `methodData` entry stored in the payment intent pi in Line 24 is the same that is stored in the payment request event $preqEvent$ in Line 20 of Algorithm 22, i.e., $\pi_1(preqEvent.\texttt{methodData}).\texttt{receiver} = \pi_1(pi.\texttt{methodData}).\texttt{receiver}$. ■

B. Proof of Intended Payments

The proof of the Intended Payments property uses Lemma 10, which we give in Appendix E. Informally, Lemma 10 relates transactions stored in the state of a payment provider server to payment request events stored at a browser. More precisely, each transaction contains a sender, and the lemma states that the browser that manages the identity of this sender stores a corresponding payment request event, i.e., with the same total and the first `methodData` entry of the event having the same receiver as in the transaction.

Lemma 2 (Intended Payments). Every WPA system in $\mathcal{WPAPI}$ fulfills Intended Payments (see Definition 3).

Proof: Let t be a transaction stored in the state of an honest payment provider pp, i.e., $t \in \mathrm{S}(pp).transactions$.

We apply Lemma 10 and conclude that, if $b := \texttt{ownerOfID}(t.\texttt{sender}) \in B$ is a browser honest in S, then $\exists\ preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ such that

$$\pi_1(preqEvent) = \texttt{PAYMENTREQUESTEVENT} \wedge t.\texttt{txId} = preqEvent.\texttt{txId} \wedge t.\texttt{total} = preqEvent.\texttt{total} \wedge \pi_1(preqEvent.\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver}.$$

Next, we apply Lemma 1 and conclude that there is a payment intent pi such that

$$pi \in^{\langle\rangle} \mathrm{S}(b).\texttt{paymentIntents}[preqEvent.\texttt{txId}] \wedge pi.\texttt{details.total} = preqEvent.\texttt{total} \wedge \pi_1(pi.\texttt{methodData}).\texttt{receiver} = \pi_1(preqEvent.\texttt{methodData}).\texttt{receiver},$$

and, in particular,

$$pi \in^{\langle\rangle} \mathrm{S}(b).\texttt{paymentIntents}[t.\texttt{txId}] \wedge pi.\texttt{details.total} = t.\texttt{total} \wedge \pi_1(pi.\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver}.$$ ■

VIII. Conclusion

In this paper, we performed the first in-depth and formal analysis of the W3C WPA. To the best of our knowledge, our analysis is the first such analysis of any web payment system, certainly the first in a detailed web infrastructure model.

Our analysis is based on the most comprehensive model of the web infrastructure to date, the WIM. To enable the analysis, we significantly extended the WIM, a contribution of independent interest. In addition to extending the browser model with the WPA, we added a framework for service workers and an extensible mechanism to trigger and process DOM events in the browser. Based on this model, we formulated precise security properties that reflect the integrity of payments performed using WPA: *Intended Payments* and *Uniqueness of Payments*.

While trying to prove these properties, we found two critical vulnerabilities that enable a malicious merchant to over-charge an unsuspecting customer. We proposed fixes that prevent these attacks and formally verified the security of the WPA with our fixes in place. This is of direct practical relevance as the WPA enjoy wide industry support and are expected to be adopted by many merchants and payment providers in the near future.

We also verified these attacks in Google Chrome, at the time of analysis one of the first browsers that implemented the WPA. We reported our findings to the responsible working group at the W3C as well as the Google Chrome developers, who both acknowledged the issues. The working group adapted the specification of the WPA according to our proposals and the Chrome developers implemented the fixes and even released a hotfix for the current Chrome version.

We are currently working on a mechanized model for the WIM based on a recent verification framework called DY* [8], a new approach for the modular symbolic security analysis of protocol code written in the F* programming language. It is interesting future work to carry out analyses as performed here with such a tool.

Acknowledgments

This work was partially supported by Deutsche Forschungsgemeinschaft (DFG) through Grant KU 1434/12-1 and Grant KU 1434/10-2.

References

[1] M. Abadi and C. Fournet. "Mobile Values, New Names, and Secure Communication". In: *POPL*. ACM Press, 2001, pp. 104–115.

[2] Apple. *Apple Pay on the Web*. URL: https://developer.apple.com/documentation/apple_pay_on_the_web (Retrieved 12/03/2020).

[3] M. Backes, M. Maffei, and K. Pecina. "A Security API for Distributed Social Networks". In: *NDSS'11*. Vol. 11. 2011, pp. 35–51.

[4] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. "Bitcoin as a Transaction Ledger: A Composable Treatment". In: *CRYPTO*. Vol. 10401. LNCS. Springer, 2017, pp. 324–356.

[5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. "Discovering Concrete Attacks on Website Authorization by Formal Analysis". In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

[6] D. Basin, R. Sasse, and J. Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *IEEE S&P*. IEEE Computer Society, 2021.

[7] R. Berjon et al., eds. *HTML5, W3C Recommendation*. Oct. 28, 2014. URL: http://www.w3.org/TR/html5/.

[8] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele. "DY*: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code". In: *EuroS&P 2021*. To appear. IEEE Computer Society, 2021.

[9] A. Bohannon and B. C. Pierce. "Featherweight Firefox: formalizing the core of a web browser". In: *WebApps'10*. USENIX Association, 2010, pp. 11–11.

[10] M. Bond, O. Choudary, S. J. Murdoch, S. P. Skorobogatov, and R. J. Anderson. "Chip and Skim: Cloning EMV Cards with the Pre-play Attack". In: *IEEE S&P*. IEEE Computer Society, 2014, pp. 49–64.

[11] E. Börger, A. Cisternino, and V. Gervasi. "Contribution to a Rigorous Analysis of Web Application Frameworks". In: *ABZ 2012*. Vol. 7321. LNCS. Springer, 2012, pp. 1–20.

[12] M. Cáceres, D. Denicola, Z. Koch, R. McElmurry, and A. Bateman. *Payment Method Identifiers*. Tech. rep. https://www.w3.org/TR/2019/CR-payment-method-id-20190905/. W3C, Sept. 2019.

[13] M. Cáceres, D. Denicola, Z. Koch, R. McElmurry, and A. Bateman. *Payment Method: Basic Card*. Tech. rep. https://www.w3.org/TR/2020/WD-payment-method-basic-card-20200213/. W3C, Feb. 2020.

[14] M. Cáceres, D. Denicola, Z. Koch, R. McElmurry, I. Jacobs, R. Solomakhin, and A. Bateman. *Payment Request API*. Tech. rep. https://www.w3.org/TR/2019/CR-payment-request-20191212/. W3C, Dec. 2019.

[15] Chromium Bug Tracker. *Issue 1028098: Disable switching payment method during retry*. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1028098.

[16] Chromium Bug Tracker. *Issue 1085712: Disallow duplicate payment method identifiers*. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=1085712.

[17] S. Delaune, S. Kremer, and G. Steel. "Formal Analysis of PKCS# 11". In: *CSF*. IEEE Computer Society, 2008, pp. 331–344.

[18] R. Dhamija, J. D. Tygar, and M. A. Hearst. "Why phishing works". In: *CHI 2006*. ACM, 2006, pp. 581–590.

[19] J. Dreier, A. Kassem, and P. Lafourcade. "Formal Analysis of E-Cash Protocols". In: *SECRYPT*. SciTePress, 2015, pp. 65–75.

[20] D. Fett, P. Hosseyni, and R. Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *IEEE S&P*. IEEE Computer Society, May 2019, pp. 1054–1072.

[21] D. Fett, R. Küsters, and G. Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System". In: *IEEE S&P*. IEEE Computer Society, 2014, pp. 673–688.

[22] D. Fett, R. Küsters, and G. Schmitz. "Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web". In: *ESORICS*. Vol. 9326. LNCS. Springer, 2015, pp. 43–65.

[23] D. Fett, R. Küsters, and G. Schmitz. "SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web". In: *ACM CCS*. ACM, 2015, pp. 1358–1369.

[24] D. Fett, R. Küsters, and G. Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *ACM CCS*. ACM, 2016, pp. 1204–1215.

[25] D. Fett, R. Küsters, and G. Schmitz. "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines". In: *CSF*. IEEE Computer Society, 2017.

[26] J. Fingas. *StockX confirms it was hacked*. 2019. URL: https://www.engadget.com/2019-08-03-stockx-hacked.html.

[27] Forter. *Why a Friction-Filled Online Checkout Process Causes Shopping Cart Abandonment*. 2019. URL: https://www.forter.com/blog/infographic-customers-wont-tolerate-friction-filled-checkout/.

[28] J. A. Garay, A. Kiayias, and N. Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *EUROCRYPT*. Vol. 9057. LNCS. Springer, 2015, pp. 281–310.

[29] Google. *Google Pay API PaymentRequest Tutorial*. URL: https://developers.google.com/pay/api/web/guides/paymentrequest/tutorial (Retrieved 12/03/2020).

[30] Google Developers. *Introduction to the Payment Request API*. 2019. URL: https://developers.google.com/web/ilt/pwa/introduction-to-the-payment-request-api.

[31] J. Hodges, J. Jones, M. B. Jones, A. Kumar, and E. Lundberg. *Web Authentication: An API for accessing Public Key Credentials*. Tech. rep. https://www.w3.org/TR/webauthn/. W3C, 2021.

[32] A. Hope-Bailie, A. Lyver, I. Jacobs, R. Solomakhin, J. Bang, T. Thorsen, and A. Roach. *Payment Handler API*. Tech. rep. https://www.w3.org/TR/2019/WD-payment-handler-20191021/. W3C, Oct. 2019.

[33] T. Jeong. *Cashing in on the JavaScript Payment Request API*. 2020. URL: https://blog.logrocket.com/javascript-payment-request-api/.

[34] E. Kitamura. *Integrating the Payment Request API with a payment service provider*. 2017. URL: https://medium.com/dev-channel/integrating-the-payment-request-api-with-a-payment-service-provider-b6a23aa44bd6.

[35] A. Kumar. "Using automated model analysis for reasoning about security of web protocols". In: *ACSAC 2012*. ACM, 2012, pp. 289–298.

[36] A. Kumar. "A Lightweight Formal Approach for Analyzing Security of Web Protocols". In: *RAID 2014*. Vol. 8688. LNCS. Springer, 2014, pp. 192–211.

[37] D. Liu, D. Denicola, and Z. Koch. *Payment Method Manifest*. Tech. rep. https://www.w3.org/TR/2017/WD-payment-method-manifest-20171212/. W3C, Dec. 2017.

[38] Z. Luo, X. Cai, J. Pang, and Y. Deng. "Analyzing an Electronic Cash Protocol Using Applied Pi Calculus". In: *ACNS*. Vol. 4521. LNCS. Springer, 2007, pp. 87–103.

[39] Microsoft. *Payment Request API (EdgeHTML)*. 2020. URL: https://docs.microsoft.com/en-us/microsoft-edge/dev-guide/windows-integration/payment-request-api.

[40] Mozilla MDN contributors. *Payment Request API*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Payment_Request_API.

[41] S. J. Murdoch, S. Drimer, R. J. Anderson, and M. Bond. "Chip and PIN is Broken". In: *IEEE S&P*. IEEE Computer Society, 2010, pp. 433–446.

[42] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. "Formal Verification of OAuth 2.0 Using Alloy Framework". In: *CSNT '11*. 2011, pp. 655–659.

[43] R. Pass, L. Seeman, and A. Shelat. "Analysis of the Blockchain Protocol in Asynchronous Networks". In: *EUROCRYPT*. Vol. 10211. LNCS. 2017, pp. 643–673.

[44] M. Roland and J. Langer. "Cloning Credit Cards: A Combined Pre-play and Downgrade Attack on EMV Contactless". In: *WOOT '13*. USENIX Association, 2013.

[45] J. de Ruiter and E. Poll. "Formal Analysis of the EMV Protocol Suite". In: *TOSCA 2011*. Vol. 6993. LNCS. Springer, 2011, pp. 113–129.

[46] A. Ruiz-Martinez. “Towards a web payment framework: State-of-the-art and challenges”. In: *Electron. Commer. Res. Appl.* 14.5 (2015), pp. 345–350.

[47] A. Russell, J. Song, J. Archibald, and M. Kruisselbrink. *Service Workers 1*. Tech. rep. https://www.w3.org/TR/service-workers-1/. W3C, Nov. 2019.

[48] S. Shepard. *Marriott Breach: Unencrypted Passport Numbers, Payment Cards Leaked*. 2019. URL: https://securitytoday.com/articles/2019/01/09/marriott-breach-unencrypted-passport-numbers-payment-cards-leaked.aspx.

[49] Stripe. *Stripe: JavaScript SDK documentation & reference*. URL: https://stripe.com/docs/js/payment_request/create (Retrieved 12/03/2020).

[50] W3C. *Web Payments Working Group*. URL: https://www.w3.org/Payments/WG/.

[51] W3C Web Payments Issue Tracker. *Issue 882: Prevent double spending through retry*. URL: https://github.com/w3c/payment-request/issues/882.

[52] W3C Web Payments Issue Tracker. *Issue 903: Discuss findings of security analysis*. URL: https://github.com/w3c/payment-request/issues/903.

[53] W3C Web Payments Issue Tracker. *Issue 904: Clarification on payment handler selection in spec*. URL: https://github.com/w3c/payment-request/issues/904.

[54] W3C Web Payments Issue Tracker. *Issue 905: Disallow ambiguous methodData declarations?* URL: https://github.com/w3c/payment-request/issues/905.

[55] W3C Web Payments Working Group. *w3c payment-request-info FAQ*. 2018. URL: https://github.com/w3c/payment-request-info/wiki/FAQ#in-what-way-to-the-payment-request-api-increase-security.

[56] J. Wakefield. *EasyJet admits data of nine million hacked*. 2020. URL: https://www.bbc.com/news/technology-52722626.

[57] R. Wang, S. Chen, X. Wang, and S. Qadeer. “How to Shop for Free Online - Security Analysis of Cashier-as-a-Service Based Web Stores”. In: *IEEE S&P*. IEEE Computer Society, 2011, pp. 465–480.

[58] M. Watson. *Web Cryptography API*. Tech. rep. https://www.w3.org/TR/WebCryptoAPI/. W3C, 2017.

[59] Z. Whittaker. *DoorDash confirms data breach affected 4.9 million customers, workers and merchants*. 2019. URL: https://techcrunch.com/2019/09/26/doordash-data-breach/.

[60] D. Winder. *Town Of Salem Hacked Leaving More Than 7.6M With Compromised Data*. 2019. URL: https://www.forbes.com/sites/daveywinder/2019/01/03/town-of-salem-hacked-leaving-more-than-7-6m-with-compromised-data/.

[61] T. Yunusov. “ApplePwn - The future of cardless fraud”. In: *BlackHat USA 2017*. 2017.

APPENDIX A
HIGH-LEVEL PROOF SKETCHES

In the following, we provide high-level proof sketches for both security properties. We refer to Appendix E for the formal proofs.

A. Intended Payments

The Intended Payments property, as defined in Section VI-A, states the following: If an honest payment provider stores a transaction, and if the sender of the transaction is an identity managed by an honest browser, then the browser stores a payment intent (for the transaction) with the same total and receiver.

In our formal model, a transaction is linked to the payment intents of the browser via the transaction identifier $transactionId$. In the full proof of the Intended Payments property, we first show that the transaction identifier stored in a transaction is indeed used by the browser for referencing a list of payment intents. Then, we show that in this list, there is one payment intent having the same total and receiver as the transaction. We now give more details on the proof.

Let t be a transaction stored in the state of an honest payment provider pp (see Definition 68 for details on the structure of a transaction). In the following, we trace back to identify the origin of the transaction identifier $t.\texttt{transactionId}$. An honest payment provider adds a transaction to the `transactions` entry of its state only after receiving an HTTPS (POST) request to its $/\texttt{pay}$ endpoint. This request contains a token which determines the sender of the transaction. More precisely, the payment provider associated the token (when issuing the token) to an identity id which is used as the sender of the transaction, i.e., $t.\texttt{sender} = id$.

Let $b = \texttt{ownerOfID}(\texttt{t.sender})$ be the (honest) browser that manages the identity of the sender of the transaction.

The payment provider only issues a token after successful authentication of an identity. As b and pp are honest, we can apply Lemma 7, and can conclude that a token associated with id does not leak to the attacker, and is only derivable by b and pp.

As an honest payment provider never sends HTTP requests, we can conclude that the token was sent by the browser b. The transaction identifier $t.\texttt{transactionId}$ is taken from the same HTTPS request that contained the token, thus, we conclude that the transaction identifier was sent by the browser b.

In the full proof, we show that the browser b sends a request to the $/\texttt{pay}$ endpoint of pp only due to an honest payment handler (that has the same origin as the payment provider, see also Section V-B).

By tracing back the origins of the transaction identifier, total, and receiver values used by the payment handler, we show that these values are provided to the payment handler by the browser via a `PaymentRequestEvent`. The browser passes a `PaymentRequestEvent` to the payment handler only by calling the DELIVER_TO_DOC helper function in Line 28 of Algorithm 22. The only place where the browser creates such an event and adds it to the pool of events $\mathrm{S}(b).\texttt{events}$ is in Line 22 of Algorithm 22. As this line is executed by b, it will also execute Line 25 of the same algorithm, and by this, add a payment intent to its state using the transaction identifier as a key, i.e., to $\mathrm{S}(b).\texttt{paymentIntents}[t.\texttt{transactionId}]$.

The values for the total and receiver of the transaction, i.e., of $t.\texttt{total}$ and $t.\texttt{receiver}$ that are used for creating the aforementioned `PaymentRequestEvent` (i.e., when executing Line 20 of Algorithm 22) are contained in the same payment intent that is added to $\mathrm{S}(b).\texttt{paymentIntents}[t.\texttt{transactionId}]$.

This is the only place where the browser model modifies $\mathrm{S}(b).\texttt{paymentIntents}[t.\texttt{transactionId}]$, therefore, we can conclude that there is a payment intent with the same total and receiver as the transaction t, by which we show the Intended Payments property.

B. Uniqueness of Payments

For the Uniqueness of Payments property, we need to prove that: (1) there is at most one transaction stored in the state of all honest payment providers of which the transaction identifier is the same with the transaction identifier of the payment intent sequence, and the account of the sender is managed by the browser; and (2) if such a transaction exists, then there is one payment intent in the sequence with such transaction identifier having the same total value and the same receiver with the values in that transaction.

Statement (2) is a trivial corollary of the Intended Payments property. We will now give a proof sketch for (1).

We prove (1) by contradiction: Let b be an honest browser and $txId$ a transaction identifier stored in the state of b. Assume that there exist two different transactions t, t' stored in the state of two honest payment providers pp, pp' (or in one payment provider, i.e., $pp = pp'$) so that both transactions have the same transaction identifier $txId$ and have a sender managed by the same browser b. Similar to the proof in

Section A-A, we show that the transaction identifier of t and t' was sent from the browser b by executing a payment handler with a `PaymentRequestEvent` with the transaction identifier is $txId$. Therefore, there must be two `PaymentRequestEvent` events with the same transaction identifier provided to the payment handler in b.

Similar to the proof presented in Section A-A, we show that the browser must have accepted two payments. The browser accepts a payment by creating a special kind of event, called `SubmitPayment` event, after the merchant starts the payment flow or initiates a retry. In the proof, by tracing back the origins of the transaction identifier in the `PaymentRequestEvent` event and the payment request nonce in the `SubmitPayment` event, we show that both `SubmitPayment` events must have the same payment request nonce.

The `SubmitPayment` event can only be generated within the browser by processing one of two script commands, either `PR_SHOW` (Line 103 of Algorithm 1) or `PRESS_RETRY` (Line 148 Algorithm 21). In the proof, we show that the second `SubmitPayment` event must be created by a retry, i.e., by processing `PRESS_RETRY`. As both `SubmitPayment` events have the same payment request nonce, and the second `SubmitPayment` is a retry, we can conclude that both resulting `PaymentRequestEvents` will be processed by the same payment handler. We highlight that this conclusion is only possible with the fix we proposed in Section III-A.

Having the same payment handler also implies that the requests that will create the transactions at some payment providers are sent to the same payment provider, i.e., $pp \equiv pp'$. Hence, t and t' are stored in one payment provider's state.

As mentioned above, two `PaymentRequestEvents` in b must have the same transaction identifier. From Lemma 17, we conclude that they also have the same payment identifier. The payment handler puts the payment identifier of the `PaymentRequestEvent` in the body of the request sent to the payment provider. Later on, this value is used as the key indexing the corresponding transaction stored in the payment provider's state. Thus, we have that t and t', both stored in the state of pp, are indexed by the same payment identifier. This implies that t and t' are the same transaction, which contradicts the assumption that t and t' are different. Therefore, (1) is proven, completing the proof sketch for Uniqueness of Payments.

$$\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x \tag{1}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x, y), y) = x \tag{2}$$
$$\mathsf{checksig}(\mathsf{sig}(x, y), \mathsf{pub}(y)) = \top \tag{3}$$
$$\mathsf{extractmsg}(\mathsf{sig}(x, y)) = x \tag{4}$$
$$\pi_i(\langle x_1, \ldots, x_n \rangle) = x_i \text{ if } 1 \leq i \leq n \tag{5}$$
$$\pi_j(\langle x_1, \ldots, x_n \rangle) = \Diamond \text{ if } j \notin \{1, \ldots, n\} \tag{6}$$

Figure 6 Equational theory for Σ.

APPENDIX B
THE WEB INFRASTRUCTURE MODEL (WIM)

Here, we provide technical definitions that complete our description of the WIM in Section IV. As mentioned in this section, we follow the descriptions in [20, 21, 22, 23, 24, 25].

A. Terms and Notations

Definition 5 (Nonces and Terms). By $X = \{x_0, x_1, \ldots\}$ we denote a set of variables and by $\mathcal{N}$ we denote an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol in Σ for some $n \geq 0$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by the theory associated with Σ (see Figure 6). For example, we have that $\pi_1(\mathsf{dec_a}(\mathsf{enc_a}(\langle \mathsf{a}, \mathsf{b} \rangle, \mathsf{pub}(k)), k)) \equiv \mathsf{a}$.

Definition 6 (Ground Terms, Messages, Placeholders, Protomessages). By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

We define the set $V_{\text{process}} = \{\nu_1, \nu_2, \ldots\}$ of variables (called placeholders). The set $\mathcal{M}^\nu := \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$ is called the set of *protomessages*, i.e., messages that can contain placeholders.

Example 1. For example, $k \in \mathcal{N}$ and $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc_a}(\langle a, b, c \rangle, \mathsf{pub}(k))$ is interpreted to be the message $\langle a, b, c \rangle$ (the sequence of constants a, b, c) encrypted by the public key $\mathsf{pub}(k)$.

Definition 7 (Events and Protoevents). An *event (over* IPs *and* $\mathcal{M}$*)* is a term of the form $\langle a, f, m \rangle$, for a, $f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events. Events over IPs and $\mathcal{M}^\nu$ are called *protoevents* and are denoted $\mathcal{E}^\nu$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^\nu\langle\rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle\rangle$, $\langle\langle a, f, m \rangle, \langle a', f', m' \rangle, \ldots\rangle$, etc.).

Definition 8 (Normal Form). Let t be a term. The *normal form* of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Figure 6. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 9 (Pattern Matching). Let $pattern \in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches* $pattern$ iff t can be acquired from $pattern$ by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim pattern$. For a sequence of patterns $patterns$ we write $t{\sim}patterns$ to denote that t matches at least one pattern in $patterns$.

For a term t' we write $t' |\, pattern$ to denote the term that is acquired from t' by removing all immediate subterms of t' that do not match $pattern$.

Example 2. For example, for a pattern $p = \langle \top, * \rangle$ we have that $\langle \top, 42 \rangle \sim p$, $\langle \bot, 42 \rangle \not\sim p$, and

$$\langle\langle \bot, \top \rangle, \langle \top, 23 \rangle, \langle \mathsf{a}, \mathsf{b} \rangle, \langle \top, \bot \rangle\rangle |\, p = \langle\langle \top, 23 \rangle, \langle \top, \bot \rangle\rangle \ .$$

Definition 10 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \ldots, x_n\})$, and $t_1, \ldots, t_n \in \mathcal{T}_N$.

By $\tau[t_1/x_1, \ldots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \ldots, n\}$.

Definition 11 (Sequence Notations). For a sequence $t = \langle t_1, \ldots, t_n \rangle$ and a set s we use $t \subset^{\langle\rangle} s$ to say that $t_1, \ldots, t_n \in s$. We define $x \in^{\langle\rangle} t \iff \exists i : t_i = x$. For a term y we write $t +^{\langle\rangle} y$ to denote the sequence $\langle t_1, \ldots, t_n, y \rangle$. For a sequence $r = \langle r_1, \ldots, r_m \rangle$ we write $t \cup r$ to denote the sequence $\langle t_1, \ldots, t_n, r_1, \ldots, r_m \rangle$. For a finite set M with $M = \{m_1, \ldots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \ldots, m_n \rangle$. (The order of the elements does not matter; one is chosen arbitrarily.)

Definition 12. A *dictionary over X and Y* is a term of the form

$$\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$$

where $k_1, \ldots, k_n \in X$, $v_1, \ldots, v_n \in Y$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \ldots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1 : v_1, \ldots, k_i : v_i, \ldots, k_n : v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$.

We note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$. Figure 7 shows the short notation for dictionary operations. For a dictionary $z = [k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n]$ we write $k \in z$ to say that there exists i such that $k = k_i$. We write $z[k_j]$ to refer to the value v_j. (Note that if a dictionary contains two elements $\langle k, v \rangle$ and $\langle k, v' \rangle$, then the notations and operations for dictionaries apply non-deterministically to one of both elements.) If $k \notin z$, we set $z[k] := \langle\rangle$.

$$[k_1 : v_1, \ldots, k_i : v_i, \ldots, k_n : v_n]\,[k_i] = v_i \tag{7}$$

$$[k_1 : v_1, \ldots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1} \ldots, k_n : v_n] - k_i = [k_1 : v_1, \ldots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1} \ldots, k_n : v_n] \tag{8}$$

Figure 7 Dictionary operators with $1 \leq i \leq n$.

Given a term $t = \langle t_1, \ldots, t_n \rangle$, we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 13. A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

Example 3. For the term $\tau = \langle a, b, \langle c, d, \langle e, f \rangle \rangle \rangle$ and the pointer $\overline{p} = \langle 3, 1 \rangle$, the subterm of τ at the position $\overline{p}$ is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3, 1 \rangle = \tau.3.\overline{p} = \tau.3.3.1 = e$.

To improve readability, we try to avoid writing, e.g., $o.2$ or $\pi_2(o)$ in this document. Instead, we will use the names of the components of a sequence that is of a defined form as pointers that point to the corresponding subterms. E.g., if an *Origin* term is defined as $\langle host, protocol \rangle$ and o is an Origin term, then we can write $o.\texttt{protocol}$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

Definition 14 (Concatenation of terms and sequences). For a term $a = \langle a_1, \ldots, a_i \rangle$ and a sequence $b = (b_1, b_2, \ldots)$, we define the *concatenation* as $a \cdot b := (a_1, \ldots, a_i, b_1, b_2, \ldots)$.

Definition 15 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a non-deterministically chosen occurence of that element in X is removed.

B. Message and Data Formats

We now provide some more details about data and message formats that are needed for the formal treatment of the web model and the analysis presented in the following.

1) URLs:

Definition 16. A *URL* is a term of the form

$$\langle \texttt{URL}, protocol, host, path, parameters, fragment \rangle$$

with $protocol \in \{\texttt{P}, \texttt{S}\}$ (for **p**lain (HTTP) and **s**ecure (HTTPS)), a domain $host \in \mathsf{Doms}$, $path \in \mathbb{S}$, $parameters \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, and $fragment \in \mathcal{T}_{\mathcal{N}}$. The set of all valid URLs is URLs.

The *fragment* part of a URL can be omitted when writing the URL. Its value is then defined to be $\bot$. We sometimes also write $\mathsf{URL}_{path}^{host}$ to denote the URL $\langle \texttt{URL}, \texttt{S}, host, path, \langle\rangle, \bot \rangle$.

As mentioned above, for specific terms, such as URLs, we typically use the names of its components as pointers (see Definition 13):

Example 4. For the URL $u = \langle \texttt{URL}, a, b, c, d \rangle$, $u.\texttt{protocol} = a$. If, in the algorithm described later, we say $u.\texttt{path} := e$ then $u = \langle \texttt{URL}, a, b, c, e \rangle$ afterwards.

2) Origins:

Definition 17. An *origin* is a term of the form $\langle host, protocol\rangle$ with $host \in \mathsf{Doms}$ and $protocol \in \{\mathtt{P}, \mathtt{S}\}$. We write Origins for the set of all origins.

Example 5. For example, $\langle \mathtt{FOO}, \mathtt{S}\rangle$ is the HTTPS origin for the domain FOO, while $\langle \mathtt{BAR}, \mathtt{P}\rangle$ is the HTTP origin for the domain BAR.

3) Cookies:

Definition 18. A *cookie* is a term of the form $\langle name, content\rangle$ where $name \in \mathcal{T}_{\mathcal{N}}$, and $content$ is a term of the form $\langle value, secure, session, httpOnly\rangle$ where $value \in \mathcal{T}_{\mathcal{N}}$, $secure$, $session$, $httpOnly \in \{\top, \bot\}$. We write Cookies for the set of all cookies and $\mathsf{Cookies}^{\nu}$ for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

If the $secure$ attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections. If the $session$ flag is set, this cookie will be deleted as soon as the browser is closed. The $httpOnly$ attribute controls whether JavaScript has access to this cookie.

Note that cookies of the form described here are only contained in HTTP(S) requests. In HTTP(S) responses, only the components $name$ and $value$ are transferred as a pairing of the form $\langle name, value\rangle$.

4) HTTP Messages:

Definition 19. An *HTTP request* is a term of the form shown in (9). An *HTTP response* is a term of the form shown in (10).

$$\langle \mathtt{HTTPReq}, nonce, method, host, path, parameters, headers, body\rangle \tag{9}$$

$$\langle \mathtt{HTTPResp}, nonce, status, headers, body\rangle \tag{10}$$

The components are defined as follows:

- $nonce \in \mathcal{N}$ serves to map each response to the corresponding request
- $method \in \mathsf{Methods}$ is one of the HTTP methods.
- $host \in \mathsf{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $path \in \mathbb{S}$ is a string indicating the requested resource at the server side
- $status \in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard)
- $parameters \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ contains URL parameters
- $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, containing request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \mathtt{Origin}, o\rangle$ where o is an origin,
  - $\langle \mathtt{Set\text{-}Cookie}, c\rangle$ where c is a sequence of cookies,
  - $\langle \mathtt{Cookie}, c\rangle$ where $c \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$ (note that in this header, only names and values of cookies are transferred),
  - $\langle \mathtt{Location}, l\rangle$ where $l \in \mathsf{URLs}$,
  - $\langle \mathtt{Referer}, r\rangle$ where $r \in \mathsf{URLs}$,
  - $\langle \mathtt{Strict\text{-}Transport\text{-}Security}, \top\rangle$,
  - $\langle \mathtt{Authorization}, \langle username, password\rangle\rangle$ where $username$, $password \in \mathbb{S}$,
  - $\langle \mathtt{ReferrerPolicy}, p\rangle$ where $p \in \{\mathtt{noreferrer}, \mathtt{origin}\}$.
- $body \in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

We write HTTPRequests/HTTPResponses for the set of all HTTP requests or responses, respectively.

Example 6 (HTTP Request and Response).

$$\begin{aligned} r :=& \langle \mathtt{HTTPReq}, n_1, \mathtt{POST}, \mathtt{example.com}, \mathtt{/show}, \langle\langle \mathtt{index}, 1\rangle\rangle, \\ & [\mathtt{Origin} : \langle \mathtt{example.com}, \mathtt{S}\rangle], \langle \mathtt{foo}, \mathtt{bar}\rangle\rangle \end{aligned} \tag{11}$$

$$s := \langle \mathtt{HTTPResp}, n_1, 200, \langle\langle \mathtt{Set\text{-}Cookie}, \langle\langle \mathtt{SID}, \langle n_2, \bot, \bot, \top\rangle\rangle\rangle\rangle\rangle, \langle \mathtt{somescript}, x\rangle\rangle \tag{12}$$

An HTTP POST request for the URL http://example.com/show?index=1 is shown in (11), with an Origin header and a body that contains $\langle \mathtt{foo}, \mathtt{bar}\rangle$. A possible response is shown in (12), which contains an httpOnly cookie with name SID and value n_2 as well as the string representation somescript of the script $\mathsf{script}^{-1}(\mathtt{somescript})$ (which should be an element of $\mathcal{S}$) and its initial state x.

a) Encrypted HTTP Messages: For HTTPS, requests are encrypted using the public key of the server. Such a request contains an (ephemeral) symmetric key chosen by the client that issued the request. The server is supported to encrypt the response using the symmetric key.

Definition 20. An *encrypted HTTP request* is of the form $\mathsf{enc_a}(\langle m, k'\rangle, k)$, where $k \in terms$, $k' \in \mathcal{N}$, and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc_s}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

We say that an HTTP(S) response matches or corresponds to an HTTP(S) request if both terms contain the same nonce.

Example 7.

$$\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{13}$$

$$\mathsf{enc_s}(s, k') \tag{14}$$

The term (13) shows an encrypted request (with r as in (11)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (14) is a response (with s as in (12)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (13).

5) DNS Messages:

Definition 21. A *DNS request* is a term of the form $\langle \texttt{DNSResolve}, domain, nonce\rangle$ where $domain \in \mathsf{Doms}$, $nonce \in \mathcal{N}$. We call the set of all DNS requests $\mathsf{DNSRequests}$.

Definition 22. A *DNS response* is a term of the form $\langle \texttt{DNSResolved}, domain, result, nonce\rangle$ with $domain \in \mathsf{Doms}$, $result \in \mathsf{IPs}$, $nonce \in \mathcal{N}$. We call the set of all DNS responses $\mathsf{DNSResponses}$.

DNS servers are supposed to include the nonce they received in a DNS request in the DNS response that they send back so that the party which issued the request can match it with the request.

C. Atomic Processes, Systems and Runs

Entities that take part in a network are modeled as atomic processes. An atomic process takes a term that describes its current state and an event as input, and then (non-deterministically) outputs a new state and a set of events.

Definition 23 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

$$p = (I^p, Z^p, R^p, s_0^p)$$

where $I^p \subseteq \mathsf{IPs}$, $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{T}^\nu\langle\rangle} \times \mathcal{T}_{\mathcal{N}}(V_{\text{process}}))$ (input event and old state map to sequence of output events and new state), and $s_0^p \in Z^p$ is the initial state of p. For any new state s and any sequence of nonces $(\eta_1, \eta_2, \dots)$ we demand that $s[\eta_1/\nu_1, \eta_2/\nu_2, \dots] \in Z^p$. A *system* $\mathcal{P}$ is a (possibly infinite) set of atomic processes.

Definition 24 (Configurations). A *configuration of a system* $\mathcal{P}$ is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence[11] $(e_1, e_2, \dots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \dots)$.

Definition 25 (Processing Steps). A *processing step of the system* $\mathcal{P}$ is of the form

$$(S, E, N) \xrightarrow[p \to E_{\text{out}}]{e_{\text{in}} \to p} (S', E', N')$$

where

1) (S, E, N) and (S', E', N') are configurations of $\mathcal{P}$,
2) $e_{\text{in}} = \langle a, f, m\rangle \in E$ is an event,
3) $p \in \mathcal{P}$ is a process,
4) E_{out} is a sequence (term) of events

such that there exists

1) a sequence (term) $E_{\text{out}}^\nu \subseteq 2^{\mathcal{T}^\nu\langle\rangle}$ of protoevents,
2) a term $s^\nu \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
3) a sequence $(v_1, v_2, \dots, v_i)$ of all placeholders appearing in E_{out}^ν (ordered lexicographically),
4) a sequence $N^\nu = (\eta_1, \eta_2, \dots, \eta_i)$ of the first i elements in N

with

[11] Here: Not in the sense of terms as defined earlier.

1) $((e_{\text{in}}, S(p)), (E^{\nu}_{\text{out}}, s^{\nu})) \in R^p$ and $a \in I^p$,
2) $E_{\text{out}} = E^{\nu}_{\text{out}}[m_1/v_1, \ldots, m_i/v_i]$
3) $S'(p) = s^{\nu}[m_1/v_1, \ldots, m_i/v_i]$ and $S'(p') = S(p')$ for all $p' \neq p$
4) $E' = E_{\text{out}} \cdot (E \setminus \{e_{\text{in}}\})$
5) $N' = N \setminus N^{\nu}$

We may omit the superscript and/or subscript of the arrow.

Intuitively, for a processing step, we select one of the processes in $\mathcal{P}$, and call it with one of the events in the list of waiting events E. In its output (new state and output events), we replace any occurences of placeholders ν_x by "fresh" nonces from N (which we then remove from N). The output events are then prepended to the list of waiting events, and the state of the process is reflected in the new configuration.

Definition 26 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0* is a finite sequence of configurations $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s^p_0$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

We denote the state $S^n(p)$ of a process p at the end of a run ρ by $\rho(p)$.

Usually, we will initiate runs with a set E^0 containing infinite trigger events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for each $a \in \mathsf{IPs}$, interleaved by address.

D. Atomic Dolev-Yao Processes

We next define atomic Dolev-Yao processes, for which we require that the messages and states that they output can be computed (more formally, derived) from the current input event and state. For this purpose, we first define what it means to derive a message from given messages.

Definition 27 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with placeholders V* if there exist $n \geq 0$, $m_1, \ldots, m_n \in M$, and $\tau \in \mathcal{T}_{\emptyset}(\{x_1, \ldots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.

For example, the term a can be derived from the set of terms $\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\}$, i.e., $a \in d_{\{\}}(\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\})$.

A *(Dolev-Yao) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

Definition 28 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s^p_0)$ such that (I^p, Z^p, R^p, s^p_0) is an atomic process and for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, with $((e, s), (E, s')) \in R^p$ it holds true that E, $s' \in d_{V_{\text{process}}}(\{e, s\})$.

E. Attackers

The so-called *attacker process* is a Dolev-Yao process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any Dolev-Yao process could possibly perform. Attackers can corrupt other parties (using corrupt messages).

Definition 29 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses* $A \subseteq \mathsf{IPs}$ is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e, and $s \in \mathcal{T}_{\mathcal{N}}$ we have that $((e, s), (E, s')) \in R$ iff $s' = \langle e, E, s\rangle$ and $E = \langle\langle a_1, f_1, m_1\rangle, \ldots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_0, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e, s\})$.

Note that in a web system, we distinguish between two kinds of attacker processes: web attackers and network attackers. Both kinds match the definition above, but differ in the set of assigned addresses in the context of a web system. While for web attackers, the set of addresses I^p is disjoint from other web attackers and honest processes, i.e., web attackers participate in the network as any other party, the set of addresses I^p of a network attacker is not restricted. Hence, a network attacker can intercept events addressed to any party as well as spoof all addresses. Note that one network attacker subsumes any number of web attackers as well as any number of network attackers.

F. Browsers

Following the informal description of the browser model in Section IV, we now present the formal model of browsers.

1) Scripts: Recall that a *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below).

Definition 30 (Placeholders for Scripts). By $V_{\text{script}} = \{\lambda_1, \dots\}$ we denote an infinite set of variables used in scripts.

Definition 31 (Scripts). A *script* is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ with $(s, s') \in R$ it follows that $s' \in d_{V_{\text{script}}}(s)$.

A script is called by the browser which provides it with state information (such as the script's last scriptstate and limited information about the browser's state) s. The script then outputs a term s', which represents the new scriptstate and some command which is interpreted by the browser. The term s' may contain variables $\lambda_1, \dots$ which the browser will replace by (otherwise unused) placeholders $\nu_1, \dots$ which will be replaced by nonces once the browser DY process finishes (effectively providing the script with a way to get "fresh" nonces).

Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

attacker script R^{att}:

Definition 32 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s, s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$.

2) Web Browser State: Before we can define the state of a web browser, we first have to define windows and documents.

Definition 33. A *window* is a term of the form $w = \langle nonce, documents, opener \rangle$ with $nonce \in \mathcal{N}$, $documents \subset^{\langle\rangle}$ Documents (defined below), $opener \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle} documents$ if $documents$ is not empty (we then call d the *active document of* w). We write Windows for the set of all windows. We write $w.\texttt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else.

We will refer to the window nonce as *(window) reference*.

The documents contained in a window term to the left of the active document are the previously viewed documents (available to the user via the "back" button) and the documents in the window term to the right of the currently active document are documents available via the "forward" button.

A window a may have opened a top-level window b (i.e., a window term which is not a subterm of a document term). In this case, the *opener* part of the term b is the nonce of a, i.e., $b.\texttt{opener} = a.\texttt{nonce}$.

Definition 34. A *document* d is a term of the form

$$\langle nonce, location, headers, referrer, script, scriptstate, scriptinputs, subwindows, active \rangle$$

where $nonce \in \mathcal{N}$, $location \in$ URLs, $headers \in \left[\mathbb{S} \times \mathcal{T}_{\mathcal{N}}\right]$, $referrer \in \text{URLs} \cup \{\bot\}$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $subwindows \subset^{\langle\rangle}$ Windows, $active \in \{\top, \bot\}$. A *limited document* is a term of the form $\langle nonce, subwindows \rangle$ with $nonce$, $subwindows$ as above. A window $w \in^{\langle\rangle} subwindows$ is called a *subwindow* (of d). We write Documents for the set of all documents. For a document term d we write $d.\texttt{origin}$ to denote the origin of the document, i.e., the term $\langle d.\texttt{location.host}, d.\texttt{location.protocol} \rangle \in$ Origins.

We will refer to the document nonce as *(document) reference*.

Definition 35. For two window terms w and w' we write

$$w \xrightarrow{\text{childof}} w'$$

if $w \in^{\langle\rangle} w'.\texttt{activedocument.subwindows}$. We write $\xrightarrow{\text{childof}^+}$ for the transitive closure and we write $\xrightarrow{\text{childof}^*}$ for the reflexive transitive closure.

In the web browser state, HTTP(S) messages are tracked using *references*. These are defined as a pairing of a an identifier for the type of the request (XHR for XMLHTTPRequests, or REQ for normal HTTP(S) requests) and a nonce (which never leaves the browser unless the browser becomes corrupted).

We can now define the set of states of web browsers. Note that we use the dictionary notation that we introduced in Definition 12.

Definition 36. The *set of states $Z_{webbrowser}$ of a web browser atomic Dolev-Yao process* consists of the terms of the form

$$\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping, \\ sts, DNSaddress, pendingDNS, pendingRequests, isCorrupted\rangle$$

with the subterms as follows:

- $windows \subset^{\langle\rangle} \mathsf{Windows}$ contains a list of window terms (modeling top-level windows, or browser tabs) which contain documents, which in turn can contain further window terms (iframes).
- $ids \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ is a list of identities that are owned by this browser (i.e., belong to the user of the browser).
- $secrets \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ contains a list of secrets that are associated with certain origins (i.e., passwords of the user of the browser at certain websites). Note that this structure allows to have a single secret under an origin or a list of secrets under an origin.
- $cookies$ is a dictionary over Doms and sequences of Cookies modeling cookies that are stored for specific domains.
- $localStorage \in \left[\mathsf{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$ stores the data saved by scripts using the localStorage API (separated by origins).
- $sessionStorage \in \left[OR \times \mathcal{T}_{\mathcal{N}}\right]$ for $OR := \{\langle o, r\rangle | \, o \in \mathsf{Origins}, r \in \mathcal{N}\}$ similar to localStorage, but the data in sessionStorage is additionally separated by top-level windows.
- $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ maps domains to TLS encryption keys.
- $sts \subset^{\langle\rangle} \mathsf{Doms}$ stores the list of domains that the browser only accesses via TLS (strict transport security).
- $DNSaddress \in \mathsf{IPs}$ defines the IP address of the DNS server.
- $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ contains one pairing per unanswered DNS query of the form $\langle reference, request, url\rangle$. In these pairings, $reference$ is an HTTP(S) request reference (as above), $request$ contains the HTTP(S) message that awaits DNS resolution, and url contains the URL of said HTTP request. The pairings in $pendingDNS$ are indexed by the DNS request/response nonce.
- $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ contains pairings of the form $\langle reference, request, url, key, f\rangle$ with $reference$, $request$, and url as in $pendingDNS$, key is the symmetric encryption key if HTTPS is used or $\perp$ otherwise, and f is the IP address of the server to which the request was sent.
- $isCorrupted \in \{\perp, \texttt{FULLCORRUPT}, \texttt{CLOSECORRUPT}\}$ specifies the corruption level of the browser.

In corrupted browsers, certain subterms are used in different ways (e.g., $pendingRequests$ is used to store all observed messages).

3) Web Browser Relation: We will now define the relation $R_{\text{webbrowser}}$ of a standard HTTP browser. We first introduce some notations and then describe the functions that are used for defining the browser main algorithm. We then define the browser relation.

a) Helper Functions: In the following description of the web browser relation $R_{\text{webbrowser}}$ we use the helper functions Subwindows, Docs, Clean, CookieMerge and AddCookie.

Subwindows. Given a browser state s, $\mathsf{Subwindows}(s)$ denotes the set of all pointers[12] to windows in the window list $s.\texttt{windows}$, their active documents, and (recursively) the subwindows of these documents. We exclude subwindows of inactive documents and their subwindows. With $\mathsf{Docs}(s)$ we denote the set of pointers to all active documents in the set of windows referenced by $\mathsf{Subwindows}(s)$.

Definition 37. For a browser state s we denote by $\mathsf{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\texttt{windows}$ there is a $\overline{p} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p} = w$. (2) For all $\overline{p} \in \mathsf{Subwindows}(s)$, the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p'} = w$.

Given a browser state s, the set $\mathsf{Docs}(s)$ of pointers to active documents is the minimal set such that for every $\overline{p} \in \mathsf{Subwindows}(s)$, there exists a pointer $\overline{p'} \in \mathsf{Docs}(s)$ with $s.\overline{p'} = s.\overline{p}.\texttt{activedocument}$.

By $\mathsf{Subwindows}^{+}(s)$ and $\mathsf{Docs}^{+}(s)$ we denote the respective sets that also include the inactive documents and their subwindows.

Clean. The function Clean will be used to determine which information about windows and documents the script running in the document d has access to.

Definition 38. Let s be a browser state and d a document. By $\mathsf{Clean}(s, d)$ we denote the term that equals $s.\texttt{windows}$ but with (1) all inactive documents removed (including their subwindows etc.), (2) all subterms that represent non-same-origin documents w.r.t. d replaced by a limited document d' with the

[12]Recall the definition of a pointer in Definition 13.

same nonce and the same subwindow list, and (3) the values of the subterms `headers` for all documents set to $\langle\rangle$. (Note that non-same-origin documents on all levels are replaced by their corresponding limited document.)

CookieMerge. The function CookieMerge merges two sequences of cookies together: When used in the browser, $oldcookies$ is the sequence of existing cookies for some origin, $newcookies$ is a sequence of new cookies that was output by some script. The sequences are merged into a set of cookies using an algorithm that is based on the *Storage Mechanism* algorithm described in RFC6265.

Definition 39. For a sequence of cookies (with pairwise different names) $oldcookies$ and a sequence of cookies $newcookies$, the set $\mathsf{CookieMerge}(oldcookies, newcookies)$ is defined by the following algorithm: From $newcookies$ remove all cookies c that have $c.\texttt{content.httpOnly} \equiv \top$. For any c, $c' \in^{\langle\rangle} newcookies$, $c.\texttt{name} \equiv c'.\texttt{name}$, remove the cookie that appears left of the other in $newcookies$. Let m be the set of cookies that have a name that either appears in $oldcookies$ or in $newcookies$, but not in both. For all pairs of cookies $(c_{\text{old}}, c_{\text{new}})$ with $c_{\text{old}} \in^{\langle\rangle} oldcookies$, $c_{\text{new}} \in^{\langle\rangle} newcookies$, $c_{\text{old}}.\texttt{name} \equiv c_{\text{new}}.\texttt{name}$, add c_{new} to m if $c_{\text{old}}.\texttt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(oldcookies, newcookies)$ is m.

AddCookie. The function AddCookie adds a cookie c received in an HTTP response to the sequence of cookies contained in the sequence $oldcookies$. It is again based on the algorithm described in RFC6265 but simplified for the use in the browser model.

Definition 40. For a sequence of cookies (with pairwise different names) $oldcookies$ and a cookie c, the sequence $\mathsf{AddCookie}(oldcookies, c)$ is defined by the following algorithm: Let $m := oldcookies$. Remove any c' from m that has $c.\texttt{name} \equiv c'.\texttt{name}$. Append c to m and return m.

NavigableWindows. The function NavigableWindows returns a set of windows that a document is allowed to navigate. We closely follow [7], Section 5.1.4 for this definition.

Definition 41. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

- If $s'.\overline{w}.\texttt{activedocument.origin} \equiv s'.\overline{w}.\texttt{activedocument.origin}$ (i.e., the active documents in $\overline{w}$ and $\overline{w'}$ are same-origin), then $\overline{w'} \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w'} \wedge \nexists\, \overline{w''} \in \mathsf{Subwindows}(s')$ with $s'.\overline{w'} \xrightarrow{\text{childof}^*} s'.\overline{w''}$ ($\overline{w'}$ is a top-level window and $\overline{w}$ is an ancestor window of $\overline{w'}$), then $\overline{w'} \in \overline{W}$, and
- If $\exists\, \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'} \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\ s'.\overline{p}.\texttt{activedocument.origin} = s'.\overline{w}.\texttt{activedocument.origin}$ ($\overline{w'}$ is not a top-level window but there is an ancestor window $\overline{p}$ of $\overline{w'}$ with an active document that has the same origin as the active document in $\overline{w}$), then $\overline{w'} \in \overline{W}$, and
- If $\exists\, \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'}.\texttt{opener} = s'.\overline{p}.\texttt{nonce} \wedge \overline{p} \in \overline{W}$ ($\overline{w'}$ is a top-level window—it has an opener—and $\overline{w}$ is allowed to navigate the opener window of $\overline{w'}$, $\overline{p}$), then $\overline{w'} \in \overline{W}$.

b) Notations for Functions and Algorithms: We use the following notations to describe the browser algorithms:

Non-deterministic chosing and iteration. The notation **let** $n \leftarrow N$ is used to describe that n is chosen non-deterministically from the set N. We write **for each** $s \in M$ **do** to denote that the following commands (until **end for**) are repeated for every element in M, where the variable s is the current element. The order in which the elements are processed is chosen non-deterministically. We write, for example,

let x, y **such that** $\langle \texttt{Constant}, x, y\rangle \equiv t$ **if possible; otherwise** doSomethingElse

for some variables x, y, a string `Constant`, and some term t to express that $x := \pi_2(t)$, and $y := \pi_3(t)$ if $\texttt{Constant} \equiv \pi_1(t)$ and if $|\langle \texttt{Constant}, x, y\rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

Stop without output. We write **stop** (without further parameters) to denote that there is no output and no change in the state.

Placeholders. In several places throughout the algorithms presented next we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 5). Table I shows a list of all placeholders used.

c) Functions: In the description of the following functions, we use a, f, m, and s as read-only global input variables. All other variables are local variables or arguments.

- The function GETNAVIGABLEWINDOW (Algorithm 2) is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference

Algorithm 2 Web Browser Model: Determine window for navigation.

function GETNAVIGABLEWINDOW($\overline{w}$, *window*, *noreferrer*, s')
  if *window* $\equiv$ `_BLANK` **then** → Open a new window when `_BLANK` is used
    if *noreferrer* $\equiv \top$ **then**
      let $w' := \langle \nu_9, \langle\rangle, \bot \rangle$
    else
      let $w' := \langle \nu_9, \langle\rangle, s'.\overline{w}.\texttt{nonce} \rangle$
    let $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$ ↪ **and let** $\overline{w}'$ be a pointer to this new element in s'
    return $\overline{w}'$
  let $\overline{w}' \leftarrow$ NavigableWindows($\overline{w}, s'$) **such that** $s'.\overline{w}'.\texttt{nonce} \equiv$ *window* ↪ **if possible; otherwise return** $\overline{w}$
  return $\overline{w}'$

Algorithm 3 Web Browser Model: Determine same-origin window.

function GETWINDOW($\overline{w}$, *window*, s')
  let $\overline{w}' \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}'.\texttt{nonce} \equiv$ *window* ↪ **if possible; otherwise return** $\overline{w}$
  if $s'.\overline{w}'.\texttt{activedocument.origin} \equiv s'.\overline{w}.\texttt{activedocument.origin}$ **then**
    return $\overline{w}'$
  return $\overline{w}$

Algorithm 4 Web Browser Model: Cancel pending requests for given window.

function CANCELNAV(*reference*, s')
  remove all $\langle$*reference*, *req*, *key*, $f\rangle$ **from** $s'.\texttt{pendingRequests}$ **for any** *req*, *key*, f
  remove all $\langle x, \langle$*reference*, *message*, *url*$\rangle\rangle$ **from** $s'.\texttt{pendingDNS}$ ↪ **for any** x, *message*, *url*
  return s'

Algorithm 5 Web Browser Model: Prepare headers, do DNS resolution, save message.

function HTTP_SEND(*reference*, *message*, *url*, *origin*, *referrer*, *referrerPolicy*, s')
  if *message*.`host` $\in^{\langle\rangle} s'.\texttt{sts}$ **then**
    let *url*.`protocol` := `S`
  let *cookies* := $\langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}[\textit{message}.\texttt{host}] \wedge (c.\texttt{content.secure} \implies (\textit{url}.\texttt{protocol} = \texttt{S}))\}\rangle$
  let *message*.`headers`[`Cookie`] := *cookies*
  if *origin* $\not\equiv \bot$ **then**
    let *message*.`headers`[`Origin`] := *origin*
  if *referrerPolicy* $\equiv$ `no-referrer` **then**
    let *referrer* := $\bot$
  if *referrer* $\not\equiv \bot$ **then**
    if *referrerPolicy* $\equiv$ `origin` **then**
      let *referrer* := $\langle$`URL`, *referrer*.`protocol`, *referrer*.`host`, /, $\langle\rangle, \bot\rangle$ → Referrer stripped down to origin.
    let *referrer*.`fragment` := $\bot$ → Browsers do not send fragment identifiers in the Referer header.
    let *message*.`headers`[`Referer`] := *referrer*
  let $s'.\texttt{pendingDNS}[\nu_8] := \langle$*reference*, *message*, *url*$\rangle$
  stop $\langle\langle s'.\texttt{DNSaddress}, a, \langle\texttt{DNSResolve}, \textit{message}.\texttt{host}, \nu_8\rangle\rangle\rangle$, s'

Algorithm 6 Web Browser Model: Navigate a window backward.

function NAVBACK($\overline{w'}$, s')
  if $\exists \bar{j} \in \mathbb{N}, \bar{j} > 1$ **such that** $s'.\overline{w'}.\texttt{documents}.\bar{j}.\texttt{active} \equiv \top$ **then**
    let $s'.\overline{w'}.\texttt{documents}.\bar{j}.\texttt{active} := \bot$
    let $s'.\overline{w'}.\texttt{documents}.(\bar{j}-1).\texttt{active} := \top$
    let $s' :=$ CANCELNAV($s'.\overline{w}'.\texttt{nonce}, s'$)

Algorithm 7 Web Browser Model: Navigate a window forward.

function NAVFORWARD($\overline{w'}$, s')
  if $\exists \bar{j} \in \mathbb{N}$ **such that** $s'.\overline{w'}.\texttt{documents}.\bar{j}.\texttt{active} \equiv \top$
      $\hookrightarrow \ \wedge\ s'.\overline{w'}.\texttt{documents}.(\bar{j}+1) \in \mathsf{Documents}$ **then**
    let $s'.\overline{w'}.\texttt{documents}.\bar{j}.\texttt{active} := \bot$
    let $s'.\overline{w'}.\texttt{documents}.(\bar{j}+1).\texttt{active} := \top$
    let $s' := \mathsf{CANCELNAV}(s'.\overline{w}'.\texttt{nonce}, s')$

Algorithm 8 Web Browser Model: Execute a script.

function RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
  let $tree := \mathsf{Clean}(s', s'.\overline{d})$
  let $cookies := \langle \{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
    $\hookrightarrow \ \wedge c.\texttt{content.httpOnly} = \bot$
    $\hookrightarrow \ \wedge \left(c.\texttt{content.secure} \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S}\right)\right)\}\rangle$
  let $tlw \leftarrow s'.\texttt{windows}$ **such that** tlw is the top-level window containing $\overline{d}$
  let $sessionStorage := s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$
  let $localStorage := s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $secrets := s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $R \leftarrow \mathsf{script}^{-1}(s'.\overline{d}.\texttt{script})$
  let $in := \langle tree, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}, s'.\overline{d}.\texttt{scriptinputs}, cookies,$
    $\hookrightarrow \ localStorage, sessionStorage, s'.\texttt{ids}, secrets\rangle$
  let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $cookies' \leftarrow \mathsf{Cookies}^{\nu}$, $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
    $\hookrightarrow \ command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $out := \langle state', cookies', localStorage', sessionStorage', command\rangle$
    $\hookrightarrow$ **such that** $out := out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \dots]$ **with** $(in, out^{\lambda}) \in R$
  let $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right] :=$
    $\hookrightarrow \langle \mathsf{CookieMerge}(s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right], cookies')\rangle$
  let $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right] := localStorage'$
  let $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right] := sessionStorage'$
  let $s'.\overline{d}.\texttt{scriptstate} := state'$
  let $referrer := s'.\overline{d}.\texttt{location}$
  let $referrerPolicy := s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
  let $reference := \langle \texttt{REQ}, s'.\overline{w}'.\texttt{nonce}\rangle$
  let $docorigin := s'.\overline{d}.\texttt{origin}$
  switch $command$ **do**
    case $\langle \texttt{HREF}, url, hrefwindow, noreferrer\rangle$
      let $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, hrefwindow, noreferrer, s')$
      let $req := \langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
      if $noreferrer \equiv \top$ **then**
        let $referrerPolicy := \texttt{noreferrer}$
      let $s' := \mathsf{CANCELNAV}(reference, s')$
      call $\mathsf{HTTP\_SEND}(reference, req, url, \bot, referrer, referrerPolicy, s')$
    case $\langle \texttt{IFRAME}, url, window\rangle$
      if $window \equiv \texttt{\_SELF}$ **then**
        let $\overline{w}' := \overline{w}$
      else
        let $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
      let $req := \langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
      let $w' := \langle \nu_5, \langle\rangle, \bot\rangle$
      let $s'.\overline{w}'.\texttt{activedocument.subwindows} := s'.\overline{w}'.\texttt{activedocument.subwindows} +^{\langle\rangle} w'$
      call $\mathsf{HTTP\_SEND}(\nu_5, req, url, \bot, referrer, referrerPolicy, s')$

36: **case** $\langle$`FORM`, *url*, *method*, *data*, *hrefwindow*$\rangle$
37: **if** *method* $\notin$ {`GET`, `POST`} **then** [13]
38: **stop**
39: **let** $\overline{w}'$:= GETNAVIGABLEWINDOW($\overline{w}$, *hrefwindow*, $\bot$, s')
40: **if** *method* = `GET` **then**
41: **let** *body* := $\langle\rangle$
42: **let** *parameters* := *data*
43: **let** *origin* := $\bot$
44: **else**
45: **let** *body* := *data*
46: **let** *parameters* := *url*.`parameters`
47: **let** *origin* := *docorigin*
48: **let** *req* := $\langle$`HTTPReq`, ν_4, *method*, *url*.`host`, *url*.`path`, *parameters*, $\langle\rangle$, *body*$\rangle$
49: **let** s' := CANCELNAV(*reference*, s')
50: **call** HTTP_SEND(*reference*, *req*, *url*, *origin*, *referrer*, *referrerPolicy*, s')
51: **case** $\langle$`SETSCRIPT`, *window*, *script*$\rangle$
52: **let** $\overline{w}'$:= GETWINDOW($\overline{w}$, *window*, s')
53: **let** $s'.\overline{w}'$.`activedocument.script` := *script*
54: **stop** $\langle\rangle$, s'
55: **case** $\langle$`SETSCRIPTSTATE`, *window*, *scriptstate*$\rangle$
56: **let** $\overline{w}'$:= GETWINDOW($\overline{w}$, *window*, s')
57: **let** $s'.\overline{w}'$.`activedocument.scriptstate` := *scriptstate*
58: **stop** $\langle\rangle$, s'
59: **case** $\langle$`XMLHTTPREQUEST`, *url*, *method*, *data*, *xhrreference*$\rangle$
60: **if** *method* $\in$ {`CONNECT`, `TRACE`, `TRACK`} $\wedge$ *xhrreference* $\notin$ {$\mathcal{N}$, $\bot$} **then**
61: **stop**
62: **if** *url*.`host` $\not\equiv$ *docorigin*.`host` $\vee$ *url* $\not\equiv$ *docorigin*.`protocol` **then**
63: **stop**
64: **if** *method* $\in$ {`GET`, `HEAD`} **then**
65: **let** *data* := $\langle\rangle$
66: **let** *origin* := $\bot$
67: **else**
68: **let** *origin* := *docorigin*
69: **let** *req* := $\langle$`HTTPReq`, ν_4, *method*, *url*.`host`, *url*.`path`, *url*.`parameters`, $\langle\rangle$, *data*$\rangle$
70: **let** *reference* := $\langle$`XHR`, $s'.\overline{d}$.`nonce`, *xhrreference*$\rangle$
71: **call** HTTP_SEND(*reference*, *req*, *url*, *origin*, *referrer*, *referrerPolicy*, s')
72: **case** $\langle$`BACK`, *window*$\rangle$
73: **let** $\overline{w}'$:= GETNAVIGABLEWINDOW($\overline{w}$, *window*, $\bot$, s')
74: NAVBACK($\overline{w'}$, s')
75: **stop** $\langle\rangle$, s'
76: **case** $\langle$`FORWARD`, *window*$\rangle$
77: **let** $\overline{w}'$:= GETNAVIGABLEWINDOW($\overline{w}$, *window*, $\bot$, s')
78: NAVFORWARD($\overline{w'}$, s')
79: **stop** $\langle\rangle$, s'
80: **case** $\langle$`CLOSE`, *window*$\rangle$
81: **let** $\overline{w}'$:= GETNAVIGABLEWINDOW($\overline{w}$, *window*, $\bot$, s')
82: **remove** $s'.\overline{w'}$ from the sequence containing it
83: **stop** $\langle\rangle$, s'
84: **case** $\langle$`POSTMESSAGE`, *window*, *message*, *origin*$\rangle$
85: **let** $\overline{w}'$ $\leftarrow$ Subwindows(s') **such that** $s'.\overline{w}'$.`nonce` $\equiv$ *window*
86: **if** $\exists \bar{j} \in \mathbb{N}$ **such that** $s'.\overline{w'}$.`documents`.$\bar{j}$.`active` $\equiv \top$
$\hookrightarrow$ $\wedge(\mathit{origin} \not\equiv \bot \implies s'.\overline{w'}$.`documents`.$\bar{j}$.`origin` $\equiv \mathit{origin})$ **then**
87: **let** $s'.\overline{w'}$.`documents`.$\bar{j}$.`scriptinputs` := $s'.\overline{w'}$.`documents`.$\bar{j}$.`scriptinputs`
$\hookrightarrow$ $+^{\langle\rangle}$ $\langle$`POSTMESSAGE`, $s'.\overline{w}$.`nonce`, *docorigin*, *message*$\rangle$
88: **stop** $\langle\rangle$, s'
89: **case** else
90: **stop**

Algorithm 9 Web Browser Model: Process an HTTP response.

function PROCESSRESPONSE($response$, $reference$, $request$, $requestUrl$, key, f, s')
  if `Set-Cookie` $\in$ $response$.`headers` **then**
    for each $c \in^{\langle\rangle}$ $response$.`headers` [`Set-Cookie`], $c \in$ Cookies **do**
      let s'.`cookies` [$request$.`host`] $\hookrightarrow$:= AddCookie(s'.`cookies` [$request$.`host`], c)
  if `Strict-Transport-Security` $\in$ $response$.`headers` $\wedge$ $requestUrl$.`protocol` $\equiv$ `S` **then**
    let s'.`sts` := s'.`sts` $+^{\langle\rangle}$ $request$.`host`
  if `Referer` $\in$ $request$.`headers` **then**
    let $referrer$:= $request$.`headers`[`Referer`]
  else
    let $referrer$:= $\bot$
  if `Location` $\in$ $response$.`headers` $\wedge$ $response$.`status` $\in \{303, 307\}$ **then**
    let url := $response$.`headers` [`Location`]
    if url.`fragment` $\equiv \bot$ **then**
      let url.`fragment` := $requestUrl$.`fragment`
    let $method'$:= $request$.`method`
    let $body'$:= $request$.`body`
    if `Origin` $\in$ $request$.`headers` **then**
      let $origin$:= $\langle request.\texttt{headers}[\texttt{Origin}], \langle request.\texttt{host}, url.\texttt{protocol}\rangle\rangle$
    else
      let $origin$:= $\bot$
    if $response$.`status` $\equiv 303 \wedge$ $request$.`method` $\notin \{$`GET`, `HEAD`$\}$ **then**
      let $method'$:= `GET`
      let $body'$:= $\langle\rangle$
    if $\exists\, \overline{w} \in$ Subwindows(s') **such that** $s'.\overline{w}$.`nonce` $\equiv$ $reference$ **then** $\rightarrow$ Do not redirect XHRs.
      let req := $\langle$`HTTPReq`, ν_6, $method'$, url.`host`, url.`path`, url.`parameters`, $\langle\rangle$, $body'\rangle$
      let $referrerPolicy$:= $response$.`headers`[`ReferrerPolicy`]
      call HTTP_SEND($reference$, req, url, $origin$, $referrer$, $referrerPolicy$, s')
  switch $\pi_1(reference)$ **do**
    case `REQ`
      let $\overline{w} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}$.`nonce` $\equiv \pi_2(reference)$ **if possible; otherwise stop** $\hookrightarrow$ $\rightarrow$ normal response
      if $response$.`body` $\not\sim \langle *, * \rangle$ **then**
        stop $\{\}$, s'
      let $script$:= $\pi_1(response.\texttt{body})$
      let $scriptstate$:= $\pi_2(response.\texttt{body})$
      let d := $\langle \nu_7, requestUrl, response.\texttt{headers}, referrer, script, scriptstate, \langle\rangle, \langle\rangle, \top\rangle$
      if $s'.\overline{w}$.`documents` $\equiv \langle\rangle$ **then**
        let $s'.\overline{w}$.`documents` := $\langle d \rangle$
      else
        let $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}$.`documents`.$\overline{i}$.`active` $\equiv \top$
        let $s'.\overline{w}$.`documents`.$\overline{i}$.`active` := $\bot$
        remove $s'.\overline{w}$.`documents`.$(\overline{i}+1)$ and all following documents from $s'.\overline{w}$.`documents` $\hookrightarrow$
        let $s'.\overline{w}$.`documents` := $s'.\overline{w}$.`documents` $+^{\langle\rangle}$ d
      stop $\{\}$, s'
    case `XHR`
      let $\overline{w} \leftarrow$ Subwindows(s'), $\overline{d}$ **such that** $s'.\overline{d}$.`nonce` $\equiv \pi_2(reference)$ $\wedge\ s'.\overline{d} = s'.\overline{w}$.`activedocument` $\hookrightarrow$ **if possible; otherwise stop** $\rightarrow$ process XHR response
      let $headers$:= $response$.`headers` $-$ `Set-Cookie`
      let $s'.\overline{d}$.`scriptinputs` := $s'.\overline{d}$.`scriptinputs` $+^{\langle\rangle}$ $\langle$`XMLHTTPREQUEST`, $headers$, $response$.`body`, $\pi_3(reference)\rangle$
      stop $\{\}$, s'

Placeholder	Usage
ν_1	Algorithm 10, new window nonces
ν_2	Algorithm 10, new HTTP request nonce
ν_3	Algorithm 10, lookup key for pending HTTP requests entry
ν_4	Algorithm 8, new HTTP request nonce (multiple lines)
ν_5	Algorithm 8, new subwindow nonce
ν_6	Algorithm 9, new HTTP request nonce
ν_7	Algorithm 9, new document nonce
ν_8	Algorithm 5, lookup key for pending DNS entry
ν_9	Algorithm 2, new window nonce
$\nu_{10}, \ldots$	Algorithm 8, replacement for placeholders in script output

Table I List of placeholders used in browser algorithms.

for navigation (e.g., for opening a link). When it is given a window reference (nonce) $window$, this function returns a pointer to a selected window term in s':
  - If $window$ is the string `_BLANK`, a new window is created and a pointer to that window is returned.
  - If $window$ is a nonce (reference) and there is a window term with a reference of that value in the windows in s', a pointer $\overline{w'}$ to that window term is returned, as long as the window is navigable by the current window's document (as defined by NavigableWindows above).

  In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).
- The function GETWINDOW (Algorithm 3) takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.
- The function CANCELNAV (Algorithm 4) is used to stop any pending requests for a specific window. From the pending requests and pending DNS requests it removes any requests with the given window reference n.
- The function HTTP_SEND (Algorithm 5) takes an HTTP request $message$ as input, adds cookie and origin headers to the message, creates a DNS request for the hostname given in the request and stores the request in s'.`pendingDNS` until the DNS resolution finishes. For normal HTTP requests, $reference$ is a window reference. For XHRs, $reference$ is a value of the form $\langle document, nonce \rangle$ where $document$ is a document reference and $nonce$ is some nonce that was chosen by the script that initiated the request. url contains the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). $origin$ is the origin header value that is to be added to the HTTP request.
- The functions NAVBACK (Algorithm 6) and NAVFORWARD (Algorithm 7), navigate a window forward or backward. More precisely, they deactivate one document and activate that document's succeeding document or preceding document, respectively. If no such successor/predecessor exists, the functions do not change the state.
- The function RUNSCRIPT (Algorithm 8) performs a script execution step of the script in the document $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the $command$ that the script issued is interpreted.
- The function PROCESSRESPONSE (Algorithm 9) is responsible for processing an HTTP response ($response$) that was received as the response to a request ($request$) that was sent earlier. In $reference$, either a window or a document reference is given (see explanation for Algorithm 5 above). $requestUrl$ contains the URL used when retrieving the document.
  The function first saves any cookies that were contained in the response to the browser state, then checks whether a redirection is requested (Location header). If that is not the case, the function creates a new document (for normal requests) or delivers the contents of the response to the respective receiver (for XHR responses).

d) Browser Relation: We can now define the *relation $R_{webbrowser}$ of a web browser atomic process* as follows:

Definition 42. The pair $((\langle\langle a, f, m\rangle\rangle, s), (M, s'))$ belongs to $R_{\text{webbrowser}}$ iff the non-deterministic Algorithm 10 (or any of the functions called therein), when given $(\langle a, f, m\rangle, s)$ as input, terminates with **stop** M, s', i.e., with output M and s'.

Recall that $\langle a, f, m\rangle$ is an (input) event and s is a (browser) state, M is a sequence of (output) protoevents, and s' is a new (browser) state (potentially with placeholders for nonces).

Algorithm 10 Web Browser Model: Main Algorithm.

Input: $\langle a, f, m\rangle, s$

let $s' := s$
if $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
  let $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests}\rangle$ → Collect incoming messages
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle, s'$
if $m \equiv \texttt{TRIGGER}$ **then** → A special trigger message.
  let $switch \leftarrow \{\texttt{script}, \texttt{urlbar}, \texttt{reload}, \texttt{forward}, \texttt{back}\}$
  let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
  let $\overline{tlw} \leftarrow \mathbb{N}$ **such that** $s'.\overline{tlw}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some top-level window.
  if $switch \equiv \texttt{script}$ **then** → Run some script.
    let $\overline{d} := \overline{w} +^{\langle\rangle} \texttt{activedocument}$
    call RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
  else if $switch \equiv \texttt{urlbar}$ **then** → Create some new request.
    let $newwindow \leftarrow \{\top, \bot\}$
    if $newwindow \equiv \top$ **then** → Create a new window.
      let $windownonce := \nu_1$
      let $w' := \langle windownonce, \langle\rangle, \bot\rangle$
      let $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
    else → Use existing top-level window.
      let $windownonce := s'.\overline{tlw}.nonce$
    let $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$
    let $host \leftarrow \mathsf{Doms}$
    let $path \leftarrow \mathbb{S}$
    let $fragment \leftarrow \mathbb{S}$
    let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
    let $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$
    let $req := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, host, path, parameters, \langle\rangle, \langle\rangle\rangle$
    call HTTP_SEND($windownonce$, req, url, $\bot$, $\bot$, $\bot$, s')
  else if $switch \equiv \texttt{reload}$ **then** → Reload some document.
    let $url := s'.\overline{w}.\texttt{activedocument.location}$
    let $req := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
    let $referrer := s'.\overline{w}.\texttt{activedocument.referrer}$
    let $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\texttt{nonce}, s')$
    call HTTP_SEND($s'.\overline{w}.\texttt{nonce}$, req, url, $\bot$, $referrer$, $\bot$, s')
  else if $switch \equiv \texttt{forward}$ **then**
    NAVFORWARD($\overline{w}$, s')
  else if $switch \equiv \texttt{back}$ **then**
    NAVBACK($\overline{w}$, s')

G. Definition of Web Browsers

Finally, we define web browser atomic Dolev-Yao processes as follows:

Definition 43 (Web Browser atomic Dolev-Yao Process). A web browser atomic Dolev-Yao process is an atomic Dolev-Yao process of the form $p = (I^p, Z_{\text{webbrowser}}, R_{\text{webbrowser}}, {s_o}^p)$ for a set I^p of addresses, $Z_{\text{webbrowser}}$ and $R_{\text{webbrowser}}$ as defined above, and an initial state ${s_0}^p \in Z_{\text{webbrowser}}$.

H. Script Notations and Helper Functions

In order to simplify the description of scripts, we use several notations and helper functions. In the formal description of the scripts we use an abbreviation for URLs. We write URL^d_{path} to describe the following URL term: $\langle \texttt{URL}, \texttt{S}, d, path, \langle\rangle\rangle$. If the domain d belongs to some distinguished process P and it is the only domain associated to this process, we may also write $\mathsf{URL}^{\mathsf{P}}_{path}$. For a (secure) origin $\langle d, \texttt{S}\rangle$ of some domain d, we also write origin_d. Again, if the domain d belongs to some distinguished process P and d is the only domain associated to this process, we may write $\mathsf{origin}_{\mathsf{P}}$.

a) CHOOSEINPUT (Algorithm 11): The state of a document contains a term, say *scriptinputs*, which records the input this document has obtained so far (via XHRs and postMessages). If the script of the document is activated, it will typically need to pick one input message from *scriptinputs* and record which input it has already processed. For this purpose, the function $\mathsf{CHOOSEINPUT}(s', scriptinputs)$ is used, where s' denotes the scripts current state. It saves the indexes of already handled messages in the

else if $m \equiv$ `FULLCORRUPT` **then** → Request to corrupt browser
  let s'.`isCorrupted` := `FULLCORRUPT`
  stop $\langle\rangle$, s'
else if $m \equiv$ `CLOSECORRUPT` **then** → Close the browser
  let s'.`secrets` := $\langle\rangle$
  let s'.`windows` := $\langle\rangle$
  let s'.`pendingDNS` := $\langle\rangle$
  let s'.`pendingRequests` := $\langle\rangle$
  let s'.`sessionStorage` := $\langle\rangle$
  let s'.`cookies` $\subset^{\langle\rangle}$ Cookies **such that**
    ↪ $(c \in^{\langle\rangle} s'.\texttt{cookies}) \iff (c \in^{\langle\rangle} s.\texttt{cookies} \wedge c.\texttt{content.session} \equiv \bot)$
  let s'.`isCorrupted` := `CLOSECORRUPT`
  stop $\langle\rangle$, s'
else if $\exists \langle reference, request, url, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
    ↪ $\pi_1(\mathsf{dec_s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
  let m' := $\mathsf{dec_s}(m, key)$
  if $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
    stop
  remove $\langle reference, request, url, key, f\rangle$ **from** s'.`pendingRequests`
  call PROCESSRESPONSE(m', $reference$, $request$, url, key, f, s')
else if $\pi_1(m) \equiv \texttt{HTTPResp} \wedge \exists \langle reference, request, url, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$ **such that**
  ↪ $m.\texttt{nonce} \equiv request.\texttt{nonce}$ **then** → Plain HTTP Response
  remove $\langle reference, request, url, \bot, f\rangle$ **from** s'.`pendingRequests`
  call PROCESSRESPONSE(m, $reference$, $request$, url, key, f, s')
else if $m \in$ DNSResponses **then** → Successful DNS response
  if $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin \mathsf{IPs} \vee m.\texttt{domain} \not\equiv \pi_2(s.\texttt{pendingDNS}).\texttt{host}$ **then**
    stop
  let $\langle reference, message, url\rangle$:= $s.\texttt{pendingDNS}[m.\texttt{nonce}]$
  if $url.\texttt{protocol} \equiv \texttt{S}$ **then**
    let s'.`pendingRequests` := s'.`pendingRequests`
      ↪ $+^{\langle\rangle} \langle reference, message, url, \nu_3, m.\texttt{result}\rangle$
    let $message$:= $\mathsf{enc_s}(\langle message, \nu_3\rangle, s'.\texttt{keyMapping}[message.\texttt{host}])$
  else
    let s'.`pendingRequests` := s'.`pendingRequests`
      ↪ $+^{\langle\rangle} \langle reference, message, url, \bot, m.\texttt{result}\rangle$
  let s'.`pendingDNS` := s'.`pendingDNS` $- m.\texttt{nonce}$
  stop $\langle\langle m.\texttt{result}, a, message\rangle\rangle$, s'
stop

Algorithm 11 Function to retrieve an unhandled input message for a script.

function CHOOSEINPUT(s', $scriptinputs$)
  let iid **such that** $iid \in \{1, \cdots, |scriptinputs|\} \wedge iid \notin^{\langle\rangle} s'.\texttt{handledInputs}$ **if possible;**
    ↪ **otherwise return** $(\bot, s')$
  let $input$:= $\pi_{iid}(scriptinputs)$
  let s'.`handledInputs` := s'.`handledInputs` $+^{\langle\rangle} iid$
  return $(input, s')$

scriptstate s' and chooses a yet unhandled input message from $scriptinputs$. The index of this message is then saved in the scriptstate (which is returned to the script).

b) CHOOSEFIRSTINPUTPAT (Algorithm 12): Similar to the function CHOOSEINPUT above, we define the function CHOOSEFIRSTINPUTPAT. This function takes the term $scriptinputs$, which as above records the input this document has obtained so far (via XHRs and postMessages, append-only), and a pattern. If called, this function chooses the first message in $scriptinputs$ that matches $pattern$ and returns it. This function is typically used in places, where a script only processes the first message that matches the pattern. Hence, we omit recording the usage of an input.

Algorithm 12 Function to extract the first script input message matching a specific pattern.

function CHOOSEFIRSTINPUTPAT($scriptinputs$, $pattern$)
  let i **such that** $i = \min\{j : \pi_j(scriptinputs) \sim pattern\}$ **if possible; otherwise return** $\bot$
  return $\pi_i(scriptinputs)$

c) PARENTWINDOW: To determine the nonce referencing the parent window in the browser, the function PARENTWINDOW$(tree, docnonce)$ is used. It takes the term $tree$, which is the (partly cleaned) tree of browser windows the script is able to see and the document nonce $docnonce$, which is the nonce referencing the current document the script is running in, as input. It outputs the nonce referencing the window which directly contains in its subwindows the window of the document referenced by $docnonce$. If there is no such window (which is the case if the script runs in a document of a top-level window), PARENTWINDOW returns $\perp$.

d) PARENTDOCNONCE: The function PARENTDOCNONCE$(tree, docnonce)$ determines (similar to PARENTWINDOW above) the nonce referencing the active document in the parent window in the browser . It takes the term $tree$, which is the (partly cleaned) tree of browser windows the script is able to see and the document nonce $docnonce$, which is the nonce referencing the current document the script is running in, as input. It outputs the nonce referencing the active document in the window which directly contains in its subwindows the window of the document referenced by $docnonce$. If there is no such window (which is the case if the script runs in a document of a top-level window) or no active document, PARENTDOCNONCE returns $docnonce$.

e) SUBWINDOWS: This function takes a term $tree$ and a document nonce $docnonce$ as input just as the function above. If $docnonce$ is not a reference to a document contained in $tree$, then SUBWINDOWS$(tree, docnonce)$ returns $\langle\rangle$. Otherwise, let $\langle docnonce, origin, script, scriptstate, scriptinputs, subwindows, active\rangle$ denote the subterm of $tree$ corresponding to the document referred to by $docnonce$. Then, SUBWINDOWS$(tree, docnonce)$ returns $subwindows$.

f) AUXWINDOW: This function takes a term $tree$ and a document nonce $docnonce$ as input as above. From all window terms in $tree$ that have the window containing the document identified by $docnonce$ as their opener, it selects one non-deterministically and returns its nonce. If there is no such window, it returns the nonce of the window containing $docnonce$.

g) AUXDOCNONCE: Similar to AUXWINDOW above, the function AUXDOCNONCE takes a term $tree$ and a document nonce $docnonce$ as input. From all window terms in $tree$ that have the window containing the document identified by $docnonce$ as their opener, it selects one non-deterministically and returns its active document's nonce. If there is no such window or no active document, it returns $docnonce$.

h) OPENERWINDOW: This function takes a term $tree$ and a document nonce $docnonce$ as input as above. It returns the window nonce of the opener window of the window that contains the document identified by $docnonce$. Recall that the nonce identifying the opener of each window is stored inside the window term. If no document with nonce $docnonce$ is found in the tree $tree$ or $docnonce$ is not a top-level window, $\Diamond$ is returned.

i) GETWINDOW: This function takes a term $tree$ and a document nonce $docnonce$ as input as above. It returns the nonce of the window containing $docnonce$.

j) GETORIGIN: To extract the origin of a document, the function GETORIGIN$(tree, docnonce)$ is used. This function searches for the document with the identifier $docnonce$ in the (cleaned) tree $tree$ of the browser's windows and documents. It returns the origin o of the document. If no document with nonce $docnonce$ is found in the tree $tree$, $\Diamond$ is returned.

k) GETPARAMETERS: Works exactly as GETORIGIN, but returns the document's parameters instead.

I. DNS Servers

Definition 44. A *DNS server* d (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form

$$s_0^d = \langle\langle \mathtt{domain}_1, a_1\rangle, \langle \mathtt{domain}_2, a_2\rangle, \ldots\rangle \ .$$

DNS queries are answered according to this table (if the requested DNS name cannot be found in the table, the request is ignored).

The relation $R^d \subseteq (\mathcal{E} \times \{s_0^d\}) \times (2^{\mathcal{E}} \times \{s_0^d\})$ of d above is defined by Algorithm 13.

J. Web Systems

The web infrastructure and web applications are formalized by what is called a web system. A web system contains, among others, a (possibly infinite) set of DY processes, modeling web browsers, web servers, DNS servers, and attackers (which may corrupt other entities, such as browsers).

Definition 45. A *web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a tuple with its components defined as follows:

Algorithm 13 Relation of a DNS server R^d.

Input: $\langle a, f, m\rangle, s$
1: **let** $domain, n$ **such that** $\langle \texttt{DNSResolve}, domain, n\rangle \equiv m$ **if possible; otherwise stop** $\{\}, s$
2: **if** $domain \in s$ **then**
3: **let** $addr := s[domain]$
4: **let** $m' := \langle \texttt{DNSResolved}, addr, n\rangle$
5: **stop** $\{\langle f, a, m'\rangle\}, s$
6: **stop** $\{\}, s$

The first component, $\mathcal{W}$, denotes a system (a set of DY processes) and is partitioned into the sets Hon, Web, and Net of honest, web attacker, and network attacker processes, respectively.
Every $p \in \mathsf{Web} \cup \mathsf{Net}$ is an attacker process for some set of sender addresses $A \subseteq \mathsf{IPs}$. For a web attacker $p \in \mathsf{Web}$, we require its set of addresses I^p to be disjoint from the set of addresses of all other web attackers and honest processes, i.e., $I^p \cap I^{p'} = \emptyset$ for all $p' \in \mathsf{Hon} \cup \mathsf{Web}$. Hence, a web attacker cannot listen to traffic intended for other processes. Also, we require that $A = I^p$, i.e., a web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on I^p) and may spoof all addresses (i.e., the set A may be IPs).
Every $p \in \mathsf{Hon}$ is a DY process which models either a *web server*, a *web browser*, or a *DNS server*. Just as for web attackers, we require that p does not spoof sender addresses and that its set of addresses I^p is disjoint from those of other honest processes and the web attackers.
The second component, $\mathcal{S}$, is a finite set of scripts such that $R^{\text{att}} \in \mathcal{S}$. The third component, script, is an injective mapping from $\mathcal{S}$ to $\mathbb{S}$, i.e., by script every $s \in \mathcal{S}$ is assigned its string representation $\mathsf{script}(s)$. Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for every $a \in \bigcup_{p\in\mathcal{W}} I^p$.
A *run* of $\mathcal{WS}$ is a run of $\mathcal{W}$ initiated by E^0.

K. Generic HTTPS Server Model

This model will be used as the base for all servers in the following. It makes use of placeholder algorithms that are later superseded by more detailed algorithms to describe a concrete relation for an HTTPS server.

Definition 46 (Base state for an HTTPS server.). The state of each HTTPS server that is an instantiation of this relation must contain at least the following subterms: $pendingDNS \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $pendingRequests \in \mathcal{T}_{\mathcal{N}}$ (both containing arbitrary terms), $DNSaddress \in \mathsf{IPs}$ (containing the IP address of a DNS server), $keyMapping \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ (containing a mapping from domains to public keys), $tlskeys \in [\mathsf{Doms} \times \mathcal{N}]$ (containing a mapping from domains to private keys), and $corrupt \in \mathcal{T}_{\mathcal{N}}$ (either $\perp$ if the server is not corrupted, or an arbitrary term otherwise).

We note that in concrete instantiations of the generic HTTPS server model, there is no need to extract information from these subterms or alter these subterms.

Let ν_{n0} and ν_{n1} denote placeholders for nonces that are not used in the concrete instantiation of the server. We now define the default functions of the generic web server in Algorithms 14–18, and the main relation in Algorithm 19.

Algorithm 14 Generic HTTPS Server Model: Sending a DNS message (in preparation for sending an HTTPS message).

1: **function** HTTPS_SIMPLE_SEND($reference$, $message$, s', a)
2: **let** $s'.\texttt{pendingDNS}[\nu_{n0}] := \langle reference, message\rangle$
3: **stop** $\langle\langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, message.\texttt{host}, \nu_{n0}\rangle\rangle\rangle$, s'

Algorithm 15 Generic HTTPS Server Model: Default HTTPS response handler.

1: **function** PROCESS_HTTPS_RESPONSE(m, $reference$, $request$, key, a, f, s')
2: **stop**

Algorithm 16 Generic HTTPS Server Model: Default trigger event handler.

function TRIGGER(s')
    stop

Algorithm 17 Generic HTTPS Server Model: Default HTTPS request handler.

function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
    stop

Algorithm 18 Generic HTTPS Server Model: Default handler for other messages.

function PROCESS_OTHER(m, a, f, s')
    stop

Algorithm 19 Generic HTTPS Server Model: Main relation of a generic HTTPS server

Input: $\langle a, f, m\rangle, s$
let $s' := s$
if $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
    let $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
    let $m' \leftarrow d_V(s')$
    let $a' \leftarrow \mathsf{IPs}$
    stop $\langle\langle a', a, m'\rangle\rangle$, s'
if $\exists\, m_{\text{dec}}, k, k', inDomain$ **such that** $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle inDomain, k'\rangle \in s.\texttt{tlskeys}$ **then**
    let n, $method$, $path$, $parameters$, $headers$, $body$ **such that**
    ↪ $\langle \texttt{HTTPReq}, n, method, inDomain, path, parameters, headers, body\rangle \equiv m_{\text{dec}}$
    ↪ **if possible; otherwise stop**
    call PROCESS_HTTPS_REQUEST(m_{dec}, k, a, f, s')
else if $m \in$ DNSResponses **then** → Successful DNS response
    if $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin \mathsf{IPs} \vee m.\texttt{domain} \not\equiv s.\texttt{pendingDNS}[m.\texttt{nonce}].2.\texttt{domain}$ **then**
        stop
    let $\langle reference, request\rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
    let $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests}$
        ↪ $+^{\langle\rangle}\ \langle reference, request, \nu_{n1}, m.\texttt{result}\rangle$
    let $message := \mathsf{enc_a}(\langle request, \nu_{n1}\rangle, s'.\texttt{keyMapping}[request.\texttt{host}])$
    let $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
    stop $\langle\langle m.\texttt{result}, a, message\rangle\rangle$, s'
else if $\exists\, \langle reference, request, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
    ↪ **such that** $\pi_1(\mathsf{dec_s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
    let $m' := \mathsf{dec_s}(m, key)$
    if $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
        stop
    remove $\langle reference, request, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$
    call PROCESS_HTTPS_RESPONSE(m', $reference$, $request$, key, a, f, s')
    stop
else if $m \equiv \texttt{TRIGGER}$ **then** → Process was triggered
    call PROCESS_TRIGGER(s')
stop

L. General Security Properties of the WIM

We now repeat general application independent security properties of the WIM [22].

Let $\mathcal{Web} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E_0)$ be a web system. In the following, we write $s_x = (S_x, E_x)$ for the states of a web system.

Definition 47 (Emitting Events). Given an atomic process p, an event e, and a finite run $\rho = ((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite run $\rho = ((S^0, E^0, N^0), \ldots)$ we say that p *emits* e iff there is a processing step in ρ of the form

$$(S^i, E^i, N^i) \xrightarrow[p \to E]{} (S^{i+1}, E^{i+1}, N^{i+1})$$

for some $i \geq 0$ and a set of events E with $e \in E$. We also say that p *emits* m iff $e = \langle x, y, m\rangle$ for some addresses x, y.

Definition 48. We say that a term t *is derivably contained in (a term)* t' *for (a set of DY processes)* P *(in a processing step* $s_i \to s_{i+1}$ *of a run* $\rho = (s_0, s_1, \ldots)$*)* if t is derivable from t' with the knowledge

available to P, i.e.,

$$t \in d_\emptyset(\{t'\} \cup \bigcup_{p \in P} S^{i+1}(p))$$

Definition 49. We say that *a set of processes P leaks a term t (in a processing step $s_i \to s_{i+1}$) to a set of processes P'* if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \backslash P$. If P is a set with a single element, we omit the set notation.

Definition 50. We say that an DY process p *created* a message m (at some point) in a run if m is derivably contained in a message emitted by p in some processing step and if there is no earlier processing step where m is derivably contained in a message emitted by some DY process p'.

Definition 51. We say that *a browser b accepted* a message (as a response to some request) if the browser decrypted the message (if it was an HTTPS message) and called the function PROCESSRESPONSE, passing the message and the request (see Algorithm 9).

Definition 52. We say that an atomic DY process p *knows a term t* in some state $s = (S, E, N)$ of a run if it can derive the term from its knowledge, i.e., $t \in d_\emptyset(S(p))$.

Definition 53. We say that a *script initiated a request r* if a browser triggered the script (in Line 10 of Algorithm 8) and the first component of the $command$ output of the script relation is either `HREF`, `IFRAME`, `FORM`, or `XMLHTTPREQUEST` such that the browser issues the request r in the same step as a result.

For a run $\rho = s_0, s_1, \ldots$ of any $\mathcal{Web}$, we state the following lemmas:

Lemma 3. If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{Web}$ an honest browser b (I) emits an HTTPS request of the form

$$m = \mathsf{enc_a}(\langle req, k \rangle, \mathsf{pub}(k'))$$

(where req is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and (II) in the initial state s_0 the private key k' is only known to u, and (III) u never leaks k', then all of the following statements are true:

(1) There is no state of $\mathcal{Web}$ where any party except for u knows k', thus no one except for u can decrypt req.
(2) If there is a processing step $s_j \to s_{j+1}$ where the browser b leaks k to $\mathcal{W} \backslash \{u, b\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \backslash \{u, b\}$ or the browser is fully corrupted in s_j.
(3) The value of the host header in req is the domain that is assigned the public key $\mathsf{pub}(k')$ in the browsers' keymapping $s_0.\texttt{keymapping}$ (in its initial state).
(4) If b accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and b is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \backslash \{u, b\}$ prior to s_j, then u created the HTTPS response m' to the HTTPS request m, i.e., the nonce of the HTTP request req is not known to any atomic process p, except for the atomic process b and u.

Proof:

(1) follows immediately from the condition. If k' is initially only known to u and u never leaks k', i.e., even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u, k' cannot be known to any other party. Thus, nobody except for u can derive req from m.

(2) We assume that b leaks k to $\mathcal{W} \backslash \{u, b\}$ in the processing step $s_j \to s_{j+1}$ without u prior leaking the key k to anyone except for u and b and that the browser is not fully corrupted in s_j, and lead this to a contradiction.

The browser is honest in s_i. From the definition of the browser b, we see that the key k is always chosen as a fresh nonce (placeholder ν_3 in Lines 61ff. of Algorithm 10) that is not used anywhere else. Further, the key is stored in the browser's state in $pendingRequests$. The information from $pendingRequests$ is not extracted or used anywhere else (in particular it is not accessible by scripts). If the browser becomes closecorrupted prior to s_j (and after s_i), the key cannot be used anymore (compare Lines 43ff. of Algorithm 10). Hence, b does not leak k to any other party in s_j (except for u and b). This proves (2).

(3) Per the definition of browsers (Algorithm 10), a host header is always contained in HTTP requests by browsers. From Line 67 of Algorithm 10 we can see that the encryption key for the request req was

chosen using the host header of the message. It is chosen from the $keymapping$ in the browser's state, which is never changed during ρ. This proves (3).

(4) An HTTPS response m' that is accepted by b as a response to m has to be encrypted with k. The nonce k is stored by the browser in the $pendingRequests$ state information. The browser only stores freshly chosen nonces there (i.e., the nonces are not used twice, or for other purposes than sending one specific request). The information cannot be altered afterwards (only deleted) and cannot be read except when the browser checks incoming messages. The nonce k is only known to u (which did not leak it to any other party prior to s_j) and b (which did not leak it either, as u did not leak it and b is honest, see (2)). The browser b cannot send responses. This proves (4).

■

Corollary 1. In the situation of Lemma 3, as long as u does not leak the symmetric key k to $\mathcal{W}' \setminus \{u, b\}$ and the browser does not become fully corrupted, k is not known to any DY process $p \notin \{b, u\}$ (i.e., $\nexists\, s' = (S', E') \in \rho$: $k \in d_{N^p}(S'(p))$).

Lemma 4. If for some $s_i \in \rho$ an honest browser b has a document d in its state $S_i(b).\texttt{windows}$ with the origin $\langle dom, \texttt{S}\rangle$ where $dom \in \mathsf{Domain}$, and $S_i(b).\texttt{keyMapping}[dom] \equiv \texttt{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then b extracted (in Line 35 in Algorithm 9) the script in that document from an HTTPS response that was created by p.

Proof: The origin of the document d is set only once: In Line 35 of Algorithm 9. The values (domain and protocol) used there stem from the information about the request (say, req) that led to loading of d. These values have been stored in $pendingRequests$ between the request and the response actions. The contents of $pendingRequests$ are indexed by freshly chosen nonces and can never be altered or overwritten (only deleted when the response to a request arrives). The information about the request req was added to $pendingRequests$ in Line 66 (or Line 69 which we can exclude as we will see later) of Algorithm 10. In particular, the request was an HTTPS request iff a (symmetric) key was added to the information in $pendingRequests$. When receiving the response to req, it is checked against that information and accepted only if it is encrypted with the proper key and contains the same nonce as the request (say, n). Only then the protocol part of the origin of the newly created document becomes S. The domain part of the origin (in our case dom) is taken directly from the $pendingRequests$ and is thus guaranteed to be unaltered.

From Line 67 of Algorithm 10 we can see that the encryption key for the request req was actually chosen using the host header of the message which will finally be the value of the origin of the document d. Since b therefore selects the public key $S_i(b).\texttt{keyMapping}[dom] = S_0(b).\texttt{keyMapping}[dom] \equiv \texttt{pub}(k)$ for p (the key mapping cannot be altered during a run), we can see that req was encrypted using a public key that matches a private key which is only (if at all) known to p. With Lemma 3 we see that the symmetric encryption key for the response, k, is only known to b and the respective web server. The same holds for the nonce n that was chosen by the browser and included in the request. Thus, no other party than p can encrypt a response that is accepted by the browser b and which finally defines the script of the newly created document.

■

Lemma 5. If in a processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{Web}$ an honest browser b issues an HTTP(S) request with the Origin header value $\langle dom, \texttt{S}\rangle$ where and $S_i(b).\texttt{keyMapping}[dom] \equiv \texttt{pub}(k)$ with $k \in \mathcal{N}$ being a private key, and there is only one DY process p that knows the private key k in all s_j, $j \leq i$, then that request was initiated by a script that b extracted (in Line 35 in Algorithm 9) from an HTTPS response that was created by p.

Proof: First, we can see that the request was initiated by a script: As it contains an origin header, it must have been a POST request (see the browser definition in Appendix B-F). POST requests can only be initiated in Lines 50, 71 of Algorithm 8 and Line 27 of Algorithm 9. In the latter instance (Location header redirect), the request contains at least two different origins, therefore it is impossible to create a request with exactly the origin $\langle dom, \texttt{S}\rangle$ using a redirect. In the other two cases (FORM and XMLHTTPRequest), the request was initiated by a script.

The Origin header of the request is defined by the origin of the script's document. With Lemma 4 we see that the content of the document, in particular the script, was indeed provided by p.

■

APPENDIX C
DATA STRUCTURES

The following definitions extend the set of data structures of the original WIM (see Appendix B) and are used throughout the extended model and its analysis.

Method Data. First, we give the definitions for the identifier of a payment method, followed by the definition of the method data object.

Definition 54 (Payment Method Identifier). A Payment Method Identifier is a (WIM) URL for which the following holds true:

$$protocol = S$$

PaymentMethodIdentifiers is the set of all possible Payment Method Identifiers.

Definition 55 (Payment Method Data). A payment method data term is a term consisting of a sequence of terms $\langle x_1, x_2, \dots \rangle$, where each x_i is a term $\langle pmi, receiver, paymentIdentifier \rangle$ with $pmi \in$ PaymentMethodIdentifiers, $receiver \in \mathcal{T}_{\mathcal{N}}$, and $paymentIdentifier \in \mathbb{S}$. Furthermore, we require for all x_i, x_j in the sequence that $x_i.\texttt{paymentIdentifier} = x_j.\texttt{paymentIdentifier}$.
The set of all possible payment method data terms is denoted by MethodDatas.

Note that while the requirement for equal `paymentIdentifiers` restricts the set of allowed payment method data terms, this definition does not weaken our properties: The Payment Request API specification [14] defines the `paymentIdentifier` of a method data object as the `paymentIdentifier` contained in the respective payment request (that also contained the method data objects in question). I.e., when a script accesses the `paymentIdentifier` of a method data object, the browser does not "look" at the method data object at all, but takes the value from the respective payment request. In our model, we chose to include the `paymentIdentifier` in the method data terms, because that makes passing the relevant data around easier.

Service Workers. In the following, we give the definition of a service worker registration and related data structures.

Definition 56 (Payment Instrument). A Payment Instrument is a term

$$\langle instrumentKey, enabledMethods \rangle$$

where $instrumentKey \in \mathbb{S}$ and $enabledMethods \in$ PaymentMethodIdentifiers. Note that despite the name, $enabledMethods$ is only a single value. This name is taken from the WPA specifications where the plural identifier is used for historical reasons.

Definition 57 (Payment Manager). A Payment Manager is a term consisting of a series of Payment Instruments, i.e., $\langle x_1, x_2, \dots \rangle$.
The set of all Payment Managers is denoted by PaymentManagers.

Definition 58 (Service Worker Registration). A Service Worker Registration is defined through a term:

$$\langle nonce, scope, script, scriptinputs, scriptstate, paymentManager, trusted \rangle$$

where $nonce \in \mathcal{N}$, $scope \in \mathsf{URLs}$, $script \in \mathcal{T}_{\mathcal{N}}$, $scriptinputs \in \mathcal{T}_{\mathcal{N}}$, $scriptstate \in \mathcal{T}_{\mathcal{N}}$, $paymentManager \in$ PaymentManagers, and $trusted \in \{\bot, \top\}$.
For a Service Worker Registration sw we write $sw.\texttt{origin}$ to denote the origin of the service worker, i.e., the term $\langle sw.\texttt{scope.host}, sw.\texttt{scope.protocol} \rangle \in \mathsf{Origins}$. We call the set of all Service Worker Registrations ServiceWorkerRegistrations.

Substructures of Payment Objects. The messages and events exchanged between the WPA parties contain some substructures which we define here.

Definition 59 (Payment Details). Payment details are terms with the following structure:

$$\langle modifiers, total, shippingOptions \rangle$$

where $modifiers \in \mathcal{T}_{\mathcal{N}}$, $total \in \mathcal{T}_{\mathcal{N}}$, and $shippingOptions \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ is a sequence of terms.
The set of all payment details is called PaymentDetails.

Definition 60 (Payment Options). A payment options element is a term with the following structure:

$$\langle requestBillingAddress, requestShipping, requestPayerInfo \rangle$$

where $requestBillingAddress, requestShipping, requestPayerInfo \in \{\top, \bot\}$.
The set of all payment options is called PaymentOptions.

Messages and Events. Here, we define the messages and events used to exchange data between the WPA parties.

Definition 61 (Payment Request). A Payment Request is a term:

$$\langle \texttt{PAYMENTREQUEST}, paymentRequestNonce, documentnonce, methodData, details,$$
$$options, state, updating \rangle$$

where $paymentRequestNonce \in \mathcal{N}$, $documentnonce \in \mathcal{N}$, $methodData \in \mathcal{T}_{\mathcal{N}}$, $details \in$ PaymentDetails, $options \in$ PaymentOptions, $state \in \{CR, IN, CL\}$ (corresponding to the states: **Cr**eated, **In**teractive and **Cl**osed), and $updating \in \{\top, \bot\}$.

Definition 62 (Can Make Payment Event). A Can Make Payment Event is a term:

$$\langle \texttt{CANMAKEPAYMENT}, handlerNonce, topOrigin, paymentRequestOrigin, methodData \rangle$$

where $handlerNonce \in \mathcal{N}$, $topOrigin \in$ Origins, $paymentRequestOrigin \in$ Origins, and $methodData \subset^{\langle\rangle}$ MethodDatas.

Definition 63 (Submit Payment Event). A Submit Payment Event is a term:

$$\langle \texttt{SUBMITPAYMENT}, paymentRequestNonce, handlerNonce \rangle$$

where $paymentRequestNonce \in \mathcal{N}$ and $handlerNonce \in \mathcal{N}$.

Definition 64 (Payment Request Event). A Payment Request Event is a term:

$$\langle \texttt{PAYMENTREQUESTEVENT}, paymentRequestEventNonce, paymentRequestNonce, handlerNonce,$$
$$methodData, total, modifiers, instrumentKey, requestBillingAddress, transactionId \rangle$$

where $paymentRequestEventNonce \in \mathcal{N}$, $paymentRequestNonce \in \mathcal{N}$, $handlerNonce \in \mathcal{N}$, $methodData \subset^{\langle\rangle}$ MethodDatas, $total \in \mathcal{T}_{\mathcal{N}}$, $modifiers \in \mathcal{T}_{\mathcal{N}}$, $instrumentKey \in \mathcal{T}_{\mathcal{N}}$, $requestBillingAddress \in \{\bot, \top\}$, and $transactionId \in \mathcal{N}$.

Definition 65 (Payment Handler Response). A Payment Handler Response is a term:

$$\langle \texttt{PAYMENTHANDLERRESPONSE}, paymentRequestNonce, handlerNonce, methodName, details \rangle$$

where $paymentRequestNonce \in \mathcal{N}$, $handlerNonce \in \mathcal{N}$, $methodName \in \mathcal{T}_{\mathcal{N}}$, and $details \in \mathcal{T}_{\mathcal{N}}$.

Definition 66 (Payment Response). A Payment Response is a term:

$$\langle \texttt{PAYMENTRESPONSE}, paymentResponseNonce, paymentRequestNonce, handlerNonce, methodName,$$
$$details, shippingAddress, shippingOption, payerInfo, complete \rangle$$

where $paymentResponseNonce \in \mathcal{N}$, $paymentRequestNonce \in \mathcal{N}$, $handlerNonce \in \mathcal{N}$, $methodName \in \mathcal{T}_{\mathcal{N}}$, $details \in \mathcal{T}_{\mathcal{N}}$, $shippingAddress \in \mathcal{T}_{\mathcal{N}}$, $shippingOption \in \mathcal{T}_{\mathcal{N}}$, $payerInfo \in \mathcal{T}_{\mathcal{N}}$, and $complete \in \{\bot, \top\}$.

Payment Storage. The payment storage entry of the browser state stores payment relevant objects that may be accessed by scripts, service workers and the browser.

Definition 67 (Payment Storage). A $paymentStorage \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary, where for a nonce $PRN \in \mathcal{N}$, the term $paymentStorage[PRN]$ has the following structure:

$$\langle paymentRequest, paymentRequestEvents, paymentResponses, transactionId, handlerNonce \rangle$$

where $paymentRequest \in \mathcal{T}_{\mathcal{N}}$ is a term as defined in Definition 61. $paymentRequestEvents \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary from nonces to payment request events (see also Definition 64). $paymentResponses \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary from nonces to payment responses (see also Definition 66). $transactionId \in \mathcal{N}$ and $handlerNonce \in \mathcal{N}$ are nonces.

Transactions. In our model, a payment provider server has a sequence of transactions in its state, storing all transactions performed by this payment provider. In the following, we define the structure of this state entry.

Definition 68 (Transaction). $transactions \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary, where for a term $key \in \mathcal{T}_{\mathcal{N}}$ (typically a payment identifier), the transaction $transactions[key]$ has the following structure:

$$\langle sender, receiver, total, transactionId \rangle$$

where $sender, receiver, total \in \mathcal{T}_{\mathcal{N}}$, and $transactionId \in \mathcal{N}$.

APPENDIX D
THE EXTENDED WEB INFRASTUCTURE MODEL WITH WEB PAYMENT APIS

In the following, we introduce the extensions and adaptions applied to the WIM to model the WPA, as well as our generic models of a merchant, payment handler, and payment provider used to exercise the extended WIM.

On a high level, the extended WIM's browser model consists of four main algorithms which we adapted/created (plus lots of small helper functions): The first one is the actual browser process (Algorithm 20), which processes events from the main event queue. It is responsible for dispatching incoming events to the different helper functions. The second one is the algorithm modeling script execution within the browser (Algorithm 21), including the merchant's interaction with the WPA. The third one of these "main" algorithms is introduced by us and processes DOM events within the browser (Algorithm 22). The fourth one too is a result of our work and models the execution of a service worker (Algorithm 23).

To allow for all possible execution orderings, the browser was also extended with a set of pending DOM events, one of which is selected non-deterministically once the browser (also non-deterministically) chooses to process a DOM event. By adding this functionality, one could say that the browser model itself becomes a Dolev-Yao style model within the overall WIM.

Within the browser, different scripts, service workers and the browser itself have different states and pending scriptinputs, from which they derive messages, DOM events and commands.

To make our contributions easier to spot, we marked all "old" code in the following algorithms in this light gray, while new code is given in black.

A. General Remarks

The functions $\mathsf{secretOfID}(id)$ and $\mathsf{ownerOfID}(id)$ are defined analogously to [21]:

$\mathsf{secretOfID}(id)$ describes a bijective mapping $\mathsf{ID} \to \mathsf{Passwords}$ where $\mathsf{Passwords}$ is the set of passwords shared between browsers and payment providers. It can be used by a browser b to get the password of an ID id owned by this browser, i.e., $\mathsf{ownerOfID}(id) = b$.

B. Browser

Definition 69. The original definition of the set of states of a browser process $Z_{\text{Webbrowser}}$ is extended to fit the needs of the WPA. We extend $Z_{\text{Webbrowser}}$ by the subterms $serviceWorkers$, $paymentStorage$, $events$, $usedEvents$, and $paymentIntents$. A browser state is therefore defined through a term of the form:

$$\langle windows, ids, secrets, cookies, localStorage, sessionStorage, keyMapping, sts, DNSaddress, pendingDNS, pendingRequests, wsConnections, rtcConnections, \textbf{serviceWorkers}, \textbf{paymentStorage}, \textbf{events}, \textbf{usedEvents}, \textbf{paymentIntents}, isCorrupted\rangle$$

The subterms $windows$, ids, $secrets$, $cookies$, $localStorage$, $sessionStorage$, $keyMapping$, sts, $DNSaddress$, $pendingDNS$, $pendingRequests$, $wsConnections$, $rtcConnections$, and $isCorrupted$ are defined as in the original WIM (see Definition 36 in Appendix B-F2).

The remaining sub terms have the following form:

- $serviceWorkers \subset^{\langle\rangle}$ ServiceWorkerRegistrations is a list of service worker registrations. This subterm stores all information concerning service workers, such as their state, their scope and their inputs.
- $paymentStorage \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ stores payment relevant objects that may be accessed later on through scripts, service workers and the browser. More precisely, the nonces that are used as keys are payment request nonces, which serve as a unique identifier of a payment request within the browser. For a nonce $PRN \in \mathcal{N}$, the term $paymentStorage[PRN]$ has the following structure: $\langle paymentRequest, paymentRequestEvents, paymentResponses, transactionId, handlerNonce\rangle$. $paymentRequest \in \mathcal{T}_{\mathcal{N}}$ is a term as defined in Definition 61. $paymentRequestEvents \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary from nonces to payment request events. Payment request events are defined in Definition 64. $paymentResponses \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary from nonces to payment responses. Payment responses are defined in Definition 66. $transactionId \in \mathcal{N}$ and $handlerNonce \in \mathcal{N}$ are nonces.
- $events \subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$ stores a list of events that the browser may process at any time. This set of events is used to allow for arbitrary orderings of asynchronous events.

Placeholder	Usage
ν_{14}	Algorithm 21, lookup key for payment request
ν_{15}	Algorithm 22, placeholder for sensitive payer information
ν_{16}	Algorithm 21, lookup key for payment request events
ν_{17}	Algorithm 22, placeholder for shipping information
ν_{18}	Algorithm 21, placeholder for transaction identifier

Table II List of placeholders added for use in browser algorithms.

- $usedEvents \subset^{\langle\rangle} \mathcal{N}$ is a list of indices into the $events$ list. These indices describe the already processed events. Just marking events as processed instead of deleting them facilitates simpler statements of certain properties which would otherwise need to talk about previous system states.
- $paymentIntents \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ stores the transactions that were intended by the user. For a nonce $transactionId \in \mathcal{N}$, the term $paymentIntents[transactionId]$ is a sequence of payment requests (as defined in Definition 61), where the `methodData` entries are a subset of the `methodData` entries of the original payment request.
  In particular, this means that $paymentIntents[transactionId]$ contains information about all intended payments of each payment requests. This list of historical intents is necessary as an arbitrary amount of intents can be expressed within one payment request due to the retry mechanism.

In an initial state s_0^b of a browser we initialize the new subterms as follows:

- $s_0^b.serviceWorkers$ is a non-deterministically sampled, finite set of service worker registrations. Note that these service worker registrations can be partitioned into honest and malicious service workers by use of their trusted flag. Furthermore, the trusted flag is constrained as follows: If there exists a payment provider server that is honest and serves the scope of the serviceWorker, then the trusted flag must be $\top$. This constraint must be added to keep the configuration in accordance with the modeled payment method manifest [37], i.e., honest payment providers only provide honest payment handlers and don't allow payment handlers from other origins for "their" payment method.
  Furthermore, we require that for each service worker registration sw, the nonce $sw.\texttt{nonce}$ is not used as a nonce for identifying a window. More precisely, we require that $\forall sw \in^{\langle\rangle} s_0^b.serviceWorkers\ \forall w \in^{\langle\rangle} s_0^b.windows : sw.\texttt{nonce} \neq w.\texttt{nonce}$.
- $s_0^b.paymentStorage \equiv \langle\rangle$
- $s_0^b.events \equiv \langle\rangle$
- $s_0^b.usedEvents \equiv \langle\rangle$
- $s_0^b.paymentIntents \equiv \langle\rangle$

Table II shows the newly introduced placeholders in the browser model. It extends the placeholder table of the browser as provided in [21]. The newly introduced placeholders either serve as placeholders for lookup nonces or as placeholders for sensitive information that an attacker should not be able to obtain, e.g., the shipping address.

Extension of Windows. The window term is extended with a boolean flag $paymentRequestShowing$. A window therefore is a term of the form $w = \langle nonce, documents, opener, paymentRequestShowing\rangle$ with $nonce$, $documents$ and $opener$ defined as in the original model and $paymentRequestShowing \in \{\top, \bot\}$. A window term is always initialized with $w.paymentRequestShowing = \bot$.

C. Main Browser Algorithm

The main algorithm of the web browser (Algorithm 20) is only extended in a few places. Its main additions are the support for running service workers and processing the internal browser events.

Algorithm 20 Web Browser Model: Main Algorithm.

Input: $\langle a, f, m\rangle, s$

let $s' := s$

Check if browser is corrupted

if $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
  let $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests}\rangle$ → Collect incoming messages
  let $n \leftarrow \mathbb{N}$
  let $m'_1, \ldots, m'_n \leftarrow d_V(s')$ → Create n new messages non-deterministically.
  let $a'_1, \ldots, a'_n \leftarrow$ IPs
  stop $\langle\langle a'_1, a, m'_1\rangle, \ldots, \langle a'_n, a, m'_n\rangle\rangle, s'$

Receive trigger message

if $m \equiv \texttt{TRIGGER}$ **then**
  let $switch \leftarrow \{\texttt{script}, \texttt{worker}, \texttt{urlbar}, \texttt{reload}, \texttt{forward}, \texttt{back}, \texttt{event}\}$
  let $\overline{w} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some window.
  let $\overline{tlw} \leftarrow \mathbb{N}$ **such that** $s'.\overline{tlw}.\texttt{documents} \neq \langle\rangle$ **if possible; otherwise stop** → Pointer to some top-level window.
  if $switch \equiv \texttt{script}$ **then** → Run some script.
    let $\overline{d} := \overline{w}.\texttt{activedocument}$
    call RUNSCRIPT$(\overline{w}, \overline{d}, s')$
  else if $switch \equiv \texttt{worker}$ **then** → Run some service worker.
    let $sw \leftarrow s'.\texttt{serviceWorkers}$
    let $\overline{sw} \leftarrow \mathbb{N}$ **such that** $s'.\texttt{serviceWorkers}.\overline{sw}.\texttt{nonce} = sw.\texttt{nonce}$
    call RUNWORKER$(\overline{sw}, s')$
  else if $switch \equiv \texttt{urlbar}$ **then** → Create some new request.
    let $newwindow \leftarrow \{\top, \bot\}$
    if $newwindow \equiv \top$ **then** → Create a new window.
      let $windownonce := \nu_1$
      let $w' := \langle windownonce, \langle\rangle, \bot\rangle$
      let $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
    else → Use existing top-level window.
      let $windownonce := s'.\overline{tlw}.nonce$
    let $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$
    let $host \leftarrow$ Doms
    let $path \leftarrow \mathbb{S}$
    let $fragment \leftarrow \mathbb{S}$
    let $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
    let $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$
    let $req := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, host, path, parameters, \langle\rangle, \langle\rangle\rangle$
    call HTTP_SEND$(\langle \texttt{REQ}, windownonce\rangle, req, url, \bot, \bot, \bot, s')$
  else if $switch \equiv \texttt{reload}$ **then** → Reload some document.
    let $url := s'.\overline{w}.\texttt{activedocument.location}$
    let $req := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
    let $referrer := s'.\overline{w}.\texttt{activedocument.referrer}$
    let $s' :=$ CANCELNAV$(s'.\overline{w}.\texttt{nonce}, s')$
    call HTTP_SEND$(\langle \texttt{REQ}, s'.\overline{w}.\texttt{nonce}\rangle, req, url, \bot, referrer, \bot, s')$
  else if $switch \equiv \texttt{forward}$ **then**
    call NAVFORWARD$(\overline{w}, s')$
  else if $switch \equiv \texttt{back}$ **then**
    call NAVBACK$(\overline{w}, s')$
  else if $switch \equiv \texttt{event}$ **then**
    let $eventIndex \leftarrow \mathbb{N}$ **such that** $s'.\texttt{events}.\overline{eventIndex} \neq \langle\rangle$ **and** $\overline{eventIndex} \notin^{\langle\rangle} s'.\texttt{usedEvents}$ **if possible; otherwise stop** → Select an event that has not been processed yet
    let $e := s'.\texttt{events}.\overline{eventIndex}$
    let $s'.\texttt{usedEvents} := s'.\texttt{usedEvents} +^{\langle\rangle} \overline{eventIndex}$
    call PROCESSEVENT$(e, \overline{w}, s')$

Change corruption status

else if $m \equiv \texttt{FULLCORRUPT}$ **then** → Request to corrupt browser
  let $s'.\texttt{isCorrupted} := \texttt{FULLCORRUPT}$
  stop $\langle\rangle, s'$
else if $m \equiv \texttt{CLOSECORRUPT}$ **then** → Close the browser
  let $s'.\texttt{secrets} := \langle\rangle$
  let $s'.\texttt{windows} := \langle\rangle$
  let $s'.\texttt{pendingDNS} := \langle\rangle$
  let $s'.\texttt{pendingRequests} := \langle\rangle$
  let $s'.\texttt{sessionStorage} := \langle\rangle$
  let $s'.\texttt{cookies} \subset^{\langle\rangle}$ Cookies **such that** $(c \in^{\langle\rangle} s'.\texttt{cookies}) \iff (c \in^{\langle\rangle} s.\texttt{cookies} \wedge c.\texttt{content.session} \equiv \bot)$
  let $s'.\texttt{isCorrupted} := \texttt{CLOSECORRUPT}$
  stop $\langle\rangle, s'$

62: **else if** $m \in$ DNSResponses **then** → DNS response
63: **if** $m.\texttt{nonce} \notin s.\texttt{pendingDNS} \vee m.\texttt{result} \notin$ IPs
↪ $\vee\ m.\texttt{domain} \not\equiv \pi_2(s.\texttt{pendingDNS}[m.\texttt{nonce}]).\texttt{host}$ **then**
64: **stop**
65: **let** $\langle reference, message, url\rangle := s.\texttt{pendingDNS}[m.\texttt{nonce}]$
66: **if** $url.\texttt{protocol} \equiv \texttt{S}$ **then**
67: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle reference, message, url, \nu_3, m.\texttt{result}\rangle$
68: **let** $message := \texttt{enc}_\texttt{a}(\langle message, \nu_3\rangle, s'.\texttt{keyMapping}[message.\texttt{host}])$
69: **else**
70: **let** $s'.\texttt{pendingRequests} := s'.\texttt{pendingRequests} +^{\langle\rangle} \langle reference, message, url, \bot, m.\texttt{result}\rangle$
71: **let** $s'.\texttt{pendingDNS} := s'.\texttt{pendingDNS} - m.\texttt{nonce}$
72: **stop** $\langle\langle m.\texttt{result}, a, message\rangle\rangle, s'$
73: **else if** $\pi_1(m) \equiv \texttt{HTTPResp} \wedge \exists\langle reference, request, url, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
↪ **such that** $m.\texttt{nonce} \equiv request.\texttt{nonce}$ **then** → Plain HTTP Response
74: **remove** $\langle reference, request, url, \bot, f\rangle$ **from** $s'.\texttt{pendingRequests}$
75: **call** PROCESSRESPONSE$(m, reference, request, url, key, f, s')$
76: **else if** $m.1 \equiv \texttt{WS\_MSG} \wedge \exists\langle reference, nonce, \bot, f\rangle \in^{\langle\rangle} s'.\texttt{wsConnections}$
↪ **such that** $m.\texttt{nonce} \equiv nonce$ **then** → Plain Websocket Message
77: **call** DELIVER_TO_DOC$(\pi_2(reference), m, s')$
Encrypted messages
78: **else if** $\exists\langle reference, request, url, key, f\rangle \in^{\langle\rangle} s'.\texttt{pendingRequests}$
↪ **such that** $\pi_1(\texttt{dec}_\texttt{s}(m, key)) \equiv \texttt{HTTPResp}$ **then** → Encrypted HTTP response
79: **let** $m' := \texttt{dec}_\texttt{s}(m, key)$
80: **if** $m'.\texttt{nonce} \not\equiv request.\texttt{nonce}$ **then**
81: **stop**
82: **remove** $\langle reference, request, url, key, f\rangle$ **from** $s'.\texttt{pendingRequests}$
83: **call** PROCESSRESPONSE$(m', reference, request, url, key, f, s')$
84: **else if** $\exists\langle reference, nonce, key, f\rangle \in^{\langle\rangle} s'.\texttt{wsConnections}$
↪ **such that** $\pi_1(\texttt{dec}_\texttt{s}(m, key)) \equiv \texttt{WS\_MSG}$ **then** → Encrypted Websocket Message
85: **let** $m' := \texttt{dec}_\texttt{s}(m, key)$
86: **if** $m'.\texttt{nonce} \not\equiv nonce$ **then**
87: **stop**
88: **call** DELIVER_TO_DOC$(\pi_2(reference), m', s')$
89: **else if** $\exists\langle nonce, info\rangle \in^{\langle\rangle} s'.\texttt{rtcConnections}$
↪ **such that** $\pi_1(\texttt{dec}_\texttt{a}(m, info.\texttt{privkey})) \equiv \texttt{RTC\_MSG}$ **then** → WebRTC message
90: **let** $m' := \texttt{dec}_\texttt{a}(m, info.\texttt{privkey})$
91: **if** $m'.\texttt{nonce} \not\equiv nonce$ **then**
92: **stop**
93: **let** $docnonce := info.\texttt{docnonce}$
94: **call** DELIVER_TO_DOC$(docnonce, m', s')$
95: **stop**

D. Running Scripts in the Browser

Algorithm 21 models the execution of a script with the function RUNSCRIPT. Since all earlier functionalities and interfaces are still exposed to the scripts, a big part of the algorithm stays untouched. The only old functionality that was adapted in the script is concerning the delivery of postMessages. Service workers can receive postMessages in the new model as well. Therefore, if not given a window nonce but a service worker nonce, the browser submits the message to the corresponding service worker.

The biggest additions to RUNSCRIPT are the extensions to the script APIs introduced by the WPA. PR_CREATE, PR_SHOW, PR_CANMAKEPAYMENT, and PR_ABORT model the corresponding JavaScript functions create(), show(), canMakePayment() and abort() of a PaymentRequest. PRES_COMPLETE, and PRES_RETRY model the corresponding JavaScript functions complete() and retry() of the PaymentRequestResponse object. PR_GET models later reading of fields of a payment request if changed. PR_UPDATE_DETAILS models the updateWith() method of the PaymentRequestUpdateEvent. To keep the model simple, this method is directly exposed to the PaymentRequest in this model, without the need for a PaymentRequestUpdateEvent.

Algorithm 21 Web Browser Model: Execute a script.

function RUNSCRIPT($\overline{w}, \overline{d}, s'$) → Pointers to window and active document within window plus browser state
  let $tree := \mathsf{Clean}(s', s'.\overline{d})$
  let $cookies := \langle \{ \langle c.\texttt{name}, c.\texttt{content.value} \rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
  ↪ $\wedge\, c.\texttt{content.httpOnly} = \bot$
  ↪ $\wedge\, \left(c.\texttt{content.secure} \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S}\right)\right)\} \rangle$
  let $tlw \leftarrow s'.\texttt{windows}$ **such that** tlw is the top-level window containing $\overline{d}$
  let $sessionStorage := s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce} \rangle\right]$
  let $localStorage := s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $secrets := s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $R \leftarrow \mathsf{script}^{-1}(s'.\overline{d}.\texttt{script})$
  let $in := \langle tree, s'.\overline{d}.\texttt{nonce}, s'.\overline{d}.\texttt{scriptstate}, s'.\overline{d}.\texttt{scriptinputs}, cookies,$
  ↪ $localStorage, sessionStorage, s'.\texttt{ids}, secrets \rangle$
  let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
  ↪ $cookies' \leftarrow \mathsf{Cookies}^{\nu}$, $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
  ↪ $sessionStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$, $command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
  ↪ $out := \langle state', cookies', localStorage',$
  ↪ $sessionStorage', command \rangle$
  ↪ **such that** $out = out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \ldots]$
  ↪ **with** $(in, out^{\lambda}) \in R$
  let $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
  ↪ $:= \mathsf{CookieMerge}(s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right], cookies')$
  let $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right] := localStorage'$
  let $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce} \rangle\right] := sessionStorage'$
  let $s'.\overline{d}.\texttt{scriptstate} := state'$
  switch $command$ **do**
    case $\langle \texttt{HREF}, url, hrefwindow, noreferrer \rangle$
      let $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, hrefwindow, noreferrer, s')$
      let $req := \langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, \langle\rangle, url.\texttt{parameters}, \langle\rangle \rangle$
      if $noreferrer \equiv \top$ **then**
        let $referrerPolicy := \texttt{noreferrer}$
      else
        let $referrerPolicy := s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
      let $s' := \mathsf{CANCELNAV}(s'.\overline{w}'.\texttt{nonce}, s')$
      call $\mathsf{HTTP\_SEND}(\langle \texttt{REQ}, s'.\overline{w}'.\texttt{nonce} \rangle, req, url, \bot, referrer, referrerPolicy, s')$
    case $\langle \texttt{IFRAME}, url, window \rangle$
      let $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
      let $req := \langle \texttt{HTTPReq}, \nu_4, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, \langle\rangle, url.\texttt{parameters}, \langle\rangle \rangle$
      let $referrer := s'.\overline{w}'.\texttt{activedocument.location}$
      let $referrerPolicy := s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
      let $w' := \langle \nu_5, \langle\rangle, \bot \rangle$
      let $s'.\overline{w}'.\texttt{activedocument.subwindows}$
      ↪ $:= s'.\overline{w}'.\texttt{activedocument.subwindows} +^{\langle\rangle} w'$
      call $\mathsf{HTTP\_SEND}(\langle \texttt{REQ}, \nu_5 \rangle, req, url, \bot, referrer, referrerPolicy, s')$
    case $\langle \texttt{FORM}, url, method, data, hrefwindow \rangle$
      if $method \notin \{\texttt{GET}, \texttt{POST}\}$ **then**
        stop $\langle\rangle, s'$
      let $\overline{w}' := \mathsf{GETNAVIGABLEWINDOW}(\overline{w}, hrefwindow, \bot, s')$
      if $method = \texttt{GET}$ **then**
        let $body := \langle\rangle$
        let $parameters := data$
        let $origin := \bot$
      else
        let $body := data$
        let $parameters := url.\texttt{parameters}$
        let $origin := s'.\overline{d}.\texttt{origin}$
      let $req := \langle \texttt{HTTPReq}, \nu_4, method, url.\texttt{host}, url.\texttt{path}, \langle\rangle, parameters, body \rangle$
      let $referrer := s'.\overline{d}.\texttt{location}$
      let $referrerPolicy := s'.\overline{d}.\texttt{headers}[\texttt{ReferrerPolicy}]$
      let $s' := \mathsf{CANCELNAV}(s'.\overline{w}'.\texttt{nonce}, s')$
      call $\mathsf{HTTP\_SEND}(\langle \texttt{REQ}, s'.\overline{w}'.\texttt{nonce} \rangle, req, url, origin, referrer, referrerPolicy, s')$
    case $\langle \texttt{SETSCRIPT}, window, script \rangle$
      let $\overline{w}' := \mathsf{GETWINDOW}(\overline{w}, window, s')$
      let $s'.\overline{w}'.\texttt{activedocument.script} := script$
      stop $\langle\rangle, s'$

```
54:         case ⟨SETSCRIPTSTATE, window, scriptstate⟩
55:             let w̄′ := GETWINDOW(w̄, window, s′)
56:             let s′.w̄′.activedocument.scriptstate := scriptstate
57:             stop ⟨⟩, s′
58:         case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
59:             if method ∈ {CONNECT, TRACE, TRACK} ∧ xhrreference ∉ {𝒩, ⊥} then
60:                 stop ⟨⟩, s′
61:             if url.host ≢ s′.d̄.origin.host
                    ↪ ∨ url ≢ s′.d̄.origin.protocol then
62:                 stop ⟨⟩, s′
63:             if method ∈ {GET, HEAD} then
64:                 let data := ⟨⟩
65:                 let origin := ⊥
66:             else
67:                 let origin := s′.d̄.origin
68:             let req := ⟨HTTPReq, ν4, method, url.host, url.path, , url.parameters, data⟩
69:             let referrer := s′.d̄.location
70:             let referrerPolicy := s′.d̄.headers[ReferrerPolicy]
71:             call HTTP_SEND(⟨XHR, s′.d̄.nonce, xhrreference⟩, req, url, origin, referrer, referrerPolicy, s′)
72:         case ⟨BACK, window⟩
73:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
74:             call NAVBACK(w̄, s′)
75:         case ⟨FORWARD, window⟩
76:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
77:             call NAVFORWARD(w̄, s′)
78:         case ⟨CLOSE, window⟩
79:             let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
80:             remove s′.w̄′ from the sequence containing it
81:             stop ⟨⟩, s′
82:         case ⟨POSTMESSAGE, window, message, origin⟩
83:             if ∃w̄′ ∈ Subwindows(s′) such that s′.w̄′.nonce ≡ window then
84:                 let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
85:                 if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
                        ↪ ∧ (origin ≢ ⊥ ⟹ s′.w̄′.documents.j̄.origin ≡ origin) then
86:                     let s′.w̄′.documents.j̄.scriptinputs
                            ↪ := s′.w̄′.documents.j̄.scriptinputs
                            ↪ +^⟨⟩ ⟨POSTMESSAGE, s′.w̄.nonce, message, s′.d̄.origin⟩
87:             else
88:                 if ∃j̄ ∈ ℕ such that s′.serviceWorkers.j̄.nonce ≡ window
                        ↪ ∧ (origin ≢ ⊥ ⟹ s′.serviceWorkers.j̄.origin ≡ origin) then
89:                     let s′.serviceWorkers.j̄.scriptinputs := s′.serviceWorkers.j̄.scriptinputs
                            ↪ +^⟨⟩ ⟨POSTMESSAGE, s′.w̄.nonce, message, s′.d̄.origin⟩
90:             stop ⟨⟩, s′
                ——— Extension with Payment APIs ———
91:         case ⟨PR_CREATE, methodData, details, options⟩
92:             if ¬(methodData ∈ MethodDatas) then stop ⟨⟩, s′  → All PMIs in methodData must be equal (Def. 55)
93:             if methodData = ⟨⟩ then stop ⟨⟩, s′  → methodData should not be empty (see Section 3.1 of [14])
94:             let seenPMIs := ⟨⟩
95:             for each ⟨pmi, receiver, paymentIdentifier⟩ ∈ methodData do
96:                 if pmi ∈ seenPMIs then stop ⟨⟩, s′  → Ambiguous method data: Forbidden after spec fix
97:                 let seenPMIs := seenPMIs +^⟨⟩ pmi
98:             let paymentRequest := ⟨PAYMENTREQUEST, ν14, s′.d̄.nonce, methodData, details, options,
                        ↪ ⟨⟩, CR, ⊥, ⟨⟩⟩
99:             let transactionId := ν18  → Create identifier for transaction (this is a modeling artifact)
100:            let s′.paymentStorage[ν14] := ⟨paymentRequest, ⟨⟩, ⟨⟩, transactionId⟩
                    ↪  → Empty sequences for paymentRequestEvents and paymentResponses
101:            let s′.d̄.scriptinputs := s′.d̄.scriptinputs +^⟨⟩ paymentRequest
102:            stop ⟨⟩, s′
103:        case ⟨PR_SHOW, paymentRequestNonce, detailsUpdate⟩
104:            let paymentRequest := s′.paymentStorage[paymentRequestNonce].paymentRequest
105:            if paymentRequest.state ≠ CR then stop ⟨⟩, s′
106:            if s′.w̄.paymentRequestShowing = ⊤ then  → Can't show two payment requests at once
107:                let s′.paymentStorage[paymentRequestNonce].paymentRequest.state := CL
108:                stop ⟨⟩, s′
```

case continued on the next page

```
109:            let s'.paymentStorage[paymentRequestNonce].paymentRequest.state := IN
110:            let s'.w̄.paymentRequestShowing := ⊤
111:            let handlers := ⟨⟩
112:            for each mds := ⟨pmi, receiver, paymentIdentifier⟩ ∈ paymentRequest.methodData do
113:                let handlersForPmi := GET_PAYMENT_HANDLERS(pmi, s')
114:                let handlers := handlers +^⟨⟩ handlersForPmi
115:                for each handler ∈ handlersForPmi do
116:                    let s'.events := s'.events +^⟨⟩ ⟨CANMAKEPAYMENT, handler.nonce,
                            ↪ tlw.origin, s'.d̄.origin, mds⟩
117:            let handler ← handlers   → Customer selects a payment handler
118:            let s'.paymentStorage[paymentRequestNonce].handlerNonce := handler.nonce
119:            if detailsUpdate ≠ ⟨⟩ then
120:                let s'.paymentStorage[paymentRequestNonce].paymentRequest.updating := ⊤
121:                let s'.events := s'.events +^⟨⟩ ⟨PR_UPDATE_DETAILS, paymentRequestNonce, detailsUpdate⟩
122:            let s'.events := s'.events +^⟨⟩ ⟨SUBMITPAYMENT, paymentRequestNonce, handler.nonce⟩
123:            stop ⟨⟩, s'
124:        case ⟨PR_CANMAKEPAYMENT, paymentRequestNonce⟩
125:            let paymentRequest := s'.paymentStorage[paymentRequestNonce].paymentRequest
126:            if paymentRequest.state ≠ CR then stop ⟨⟩, s'
127:            let handlers := ⟨⟩
128:            for each ⟨pmi, receiver, paymentIdentifier⟩ ∈ paymentRequest.methodData do
129:                let handlers := handlers +^⟨⟩ GET_PAYMENT_HANDLERS(pmi, s')
130:            if handlers ≠ ⟨⟩ then
131:                let s'.d̄.scriptinputs := s'.d̄.scriptinputs
                    ↪ +^⟨⟩ ⟨CANMAKEPAYMENTRESPONSE, paymentRequestNonce, ⊤⟩
                        → Let merchant script know that payment can be made
132:            else
133:                let s'.d̄.scriptinputs := s'.d̄.scriptinputs
                    ↪ +^⟨⟩ ⟨CANMAKEPAYMENTRESPONSE, paymentRequestNonce, ⊥⟩
                        → Let merchant script know that payment cannot be made
134:            stop ⟨⟩, s'
135:        case ⟨PR_ABORT, paymentRequestNonce⟩
136:            let paymentRequest := s'.paymentStorage[paymentRequestNonce].paymentRequest
137:            if paymentRequest.state ≠ IN then stop ⟨⟩, s'
138:            if s'.paymentStorage[paymentRequestNonce].paymentResponses ≠ ⟨⟩ then stop ⟨⟩, s'
139:            let s'.paymentStorage[paymentRequestNonce].paymentRequest.state := CL
140:            let s'.w̄.paymentRequestShowing := ⊥
141:            stop ⟨⟩, s'
142:        case ⟨PRES_COMPLETE, paymentRequestNonce, paymentResponseNonce⟩
143:            let paymentResponse := (s'.paymentStorage[paymentRequestNonce]
                    ↪ .paymentResponses)[paymentResponseNonce]   → Retrieve payment response
144:            if paymentResponse.complete ≡ ⊤ then stop ⟨⟩, s'
145:            let (s'.paymentStorage[paymentRequestNonce].paymentResponses)[paymentResponseNonce]
                    ↪ .complete := ⊤
146:            let s'.w̄.paymentRequestShowing := ⊥
147:            stop ⟨⟩, s'
148:        case ⟨PRES_RETRY, paymentRequestNonce, paymentResponseNonce, errorFields⟩
149:            let paymentResponse := (s'.paymentStorage[paymentRequestNonce]
                    ↪ .paymentResponses)[paymentResponseNonce]   → Retrieve payment response
150:            if paymentResponse.complete ≡ ⊤ then stop ⟨⟩, s'
151:            let s'.paymentStorage[paymentRequestNonce].paymentRequest.state := IN
152:            let handlerNonce := s'.paymentStorage[paymentRequestNonce].handlerNonce
                    → Fix: No change of payment handler during retry
153:            let s'.events := s'.events +^⟨⟩ ⟨SUBMITPAYMENT, paymentRequestNonce, handlerNonce⟩
154:            stop ⟨⟩, s'
155:        case ⟨PR_GET_PREQ, paymentRequestNonce⟩   → (Merchant) script can retrieve payment request object
156:            let paymentRequest := s'.paymentStorage[paymentRequestNonce].paymentRequest
157:            let s'.d̄.scriptinputs := s'.d̄.scriptinputs +^⟨⟩ paymentRequest
158:            stop ⟨⟩, s'
```

```
159:      case ⟨PR_GET_PRESP, paymentRequestNonce, paymentResponseNonce⟩  → Analogous for responses
160:          let responses := s′.paymentStorage[paymentRequestNonce].paymentResponses
161:          let paymentResponse := responses[paymentResponseNonce]
162:          let s′.d̄.scriptinputs := s′.d̄.scriptinputs +^⟨⟩ paymentResponse
163:          stop ⟨⟩, s′
164:      case ⟨PR_UPDATE_DETAILS, paymentRequestNonce, details⟩  → According to spec, updateDetails can
            only occur as a reaction to a PaymentRequestUpdateEvent. This is simplified in the model.
165:          let paymentRequest := s′.paymentStorage[paymentRequestNonce].paymentRequest
166:          if paymentRequest.state ≠ IN then stop ⟨⟩, s′
167:          let s′.paymentStorage[paymentRequestNonce].paymentRequest.updating := ⊤
168:          let s′.events := s′.events +^⟨⟩ ⟨PR_UPDATE_DETAILS, paymentRequestNonce, details⟩
169:          stop ⟨⟩, s′
170:      case else stop ⟨⟩, s′
```

E. Handle DOM Events

Algorithm 22 shows how the function PROCESSEVENT models the browser's DOM event processing. The events CANMAKEPAYMENT, PAYMENTREQUESTEVENT, and PAYMENTRESPONSE are simply processed by transmitting the relevant event to the corresponding party.

PAYMENTHANDLERRESPONSE is more complex since it integrates the information provided by the payment handler through the PaymentHandlerResponse into a PaymentResponse object and submits it to the event set for further processing.

Algorithm 22 Web Browser Model: Process an event.

```
1: function PROCESSEVENT(e, w̄, s′)
2:    switch e do
                          pre-flight request whether payment can be made
3:      case ⟨CANMAKEPAYMENT, handlerNonce, topOrigin, paymentRequestOrigin, methodData⟩
4:          call DELIVER_TO_DOC(handlerNonce, e, s′)
              → The browser non-deterministically chooses a payment handler (modeling the customer's
                selection), thus, we can safely over-approximate here and do not have to wait for the
                reply to our CANMAKEPAYMENT (we over-approximate in that a payment handler can
                reply "no" to CANMAKEPAYMENT and still get selected).
                          User accepts the payment request
5:      case ⟨SUBMITPAYMENT, paymentRequestNonce, handlerNonce⟩
6:          let paymentRequest := s.paymentStorage[paymentRequestNonce].paymentRequest
7:          if paymentRequest.updating = ⊤ then stop ⟨⟩, s′
8:          if paymentRequest.state ≠ IN then stop ⟨⟩, s′
9:          let handler ← s′.serviceWorkers such that handler.nonce ≡ handlerNonce
              ↪ if possible; otherwise stop ⟨⟩, s′
10:         let total := paymentRequest.details.total
11:         let modifiers := paymentRequest.details.modifiers  → Abstraction: pass all modifiers
12:         let requestBillingAddress := paymentRequest.options.requestBillingAddress
13:         let instrument ← handler.paymentManager such that ∃mds ∈^⟨⟩ paymentRequest.methodData ∧
              ↪ mds.pmi = instrument.enabledMethods if possible; otherwise stop ⟨⟩, s′  → Choose an
              ↪ instrument that supports one of the payment methods.
14:         let instrumentKey := instrument.instrumentKey
15:         let methodData := ⟨⟩
16:         for each mds := ⟨pmi, receiver, paymentIdentifier⟩ ∈ paymentRequest.methodData do
17:             if pmi ≡ instrument.enabledMethods then  → enabledMethods is a single value (cf. Def. 56)
18:                 let methodData := methodData +^⟨⟩ mds
19:         let transactionId := s′.paymentStorage[paymentRequestNonce].transactionId
20:         let pre := ⟨PAYMENTREQUESTEVENT, ν16, paymentRequestNonce, handler.nonce, methodData,
              ↪ total, modifiers, instrumentKey, requestBillingAddress, transactionId⟩
21:         let (s′.paymentStorage[paymentRequestNonce].paymentRequestEvents)[ν16] := pre
22:         let s′.events := s′.events +^⟨⟩ pre
23:         let paymentIntent := paymentRequest
24:         let paymentIntent.methodData := methodData
25:         let s′.paymentIntents[transactionId] := s′.paymentIntents[transactionId] +^⟨⟩ paymentIntent
26:         stop ⟨⟩, s′
```

```
PAYMENTREQUESTEVENT is given to payment handler to process it
case ⟨PAYMENTREQUESTEVENT, paymentRequestEventNonce, paymentRequestNonce, handlerNonce,
    ↪ methodData, total, modifiers, instrumentKey, requestBillingAddress, transactionId⟩
  call DELIVER_TO_DOC(handlerNonce, e, s′)
Payment handler's response is merged with relevant data from payment UI
case ⟨PAYMENTHANDLERRESPONSE, paymentRequestEventNonce, paymentRequestNonce,
    ↪ handlerNonce, methodName, details⟩
  let paymentRequest := s′.paymentStorage[paymentRequestNonce].paymentRequest
  let paymentRequestEvent := s′.paymentStorage[paymentRequestNonce]
    ↪ .paymentRequestEvents[paymentRequestEventNonce]
  if paymentRequestEvent.methodData | ⟨methodName, ∗, ∗⟩ ≡ ⟨⟩ then
    stop ⟨⟩, s′   → Selected method not accepted by merchant
  if paymentRequest.updating ≡ ⊤ then stop ⟨⟩, s′
  if paymentRequest.state ≢ IN then stop ⟨⟩, s′
  let shippingAddress := ⟨⟩
  let shippingOption := ⟨⟩
  if paymentRequest.options.requestShipping ≡ ⊤ then
    let shippingAddress := ν17
    let shippingOption ← paymentRequest.details.shippingOptions
  let payerInfo := ⟨⟩
  if paymentRequest.options.requestPayerInfo ≡ ⊤ then
    let payerInfo := ν15   → payer info (phone, name, email)
  let respNonce := ν16
  let response := ⟨PAYMENTRESPONSE, respNonce, paymentRequestNonce, handlerNonce,
      ↪ methodName, details, shippingAddress, shippingOption, payerInfo, ⊥⟩
  let (s′.paymentStorage[paymentRequestNonce].paymentResponses)[respNonce] := response
  let s′.events := s′.events +^⟨⟩ response   → Create PAYMENTRESPONSE Event
  stop ⟨⟩, s′
Payment Response is forwarded to script in user agent
case ⟨PAYMENTRESPONSE, responseNonce, paymentRequestNonce, handlerNonce, methodName,
    ↪ details, shippingAddress, shippingOption, payerInfo, complete⟩
  let requestingWindowNonce
      ↪ := s′.paymentStorage.[paymentRequestNonce].paymentRequest.documentnonce
  call DELIVER_TO_DOC(requestingWindowNonce, e, s′)
Update payment details
case ⟨PR_UPDATE_DETAILS, paymentRequestNonce, details⟩
  let s′.paymentStorage[paymentRequestNonce].paymentRequest.details := details
  let s′.paymentStorage[paymentRequestNonce].paymentRequest.updating := ⊥
  stop ⟨⟩, s′
```

F. Service Workers

Algorithm 23 with its function RUNWORKER models how service workers are executed. The algorithm is very similar to Algorithm 21 with its function RUNSCRIPT. The biggest difference lies within the non-existence of a relevant document and the available command set.

The six commands that are available are PAYMENTHANDLERRESPONSE, SET_PAYMENTMANAGER, GET_PAYMENTMANAGER, XMLHTTPREQUEST, POSTMESSAGE, and OPEN_WINDOW.

Their basic functionalities are the following:

PAYMENTHANDLERRESPONSE is called by a script to submit a PaymentHandlerResponse to the payment issuer. By doing so, it signals the completion of its processing of the PaymentRequestEvent.

SET_PAYMENTMANAGER A service worker can define which payment instruments it supports. This command is used to signal updates of this set to the browser.

GET_PAYMENTMANAGER A service worker can get a list of its supported payment instruments. This command offers such functionality.

XMLHTTPREQUEST as in RUNSCRIPT.

POSTMESSAGE posts a message to a window or another service worker via postMessage.

OPEN_WINDOW allows a payment handler to open a window for further user interaction.

Algorithm 23 Web Browser Model: Execute a Service Worker.

function RUNWORKER($\overline{sw}, s'$) → Pointer into browser's list of service workers, and browser state
  let $sw := s'.\texttt{serviceWorkers}.\overline{sw}$
  let $swOrigin := \langle sw.\texttt{scope.host}, sw.\texttt{scope.protocol}\rangle$
  let $cookies := \langle \{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\,[sw.\texttt{scope.host}]$
    ↪ $\wedge\, c.\texttt{content.httpOnly} = \bot \wedge (c.\texttt{content.secure} \implies (sw.\texttt{scope.protocol} \equiv \texttt{S}))\}\rangle$
  let $localStorage := s'.\texttt{localStorage}\,[swOrigin]$
  let $secrets := s'.\texttt{secrets}\,[swOrigin]$
  let $R \leftarrow \mathsf{script}^{-1}(sw.\texttt{script})$
  let $in := \langle sw.\texttt{nonce}, sw.\texttt{scriptstate}, sw.\texttt{scriptinputs}, cookies, localStorage, secrets, swOrigin\rangle$
  let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
    ↪ $cookies' \leftarrow \mathsf{Cookies}^{\nu}$,
    ↪ $localStorage' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
    ↪ $command \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
    ↪ $out^{\lambda} := \langle state', cookies', localStorage', command\rangle$
    ↪ **such that** $(in, out^{\lambda}) \in R$
  let $s'.\texttt{cookies}\,[sw.\texttt{scope.host}] := \mathsf{CookieMerge}(s'.\texttt{cookies}\,[sw.\texttt{scope.host}],\ cookies')$
  let $s'.\texttt{localStorage}\,[swOrigin] := localStorage'$
  let $s'.\texttt{serviceWorkers}.\overline{sw}.\texttt{scriptstate} := state'$
  switch *command* **do**
    case $\langle \texttt{PAYMENTHANDLERRESPONSE}, paymentRequestEventNonce,$
      ↪ $paymentRequestNonce, handlerNonce, methodName, details\rangle$
      let $\overline{pre} \leftarrow \mathbb{N}$ **such that** $\pi_1(sw.\texttt{scriptinputs}.\overline{pre}) = \texttt{PAYMENTREQUESTEVENT}$
        ↪ $\wedge\, sw.\texttt{scriptinputs}.\overline{pre}.\texttt{paymentRequestNonce} \equiv paymentRequestNonce$
        ↪ **if possible; otherwise stop** $\langle\rangle, s'$
      let $s'.\texttt{events} := s'.\texttt{events} +^{\langle\rangle} \langle \texttt{PAYMENTHANDLERRESPONSE}, paymentRequestEventNonce,$
        ↪ $paymentRequestNonce, handlerNonce, methodName, details\rangle$
      stop $\langle\rangle, s'$
    case $\langle \texttt{SET\_PAYMENTMANAGER}, newValue\rangle$
      let $s'.\texttt{serviceWorkers}.\overline{sw}.\texttt{paymentManager} := newValue$
      stop $\langle\rangle, s'$
    case $\langle \texttt{GET\_PAYMENTMANAGER}\rangle$
      let $s'.\texttt{serviceWorkers}.\overline{sw}.\texttt{scriptinputs} := s'.\texttt{serviceWorkers}.\overline{sw}.\texttt{paymentManager}$
      stop $\langle\rangle, s'$
    case $\langle \texttt{XMLHTTPREQUEST}, url, method, data, xhrreference\rangle$
      if $method \in \{\texttt{CONNECT}, \texttt{TRACE}, \texttt{TRACK}\} \wedge xhrreference \notin \{\mathcal{N}, \bot\}$ **then stop**
      if $url.\texttt{host} \not\equiv swOrigin.\texttt{host} \vee url.\texttt{protocol} \not\equiv swOrigin.\texttt{protocol}$ **then stop**
      if $method \in \{\texttt{GET}, \texttt{HEAD}\}$ **then**
        let $data := \langle\rangle$
        let $origin := \bot$
      else
        let $origin := swOrigin$
      let $req := \langle \texttt{HTTPReq}, \nu_4, method, url.\texttt{host}, url.\texttt{path}, , url.\texttt{parameters}, data\rangle$
      let $referrer := s'.\texttt{serviceWorkers}.\overline{sw}.\texttt{scope}$
      let $referrerPolicy := \texttt{noreferer}$
      call HTTP_SEND($\langle \texttt{XHR}, sw.\texttt{nonce}, xhrreference\rangle, req, url, origin, referrer, referrerPolicy, s'$)
    case $\langle \texttt{POSTMESSAGE}, window, message, origin\rangle$
      if $\exists \overline{w} \in \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{nonce} \equiv window$ **then**
        let $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{nonce} \equiv window$
        if $\exists \overline{j} \in \mathbb{N}$ **such that** $s'.\overline{w}.\texttt{documents}.\overline{j}.\texttt{active} \equiv \top$
          ↪ $\wedge\, (origin \not\equiv \bot \implies s'.\overline{w}.\texttt{documents}.\overline{j}.\texttt{origin} \equiv origin)$ **then**
          let $s'.\overline{w}.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
            ↪ $:= s'.\overline{w}.\texttt{documents}.\overline{j}.\texttt{scriptinputs}$
            ↪ $+^{\langle\rangle} \langle \texttt{POSTMESSAGE}, s'.\overline{w}.\texttt{nonce}, message, s'.\overline{d}.\texttt{origin}\rangle$
      else
        if $\exists \overline{j} \in \mathbb{N}$ **such that** $s'.\texttt{serviceWorkers}.\overline{j}.\texttt{nonce} \equiv window$ **then**
          let $s'.\texttt{serviceWorkers}.\overline{j}.\texttt{scriptinputs} := s'.\texttt{serviceWorkers}.\overline{j}.\texttt{scriptinputs}$
            ↪ $+^{\langle\rangle} \langle \texttt{POSTMESSAGE}, window, s'.\overline{w}.\texttt{nonce}, \langle\rangle\rangle$
      stop $\langle\rangle, s'$
    case $\langle \texttt{OPEN\_WINDOW}, windownonce, url\rangle$ → Allow payment handler to open window
      let $w' := \langle windownonce, \langle\rangle, \bot\rangle$
      let $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
      let $req := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, url.\texttt{host}, url.\texttt{path}, url.\texttt{parameters}, \langle\rangle, \langle\rangle\rangle$
      call HTTP_SEND($\langle \texttt{REQ}, windownonce\rangle, req, url, \bot, \bot, \bot, s'$)
    case else **stop**

G. Payment Handlers

To model a payment handler, Algorithm 24 was designed. It models a payment handler that, upon receiving a PaymentRequestEvent, opens a window through which it obtains an authentication token (i.e., the customer authenticates herself). With this token it can issue a payment and craft a payment handler response afterwards.

Algorithm 24 Relation of *script_default_payment_handler*

Input: $\langle handlerNonce, scriptstate, scriptinputs, cookies, localStorage, secrets, swOrigin\rangle$ → **Script that models the behavior of a payment handler.** Reacts to payment request event and communicates to a given endpoint of the payment service. Crafts a payment handler response.

1: **let** $switch \leftarrow \{\texttt{request}, \texttt{pay}, \texttt{craft}, \texttt{xmlhttp}, \texttt{setmgr}, \texttt{getmgr}\}$
2: **if** $switch \equiv \texttt{request}$ **then** → **Start to process payment request event (authN at payment provider)**
3: **let** $\overline{pre'} \leftarrow \mathbb{N}$ **such that** $\pi_1(scriptinputs.\overline{pre'}) = \texttt{PAYMENTREQUESTEVENT}$ **if possible; otherwise stop** $\langle scriptstate, cookies, localStorage, \langle\rangle\rangle$ → Choose a random payment request event to process
4: **let** $url := \langle \texttt{URL}, S, swOrigin.\texttt{host}, /\texttt{index}, \langle\rangle, \langle\rangle\rangle$
5: **let** $command := \langle \texttt{OPEN\_WINDOW}, \nu_{18}, url\rangle$ → Open window for customer authentication
6: **stop** $\langle scriptstate, cookies, localStorage, command\rangle$
7: **else if** $switch \equiv \texttt{pay}$ **then** → **Send payment details to payment provider**
8: **let** $\overline{pre'} \leftarrow \mathbb{N}$ **such that** $\pi_1(scriptinputs.\overline{pre'}) = \texttt{PAYMENTREQUESTEVENT}$ **if possible; otherwise stop** $\langle scriptstate, cookies, localStorage, \langle\rangle\rangle$
9: **let** $url := \langle \texttt{URL}, S, swOrigin.\texttt{host}, /\texttt{pay}, \langle\rangle, \langle\rangle\rangle$
10: **let** $total := scriptinputs.\overline{pre'}.\texttt{total}$
11: **let** $tokenMessage$ **such that** $\pi_1(tokenMessage) = \texttt{POSTMESSAGE} \wedge tokenMessage \in scriptinputs$ **if possible; otherwise stop** $\langle scriptstate, cookies, localStorage, \langle\rangle\rangle$
12: **let** $token := \pi_3(tokenMessage)$
13: **let** $methodData \leftarrow scriptinputs.\overline{pre'}.\texttt{methodData}$ → Choose a methodData entry.
14: **let** $receiver := methodData.\texttt{receiver}$
15: **let** $paymentIdentifier := methodData.\texttt{paymentIdentifier}$
16: **let** $paymentRequestNonce := scriptinputs.\overline{pre'}.\texttt{paymentRequestNonce}$
17: **let** $transactionId := scriptinputs.\overline{pre'}.\texttt{transactionId}$
18: **let** $command := \langle \texttt{XMLHTTPREQUEST}, url, \texttt{POST}, \langle token, receiver, total, paymentIdentifier, transactionId\rangle, \bot\rangle$
19: **stop** $\langle scriptstate, cookies, localStorage, command\rangle$
20: **else if** $switch \equiv \texttt{craft}$ **then** → **Craft payment handler response**
21: **let** $\overline{pre'} \leftarrow \mathbb{N}$ **such that** $\pi_1(scriptinputs.\overline{pre'}) = \texttt{PAYMENTREQUESTEVENT}$ **if possible; otherwise stop** $\langle scriptstate, cookies, localStorage, \langle\rangle\rangle$
22: **let** $methodName \leftarrow \mathcal{T}_{\mathcal{N}}$
23: **let** $details \leftarrow \mathcal{T}_{\mathcal{N}}$
24: **let** $command := \langle \texttt{PAYMENTHANDLERRESPONSE}, scriptinputs.\overline{pre'}.\texttt{paymentRequestEventNonce}, scriptinputs.\overline{pre'}.\texttt{paymentRequestNonce}, handlerNonce, methodName, details\rangle$
25: **stop** $\langle scriptstate, cookies, localStorage, command\rangle$
26: **else if** $switch \equiv \texttt{setmgr}$ **then** → Set Payment Manager
27: **let** $newValue \leftarrow \mathcal{T}_{\mathcal{N}}$
28: **let** $command := \langle \texttt{SET\_PAYMENTMANAGER}, newValue\rangle$
29: **stop** $\langle scriptstate, cookies, localStorage, command\rangle$
30: **else if** $switch \equiv \texttt{getmgr}$ **then** → Get Payment Manager
31: **let** $command := \langle \texttt{GET\_PAYMENTMANAGER}\rangle$
32: **stop** $\langle scriptstate, cookies, localStorage, command\rangle$
33: **else if** $switch \equiv \texttt{xmlhttp}$ **then** → Perform XMLHTTPRequest
34: **let** $protocol \leftarrow \{\texttt{P}, \texttt{S}\}$
35: **let** $host \leftarrow \mathsf{Doms}$
36: **let** $path \leftarrow \mathbb{S}$
37: **let** $fragment \leftarrow \mathbb{S}$
38: **let** $parameters \leftarrow [\mathbb{S} \times \mathbb{S}]$
39: **let** $url := \langle \texttt{URL}, protocol, host, path, parameters, fragment\rangle$
40: **let** $command := \langle \texttt{XMLHTTPREQUEST}, url, \texttt{GET}, \langle\rangle, \bot\rangle$
41: **stop** $\langle scriptstate, cookies, localStorage, command\rangle$

Helper Funtions. Algorithm 25 defines a helper function `GET_PAYMENT_HANDLERS` that returns all relevant payment handlers for a given payment method identifier.

The function `DELIVER_TO_DOC` in Algorithm 26 is extended to be able to deliver messages as well to service workers.

Documents of the same scope as to which a service worker is registered have a reference to these service workers. This is provided through extending Algorithm 28 by a few lines and using the new Algorithm 27 with its function `GET_SWS`. It returns a set of relevant service workers for a given URL.

Algorithm 25 Web Browser Model: Process script interaction with Payment APIs

```
function GET_PAYMENT_HANDLERS(pmi, s')
    let handlers := {}
    for each handler ∈ s'.serviceWorkers do
        for each instrument ∈ handler.paymentManager.instruments do
            if instrument.method ≡ pmi then
                let handlers := handlers ∪ {handler}
    return handlers
```

Algorithm 26 Web Browser Model: Deliver a message to the script in a document or a service worker.

```
function DELIVER_TO_DOC(nonce, data, s')
    if ∃sw ∈ s'.serviceWorkers such that sw.nonce ≡ nonce then
        let sw̄ ← ℕ such that s'.serviceWorkers.sw̄.nonce ≡ nonce
        let s'.serviceWorkers.sw̄.scriptinputs := s'.serviceWorkers.sw̄.scriptinputs +^⟨⟩ data
        stop ⟨⟩, s'
    else
        let w̄ ← Subwindows(s'), d̄ such that s'.d̄.nonce ≡ nonce ∧ s'.d̄ = s'.w̄.activedocument
          ↪ if possible; otherwise stop ⟨⟩, s'
        let s'.d̄.scriptinputs := s'.d̄.scriptinputs +^⟨⟩ data
        stop ⟨⟩, s'
```

Algorithm 27 Web Browser Model: Get relevant service workers for URL

```
function GET_SWS(url, s')
    let handlers := {}
    for each handler ∈ s'.serviceWorkers do
        if handler.scope.domain = url.domain ∧ handler.scope.protocol = url.protocol then
            let handlers := handlers ∪ {handler}
    return handlers
```

Algorithm 28 Web Browser Model: Process an HTTP response.

```
function PROCESSRESPONSE(response, reference, request, requestUrl, key, f, s')
                                        Process headers in response
    if Set-Cookie ∈ response.headers then
        for each c ∈^⟨⟩ response.headers[Set-Cookie], c ∈ Cookies do
            let s'.cookies[request.host] := AddCookie(s'.cookies[request.host], c)
    if Strict-Transport-Security ∈ response.headers ∧ requestUrl.protocol ≡ S then
        let s'.sts := s'.sts +^⟨⟩ request.host
    if Referer ∈ request.headers then
        let referrer := request.headers[Referer]
    else
        let referrer := ⊥
    if Location ∈ response.headers ∧ response.status ∈ {303, 307} then
        let url := response.headers[Location]
        if url.fragment ≡ ⊥ then
            let url.fragment := requestUrl.fragment
        let method' := request.method
        let body' := request.body
        if response.status ≡ 303 ∧ request.method ∉ {GET, HEAD} then
            let method' := GET
            let body' := ⟨⟩
        if Origin ∈ request.headers ∧ method' ≡ POST then
            let origin := ◇
        else
            let origin := ⊥
        if π1(reference) ≢ XHR then   → Do not redirect XHRs.
            let req := ⟨HTTPReq, ν6, method', url.host, url.path, url.parameters, ⟨⟩, body'⟩
            let referrerPolicy := response.headers[ReferrerPolicy]
            call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, s')
```

Deliver/process data in response

28: **switch** $\pi_1(reference)$ **do**
29: **case** `REQ` → normal response
30: **let** $\overline{w} \leftarrow$ Subwindows(s') **such that** $s'.\overline{w}.\texttt{nonce} \equiv \pi_2(reference)$ ↪ **if possible; otherwise stop**
31: **if** $response.\texttt{body} \not\sim \langle *, * \rangle$ **then**
32: **stop** $\{\}$, s'
33: **let** $script := \pi_1(response.\texttt{body})$
34: **let** $scriptinputs := \pi_2(response.\texttt{body}) +^{\langle\rangle}$ GET_SWS($requestUrl, s'$) → Payment Handler Extension
35: **let** $d := \langle \nu_7, requestUrl, response.\texttt{headers}, referrer, script, \langle\rangle, scriptinputs, \langle\rangle, \top \rangle$
36: **if** $s'.\overline{w}.\texttt{documents} \equiv \langle\rangle$ **then**
37: **let** $s'.\overline{w}.\texttt{documents} := \langle d \rangle$
38: **else**
39: **let** $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}.\texttt{documents}.\overline{i}.\texttt{active} \equiv \top$
40: **let** $s'.\overline{w}.\texttt{documents}.\overline{i}.\texttt{active} := \bot$
41: **remove** $s'.\overline{w}.\texttt{documents}.(\overline{i}+1)$ and all following documents from $s'.\overline{w}.\texttt{documents}$
42: **let** $s'.\overline{w}.\texttt{documents} := s'.\overline{w}.\texttt{documents} +^{\langle\rangle} d$
43: **stop** $\{\}$, s'
44: **case** `XHR` → process XHR response
45: **let** $headers := response.\texttt{headers} - \texttt{Set-Cookie}$
46: **let** $m := \langle \texttt{XMLHTTPREQUEST}, headers, response.\texttt{body}, \pi_3 reference \rangle$
47: **call** DELIVER_TO_DOC($\pi_2(reference)$, m, s')
48: **stop** $\{\}$, s'
49: **case** `WS` → process WebSocket response
50: **if** $response.\texttt{status} \not\equiv 101 \vee response.\texttt{headers}[\texttt{Upgrade}] \not\equiv \texttt{websocket}$ **then**
51: **stop**
52: **let** $wsconn := \langle reference, request.\texttt{nonce}, key, f \rangle$
53: **let** $s'.\texttt{wsConnections} := s'.\texttt{wsConnections} +^{\langle\rangle} wsconn$

H. Payment Provider Server

Our generic payment provider server is based on the template for HTTPS servers offered in the WIM [21].

Definition 70. The set of possible states Z^{pp} of a payment provider server pp is the set of terms of the form:

$$\langle DNSaddress, pendingDNS, pendingRequests, corrupt, keyMapping, tlskeys, ids, \textbf{transactions}, \textbf{tokens} \rangle$$

Where $DNSaddress$, $pendingDNS$, $pendingRequests$, $corrupt$, $keyMapping$, $tlskeys$, and ids defined as in the Web Infrastructure Model [21].

The remaining sub terms have the following form:

- $transactions \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$ stores all transactions that have been submitted and accepted by the payment provider server. More precisely, for a term $key \in \mathcal{T}_{\mathcal{N}}$ (typically a payment identifier), the transaction $transactions[key]$ has the following structure: $\langle sender, receiver, total, transactionId \rangle$, where $sender, receiver, total \in \mathcal{T}_{\mathcal{N}}$, and $transactionId \in \mathcal{N}$.
- $tokens \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{N}]$ stores all tokens that have been authorized by the payment provider through basic authorization.

An initial state s_0^{pp} of pp is a state of pp with $s_0^{pp}.\texttt{pendingDNS} \equiv \langle\rangle$, $s_0^{pp}.\texttt{pendingRequests} \equiv \langle\rangle$, $s_0^{pp}.\texttt{corrupt} \equiv \bot$, $s_0^{pp}.\texttt{keyMapping}$ being the same as the key mapping for browsers [20], $s_0^{pp}.\texttt{tlskeys} \equiv \texttt{tlskeys}^{pp}$, $s_0^{pp}.\texttt{transactions} \equiv \langle\rangle$, and $s_0^{pp}.\texttt{tokens} \equiv \langle\rangle$.

Placeholder	Usage
ν_1	Algorithm 29, placeholder for auth token used by service worker

Table III List of placeholders used in payment provider server algorithms.

Table III shows the placeholder that was added to the generic HTTPS model offered in [21]. It is used to create authentication tokens with which a payment can be authenticated.

The template is modeled in Algorithm 29 with a basic interface offering the ability to process HTTPS requests to three different endpoints:

`/index` serves an entry index page with the `script_payment_provider_index` script.

`/authenticate` expects the customer's credentials as input and responds with an authentication token that can later be used to pay requests (if the credentials where correct).

`/pay` is used to process a transaction given a receiver, a total and an authentication token.

In the algorithm, transactions are stored within a dictionary. This allows for an easy implementation of the retry mechanism, since earlier requests for the same request id are simply overwritten.

Algorithm 29 Relation of a payment provider server

function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
  if m.`path` $\equiv$ `/index` **then**
    let *headers* $:= \langle\langle$`ReferrerPolicy`, `origin`$\rangle\rangle$
    let $m' := \mathsf{enc_s}(\langle$`HTTPResp`, m.`nonce`, 200, *headers*, $\langle$`script_payment_provider_index`, $\langle\rangle\rangle\rangle, k)$
  else if m.`path` $\equiv$ `/authenticate` **then**
    let *identity* $:= m$.`body`[`id`]
    let *password* $:= m$.`body`[`secret`]
    if *password* $\neq$ secretOfID(*identity*) **then**
      stop $\langle\rangle, s'$
    let *token* $:= \nu_1$
    let s'.`tokens`[*identity*] $:= s'$.`tokens`[*identity*] $+^{\langle\rangle}$ *token*
    let $m' := \mathsf{enc_s}(\langle$`HTTPResp`, m.`nonce`, 200, $\langle\rangle, \langle$`TOKEN`, *token*$\rangle\rangle, k)$
    stop $\langle\langle f, a, m'\rangle\rangle$, s'
  else if m.`path` $\equiv$ `/pay` $\wedge$ m.`method` $\equiv$ `POST` **then**
    let *token_seq* $:= m$.`body`[`token`] $\rightarrow \langle$`TOKEN`, *token*$\rangle$
    let *token* $:= \pi_2$(*token_seq*)
    let *sender* $\leftarrow s'$.`ids` **such that** *token* $\in^{\langle\rangle}$ s'.`tokens`[*sender*] $\hookrightarrow$ **if possible; otherwise stop** $\langle\rangle, s'$
    let *receiver* $:= m$.`body`[`receiver`]
    let *total* $:= m$.`body`[`total`]
    let *paymentIdentifier* $:= m$.`body`[`paymentIdentifier`]
    let *transactionId* $:= m$.`body`[`transactionId`]
    let s'.`transactions`[$\langle$*paymentIdentifier*$\rangle$] $:= \langle$*sender*, *receiver*, *total*, *transactionId*$\rangle$ $\rightarrow$ Seamlessly integrate retry updates
    let $m' := \mathsf{enc_s}(\langle$`HTTPResp`, m.`nonce`, 200, $\langle\rangle, \langle\rangle\rangle, k)$
    stop $\langle\langle f, a, m'\rangle\rangle$, s'

The `script_payment_provider_index` script is defined in Algorithm 30. It can only do two different things: First, it can obtain an authorization token by use of a `XMLHTTPREQUEST`. Second, it can submit such a token with a post request to the service worker waiting for it.

Algorithm 30 Relation of *script_payment_provider_index*.

Input: $\langle$*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*$\rangle$
let *switch* $\leftarrow$ {`auth`, `postToken`}
if *switch* $\equiv$ `auth` **then**
  let *url* $:=$ GETURL(*tree*, *docnonce*)
  let *id* $\leftarrow$ *ids*
  let *username* $:= \pi_1$(*id*)
  let *domain* $:= \pi_2$(*id*)
  let *interactive* $\leftarrow \{\bot, \top\}$
  let *url* $:= \langle$`URL`, `S`, *url*.`host`, `/authenticate`, $\langle\rangle, \langle\rangle\rangle$
  let *secret* $\leftarrow$ *secrets* **such that** *secret* $=$ secretOfID(*id*) **if possible; otherwise** $\hookrightarrow$ **stop** $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, $\langle\rangle\rangle$
  let *command* $:= \langle$`XMLHTTPREQUEST`, *url*, `POST`, $\langle$*id*, *secret*$\rangle, \bot\rangle$
  stop $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$
else
  let *url* $:=$ GETURL(*tree*, *docnonce*)
  let *origin* $:= \langle$*url*.`host`, `S`$\rangle$
  let *token* **such that** π_1(*token*) $=$ `TOKEN` $\wedge$ *token* $\in$ *scriptinputs* **if possible; otherwise** $\hookrightarrow$ **stop** $\langle s$, *cookies*, *localStorage*, *sessionStorage*, $\langle\rangle\rangle$
  let *swNonce* **such that** π_1(*swNonce*) $=$ `SWNONCE` $\wedge$ *swNonce* $\in$ *scriptinputs* **if possible; otherwise** $\hookrightarrow$ **stop** $\langle s$, *cookies*, *localStorage*, *sessionStorage*, $\langle\rangle\rangle$
  let *command* $:= \langle$`POSTMESSAGE`, *swNonce*, *token*, *origin*$\rangle$
  stop $\langle$*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*$\rangle$

I. Merchant Server

Our model of a generic merchant server is based on the template for HTTPS servers offered in the WIM [21].

Definition 71. The set of possible states Z^{ms} of a merchant server ms is the set of terms of the form:

$$\langle DNSaddress, pendingDNS, pendingRequests, tlskeys, keyMapping, corrupt\rangle$$

Where $DNSaddress$, $pendingDNS$, $pendingRequests$, $tlskeys$, $keyMapping$, and $corrupt$ defined as in the WIM [21].

The definition of the server state is therefore exactly the same as in the WIM.

An initial state s_0^{ms} of ms is a state of as with $s_0^{ms}.\texttt{pendingDNS} \equiv \langle\rangle$, $s_0^{ms}.\texttt{pendingRequests} \equiv \langle\rangle$, $s_0^{ms}.\texttt{corrupt} \equiv \bot$, $s_0^{ms}.\texttt{keyMapping}$ being the same as the keymapping for browsers [20], and $s_0^{ms}.\texttt{tlskeys} \equiv \texttt{tlskeys}^{as}$.

The template is implemented in Algorithm 31 with a basic interface offering the ability to process HTTPS requests of only a single kind:

`/index` serves an entry index page with the `script_merchant` script.

Algorithm 31 Relation of a merchant server R^i: Processing HTTPS Requests.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:     if m.path ≡ /index then
3:         let headers := ⟨⟨ReferrerPolicy, origin⟩⟩
4:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, headers, ⟨script_merchant, ⟨⟩⟩⟩, k)
5:         stop ⟨⟨f, a, m'⟩⟩, s'
6:     stop ⟨⟩, s'
```

`script_merchant` is not defined through code but through the following textual definition. `script_merchant` non-deterministically chooses a command to output for each iteration of the RUNSCRIPT algorithm (Algorithm 21).

J. Web Payment APIs model with attackers

The formal model of the Web Payment APIs is based on the definition of a web system given in Definition 27 of the WIM [21].

Definition 72. A web system $(\mathcal{W}, \mathcal{S}, \texttt{script}, E_0)$ belongs to $\mathcal{WPAPI}$ iff the following conditions are satisfied:

- $\mathcal{W}$ denotes a system described by a set of Dolev-Yao processes. It is partitioned into the sets Hon and Net. Net includes a network attacker process. Hon consists out of a finite set of web browsers B, a finite set of web servers for the merchants C and a finite set of payment provider servers PP with $\mathsf{Hon} := B \cup C \cup PP$.
  As in [20], DNS servers are not modeled directly as they are subsumed by the network attacker.
- $\mathcal{S}$ contains the scripts of the model with a mapping to their string representation. They are shown in table IV.
- E^0 is defined as in the WIM[21] as an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for every $a \in \cup_{p\in\mathcal{W}} I^p$.

$s \in \mathcal{S}$	script(s)
script_payment_provider_index	script_payment_provider_index
script_default_payment_handler	script_default_payment_handler
script_merchant	script_merchant

Table IV List of scripts with their mapping to a string representation

K. Mentionable simplifications and exclusions

In the model of the WPA presented here, some details differ from the exact specification. This section is intended to give a short overview over the aspects that differ and the reasoning for why these aspects were modeled differently.

During the processing of a payment request, there exists a set of situations in which a payment request can be updated by the merchant after the browser notifies the merchant of certain changes (e.g., after the customer entered a shipping address, so the merchant can update shipping costs if necessary). In our

model, a payment request can be updated independently of the existence of such a browser notification, i.e., we over-approximate what the merchant can do. This decision simplifies the resulting model significantly while not weakening the resulting proof. Note that in addition, the browser (as specified and also in our model) does not accept any updates once the customer submitted a payment.

The feature set of service workers that was modeled is not exhaustive. Features such as the installation process and their functionality as network proxies are not captured yet. Offering an exhaustive model of the service worker API would have gone beyond the scope of this work, but is planned for future work. The installation process is complex and issues within it would not have been linked to the WPA, but to the service worker API. Therefore, we assume that the service workers are already installed at the beginning of a system run. Furthermore, only the necessary features of service workers that directly affect the WPA are modeled.

In addition, we excluded the "merchant validation" sub-flow from our analysis, as it was considered a very early draft when we started our work and has since been removed completely from the specifications.

APPENDIX E
PROOFS

A. General Properties

Lemma 6 (Credentials do not leak).

For every WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, for every browser $b \in B$ honest in S, every $id \in b.\texttt{ids}$ with $pp = \texttt{governor}(id)$ and pp honest in S, it holds true:

$$\forall p \in \mathcal{W} \backslash \{b, pp\} : \texttt{secretOfID}(id) \notin d_\emptyset(S(p))$$

Proof:

Let b be a browser honest in S, $id \in b.\texttt{ids}$, and $pp = \texttt{governor}(id)$ a payment provider honest in S. Let $s := \texttt{secretOfID}(id)$ be the secret of the identity. We show that s can not be obtained by any other party $p \in \mathcal{W} \backslash \{b, pp\}$.

In the initial state S^0 of the run, only b and pp can derive s (see the definition of secretOfID in Section D-A and Definition 69).

In the following, we show that this secret is only sent out by b to pp, and to no other process. In particular, pp never sends out the secret.

We start by considering the browser b: By definition of the browser, only scripts and service workers from the origin $\langle id.\texttt{domain}, S \rangle$ can obtain a secret of this origin (see Line 7 of Algorithm 21 and Line 6 of Algorithm 23). The only scripts and service workers that have the same origin as pp are $script\_payment\_provider\_index$ (Algorithm 30) and $script\_default\_payment\_handler$ (Algorithm 24).

Case 1: $script\_payment\_provider\_index$**:** This script uses the credentials contained in the $secrets$ entry of its input only in Line 9 (Algorithm 30).

The secret is transmitted through an HTTPS connection to the host that served the script. Since this script has the same origin as the secret, this can only be pp. pp already knows the secret and does not leak them (as we will show below). More precisely, the script creates the command $\langle \texttt{XMLHTTPREQUEST}, url, \texttt{POST}, \langle id, secret \rangle, \bot \rangle$ (Line 10 of Algorithm 30), which the browser processes in Line 58 to Line 71 of Algorithm 21. Here, the browser checks the origin contained in the command, and only sends the request if the origin is equal to the origin of the script (Line 61 and Line 71 of Algorithm 21).

Case 2: $script\_default\_payment\_handler$**:** This service worker does not use the credentials contained in the $secrets$ entry of its input.

We note that the browser sends out the secret as part of an HTTPS message to pp. By applying Lemma 3, we can conclude that only pp can decrypt this request, therefore, a network attacker cannot derive the content of the request.

The payment provider pp only uses the credentials in Line 8 of Algorithm 29. In this line, the credentials are only compared to the submitted credentials of the user. At all other paths, even if s is part of the request, the payment provider will not send out the secret of an identity it governs.

Therefore, neither b nor pp do leak s to any other p, which proves the lemma.

■

Lemma 7 (Authorization tokens do not leak).

For every WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every payment provider server $pp \in PP$ honest in S, every $token \in \text{S}(pp).\texttt{tokens}$ with id such that $token \in^{\langle\rangle} \text{S}(pp).\texttt{tokens}[id]$ and $b \in B$ such that $b = \texttt{ownerOfID}(id)$ and b honest in S, it holds true that

$$\forall p \in \mathcal{W} \backslash \{b, pp\} : token \notin d_\emptyset(S(p))$$

Proof:

Let $token \in \text{S}(pp).\texttt{tokens}$ be a nonce with id such that $token \in^{\langle\rangle} \text{S}(pp).\texttt{tokens}[id]$ and $b \in B$ such that $b = \texttt{ownerOfID}(id)$ and b honest in S.

An honest payment provider modifies the tokens entry of its state only in Line 11 of Algorithm 29. In this line, the payment provider adds a token. This token is a nonce that is freshly created in Line 10 of Algorithm 29. In the following, we show that the payment provider sends $token$ within an HTTPS response to b, and to no other process.

As the payment provider pp stores $token$ in $\text{S}(pp).\texttt{tokens}[id]$ (Line 11 of Algorithm 29), we conclude that pp received an HTTP request to its /authenticate endpoint (Line 5 of Algorithm 29). Let $req_{\texttt{authn}}$

be this request. The key used for storing the token into $tokens$ (i.e., the identity) is taken from the request in Line 6 of Algorithm 29, i.e., $req_{\texttt{authn}}.\texttt{body}[\texttt{id}] = id$. Furthermore, this request contains the secret of id in $req_{\texttt{authn}}.\texttt{body}[\texttt{secret}]$ (see Lines 6 to 8 of Algorithm 29).

By applying Lemma 6, we conclude that $\forall p \in \mathcal{W} \backslash \{b, pp\} : \texttt{secretOfID}(id) \notin d_{\emptyset}(S(p))$. In particular, this means that either b or pp sent the request to the /authenticate endpoint of the payment provider. As an honest payment provider never sends HTTP requests, we conclude that the HTTP request was created and sent by the browser b.

The payment provider pp sends $token$ via an HTTPS response to the HTTPS request $req_{\texttt{authn}}$ in Line 12 of Algorithm 29.

By applying Corollary 1, we conclude that the symmetric key k is only known to b and pp, i.e., no other process can decrypt the response and obtain the token by decrypting the response (we note that an honest payment provider never leaks its own private decryption key, and also does not leak symmetric keys contained in HTTPS requests).

We highlight that the payment provider does not return a token (i.e., a nonce stored in the tokens entry of its state) when processing requests that are received at the /index or /pay endpoints.

It is left to show that the browser b does not leak $token$ to any other process.

The browser b only sends requests with credentials of an user to the /authenticate endpoint of pp after receiving an XMLHTTPREQUEST command by $script\_payment\_provider\_index$ (Line 10 of Algorithm 30). Furthermore, this is only possible if the script is running under an origin of pp, as only these scripts can obtain a secret of this origin (see Line 7 of Algorithm 21).

After the script $script\_payment\_provider\_index$ receives the response to the XMLHTTPREQUEST, it will (non-deterministically) either send another request to the /authenticate endpoint of pp (see Line 2 of Algorithm 30), or send a postMessage containing the token (see Line 12 of Algorithm 30). In the first case (request to the /authenticate endpoint) the script does not include values from its inputs $scriptinputs$ (which contains the token of the response) into the request. In the second case, the script takes the token from the $scriptinputs$ (Line 15 of Algorithm 30), and creates a command (Line 17 of Algorithm 30) that initiates the browser to send the token via the postMessage mechanism: $\langle \texttt{POSTMESSAGE}, swNonce, token, origin \rangle$. The last entry of this sequence is the origin of the script itself (see Line 13 and 14 of the same script), i.e., the origin of pp.

The second entry of the command, i.e., $swNonce$, is taken from the $scriptinputs$. This service worker nonce is added to the input $scriptinputs$ in Line 34 of Algorithm 28 by GET_SWS (Algorithm 27). GET_SWS only returns service workers that are of the same scope as the url of the corresponding HTTPS request (see Line 34 of Algorithm 28), i.e., service workers of pp.

The browser processes this POSTMESSAGE command in Line 82 of Algorithm 21. As the definition of the initial state of the browser does not allow a nonce to be used for both identifying a service worker and a window at the same time (see Definition 69), we conclude that the condition in Line 83 of Algorithm 21 is not true. If the second check in Line 88 of the same algorithm succeeds, then the message (with the token) is added to the inputs of a service worker with the same origin as the script, i.e., an origin of the payment provider. (If the second check fails, the message will be dropped, and the token cannot leak).

This service worker, therefore, has to be $script\_default\_payment\_handler$ (Algorithm 24), since no other payment handler is running under an origin of pp.

$script\_default\_payment\_handler$ (Algorithm 24) accesses its $scriptinputs$ in several lines, but in all of them the first field of the term is checked for the correct value. As for postMessages, the first entry is the string POSTMESSAGE, only Line 11 of Algorithm 24 is relevant.

The resulting message that is initiated by an XMLHTTPREQUEST command in Line 18 of Algorithm 24 is sent via an HTTPS request to the host of the service worker, i.e., to pp. As shown above, the payment provider pp does not leak any tokens contained in requests. Therefore, we conclude that $token$ does not leak to any other process.

■

Lemma 8 (No two payment storage entries have same transactionID). For every run ρ of a WPA system in $\mathcal{WPAPI}$, every configuration (S, E, N) in ρ, every honest $b \in B$, any transaction identifier $transaction\_id \in \mathcal{N}$ there exists at most one payment request nonce $\texttt{prn} \in \mathcal{N}$ such that:

$$(\mathrm{S}(b).paymentStorage[prn]).transactionID = transaction\_id$$

Proof:

Since b is honest by assumption, we only have to consider a non-corrupted behaving browser b. Note, that the $transactionID$ of a payment storage is never changed in the model after creation.

Therefore, we only have to consider the creation of objects in the $paymentStorage$ that have the required first value of $transactionID$. Within the model, such a payment storage entry is only generated in Line 100 of Algorithm 21. Each created payment storage is initialized with a new and random $transactionID$ (Line 99 of Algorithm 21). Therefore, there can not be two payment storage entries sharing the same transaction identifier.

■

Lemma 9 (methodData of payment request event contains one element). Given a WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every honest $b \in B$, any payment request event (as defined in Definition 64) $preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$, it holds true that $preqEvent.\texttt{methodData}$ contains one entry.

Proof:

Let $preqEvent$ be a payment request event s.t. $preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$.

The only place where an honest browser adds a payment request event to its `events` state entry is in Line 22 of Algorithm 22. We note that the `events` entry is initially empty, per Definition 69, and that the browser never modifies a payment request event stored in `events`.

Hence, the browser created $preqEvent$ in Line 20 of Algorithm 22.

The `methodData` entry of $preqEvent$ is created in Lines 15 to 18 of Algorithm 22, where it is initialized to an empty list in Line 15.

Method data contains at most one element. We now assume that $preqEvent.\texttt{methodData}$ contains at least two entries, and show that this assumption leads to a contradiction.

As $preqEvent.\texttt{methodData}$ contains two elements, we conclude that $\exists\ i_1, j_1$ s.t. $i_1 \neq j_1$ and

$$\pi_{i_1}(preqEvent.\texttt{methodData}).\texttt{pmi} = instrument.\texttt{enabledMethods} \tag{15}$$

$$\pi_{j_1}(preqEvent.\texttt{methodData}).\texttt{pmi} = instrument.\texttt{enabledMethods} \tag{16}$$

because a method data entry is only added to $methodData$ if the payment method identifier is equal to the `enabledMethods` of the instrument (Line 17 of Algorithm 22). Note that $instrument.\texttt{enabledMethods}$ is a single value, see Definition 56.

As these method data elements were added to the $methodData$ list in Line 18, we conclude that they were also contained in a payment request (as the values mds are taken from $paymentRequest.\texttt{methodData}$, see Line 16).

Let $preq$ be this payment request. We conclude that $\exists\ i_2, j_2$ s.t. $i_2 \neq j_2$ and

$$\pi_{i_2}(preq.\texttt{methodData}).\texttt{pmi} = instrument.\texttt{enabledMethods} \tag{17}$$

$$\pi_{j_2}(preq.\texttt{methodData}).\texttt{pmi} = instrument.\texttt{enabledMethods} \tag{18}$$

and in particular

$$\pi_{i_2}(preq.\texttt{methodData}).\texttt{pmi} = \pi_{j_2}(preq.\texttt{methodData}).\texttt{pmi} \tag{19}$$

Next, we trace back the origin of $preq$. This payment request is taken from $S'.\texttt{paymentStorage}[prn].\texttt{paymentRequest}$ (see Line 6 of Algorithm 22), for some nonce $prn \in \mathcal{N}$. The `paymentStorage` entry is initially empty (see Definition 69), and new entries are added only in Line 100 of Algorithm 21. While there are places where values of a payment request stored in `paymentStorage` are modified, the browser algorithms never modify the value of $s.\texttt{paymentStorage}[prn_2].\texttt{paymentRequest.methodData}$, for a browser state $s \in Z_{\text{Webbrowser}}$ and a nonce $prn_2 \in \mathcal{N}$.

In the processing step in which the browser created $preq$ and added it to `paymentStorage`, the `stop` in Line 96 of Algorithm 21 was not executed.

Let w.l.o.g. $i_2 < j_2$.

After $\pi_{i_2}(preq.\texttt{methodData})$ was processed by Lines 95 to 97 of Algorithm 21, $seenPMIs$ contains the value $\pi_{i_2}(preq.\texttt{methodData}).\texttt{pmi}$ (as the `stop` of the for loop was not executed).

As $\pi_{j_2}(preq.\texttt{methodData})$ was also processed by the for loop without executing the `stop`, we conclude that $\pi_{i_2}(preq.\texttt{methodData}).\texttt{pmi} \notin^{\langle\rangle} seenPMIs$, and in particular,

$$\pi_{i_2}(preq.\texttt{methodData}).\texttt{pmi} \neq \pi_{j_2}(preq.\texttt{methodData}).\texttt{pmi}$$

contradicting (19).

Method data contains at least one element. We now assume that $preqEvent$.methodData contains no entries, and show that this assumption leads to a contradiction. As shown above, there exists a payment request $preq$ from which the entries in $preqEvent$.methodData originate. We distinguish the following cases:

Case 1: $preq$.methodData $= \langle\rangle$

As shown above, the payment request $preq$ was created in Line 98 of Algorithm 21. However, the check in Line 93 of Algorithm 21 ensures that the method data entry of the payment request is not empty, i.e., $preq$.methodData $\neq \langle\rangle$.

Case 2: $preq$.methodData $\neq \langle\rangle$

As the $methodData$ entry of the payment request event is empty, i.e., $preqEvent$.methodData $= \langle\rangle$, it follows that $\forall mds \in^{\langle\rangle}$ $preq$.methodData. mds.pmi $\neq$ $instrument$.enabledMethods (see the loop in Line 16 of Algorithm 22), where $instrument$ is an instrument that supports one of the payment methods of the payment request (see Line 13 of Algorithm 22). However, as the check in Line 13 succeeded, such a list entry must exist, i.e., $\exists mds \in^{\langle\rangle}$ $preq$.methodData. mds.pmi $=$ $instrument$.enabledMethods.

■

Lemma 10 (Transaction in payment provider's state implies PaymentRequestEvent in browser's state). Given a WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every payment provider server $pp \in PP$ honest in S, and every $t \in$ S$(pp).transactions$ it holds true that:

If $b :=$ ownerOfID$(t$.sender$) \in B$ is a browser honest in S, then $\exists$ $preqEvent \in^{\langle\rangle}$ S(b).events such that:

$$\begin{aligned}
&\pi_1(preqEvent) = \texttt{PAYMENTREQUESTEVENT} \wedge \\
&preqEvent.\texttt{transactionId} = t.\texttt{transactionId} \wedge \\
&preqEvent.\texttt{total} = t.\texttt{total} \wedge \\
&\pi_1(preqEvent.\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver} \wedge \\
&\text{S}(pp).\texttt{transactions}[\langle \pi_1(preqEvent.\texttt{methodData}).\texttt{paymentIdentifier}\rangle] = t
\end{aligned}$$

and such an event has been processed by a payment handler installed in b and provided by pp to send an HTTPS request to pp to generate t.

Proof:

HTTP Request was created by b. We first show that the payment provider pp must have received an HTTPS request, and that this HTTPS request was created by b.

Let t be a transaction stored in the state of a payment provider pp honest in S, with $b :=$ ownerOfID$(t$.sender$) \in B$ being a browser honest in S.

Per Definition 70, the initial transactions state entry of the honest payment provider pp is empty, i.e., s_0^{pp}.transactions $\equiv \langle\rangle$. Therefore, we conclude that the payment provider pp added t to its state in a processing step.

Let $t_{\texttt{dictkey}}$ be a key under which t is stored in the transactions state entry (here, we do not assume that there is exactly one such key). Let $(S', E', N') \rightarrow (S'', E'', N'')$ be the last processing step that modifies the transactions$[t_{\texttt{dictkey}}]$ entry of pp prior to (S, E, N).

As pp is honest, it only modifies its transactions state entry in Line 22 of Algorithm 29, after receiving an HTTP POST request to its /pay endpoint (Line 14 of Algorithm 29).

Let req_t be the corresponding request that was processed in the aforementioned processing step. This request contains a token $t_{\texttt{token}}$, a receiver, a total, and the key that pp uses for storing the transaction t in its transactions state entry, i.e.,

$$t.\texttt{transactionId} = req_t.\texttt{body}[\texttt{transactionId}] \text{ (L. 21, Alg. 29)} \tag{20}$$

$$t.\texttt{total} = req_t.\texttt{body}[\texttt{total}] \text{ (L. 19, Alg. 29)} \tag{21}$$

$$t.\texttt{receiver} = req_t.\texttt{body}[\texttt{receiver}] \text{ (L. 18, Alg. 29)} \tag{22}$$

$$t_{dictkey} = req_t.\texttt{body}[\texttt{paymentIdentifier}] \text{ (L. 20, Alg. 29)} \tag{23}$$

As the transaction entry of the state of pp is updated, we can conclude that the stop in Line 17 was not executed, and that $t_{\texttt{token}} = req_t.\texttt{body}[\texttt{token}] \in S'(pp).\texttt{tokens}[t.\texttt{sender}]$.

As pp and $b =$ ownerOfID$(t$.sender$)$ were honest in S', we can apply Lemma 7, and we conclude that

$$\forall p \in \mathcal{W} \backslash \{b, pp\} : t_{\texttt{token}} \notin d_{\emptyset}(S'(p))$$

Therefore, we conclude that either pp or b must have constructed the HTTP request req_t. As an honest payment provider never sends HTTP requests (see also Algorithm 29), we can conclude that the HTTP request req_t was created by the browser b.

We highlight that the transaction identifier $t.\texttt{transactionId}$, the total $t.\texttt{total}$, the receiver $t.\texttt{receiver}$, and the payment identifier used as the dictionary key $t_{dictkey}$ that are taken from the same HTTPS request that contained the token, i.e., in req_t, and therefore, were sent by the browser b.

Origin of req_t inside the browser b. Next, we show that the request was initiated by an payment handler with the same origin as pp.

In general, a call to the /pay endpoint of the payment provider pp can have had several origins, such as an URL navigation, a FORM command of a script, an IFRAME command of a script, a XMLHTTPREQUEST command of a script or a service worker and so forth.

But since the call has to be authorized by a token $token$ and has to be a POST call, only $script\_default\_payment\_handler$ can be the source of the call.

$script\_payment\_provider\_index$ can not issue a POST call to /pay and only shares tokens with a $script\_default\_payment\_handler$ served from the same domain as $script\_payment\_provider\_index$. As argued in Lemma 7, the token is only used in Line 11 of Algorithm 24. Which is followed by the relevant XMLHTTPREQUEST command, that calls /pay.

Therefore, we conclude that in the processing step in which the browser b creates req_t, the browser executed the default payment handler, and in particular, this payment handler has the same origin as the payment provider pp.

Exists payment request event with same transaction id, receiver, total, and payment identifier used for t.

Now, we track back where the values used by the payment handler for the XMLHTTPREQUEST originate from.

An honest payment handler creates XMLHTTPREQUEST commands only in Line 18 and Line 40 of Algorithm 24. As argued above, req_t is a POST request. Thus, we conclude that the honest payment handler created the XMLHTTPREQUEST in Line 18 of Algorithm 24, as the honest payment handler does not create a POST request in Line 40.

The browser, when executing such a command of a payment handler, does not alter the values inside the request: An XMLHTTPREQUEST command triggers the browser b to send an HTTPS request, see Line 32 of Algorithm 23 (we note that the URL has the same origin as the payment handler, see also Line 26 of Algorithm 23).

As this is a POST request, the body of the corresponding HTTP request is taken from the $data$ field of the input XMLHTTPREQUEST (Line 24 of Algorithm 23). Therefore, we have that

$$req_t.\texttt{body} = data \tag{24}$$

The content of the $data$ entry is specified by the payment handler in Line 18 of Algorithm 24.

We now show that values contained in the data entry of the XMLHTTPREQUEST are taken by the payment handler from an PAYMENTREQUESTEVENT.

As Line 8 shows, an event term with its first field being PAYMENTREQUESTEVENT is needed in the script inputs of the payment handler script for the payment handler to create the XMLHTTPREQUEST command. Let $preEvent$ denote this event.

In Line 13 of Algorithm 24, the payment handler selects one method data entry.

We now conclude the following:

$$\pi_1(preqEvent) = \texttt{PAYMENTREQUESTEVENT} \text{ (L. 8, Alg. 24)} \tag{25}$$

$$\exists i.\pi_i(preqEvent.\texttt{methodData}).\texttt{paymentIdentifier} = data.\texttt{paymentIdentifier} \text{ (L. 15, Alg. 24)} \tag{26}$$

$$\exists i.\pi_i(preqEvent.\texttt{methodData}).\texttt{receiver} = data.\texttt{receiver} \text{ (L. 14, Alg. 24)} \tag{27}$$

$$preqEvent.\texttt{transactionId} = data.\texttt{transactionId} \text{ (L. 17, Alg. 24)} \tag{28}$$

$$preqEvent.\texttt{total} = data.\texttt{total} \text{ (L. 10, Alg. 24)} \tag{29}$$

Payment Request Event was in Pool of Events. The only place inside the browser b where such an event is added to the inputs of a payment handler is in the DELIVER_TO_DOC helper function (Line 4 of Algorithm 26), when called in Line 28 of Algorithm 22. The event is passed to the service worker in an unmodified manner.

This happens only if the function PROCESSEVENT (Algorithm 22) is called with an PAYMENTREQUESTEVENT, i.e., PROCESSEVENT was called with the payment request event $preEvent$. The function PROCESSEVENT (Algorithm 22) is called within the main function of the browser b (Line 49 of Algorithm 20).

The events passed to the function PROCESSEVENT are only taken from the pool of events store in the browsers state (Line 47 of Algorithm 20). Therefore, $preEvent$ must have been stored in S'''(b).events for some state S''' of a configuration prior to (S, E, N).

As an honest browser never removes events from its events sequence, we conclude that $preEvent \in^{\langle\rangle}$ S(b).events.

Now, we apply Lemma 9, and conclude that $preEvent$.methodData contains exactly one element, i.e.,

$$\pi_1(preqEvent.\texttt{methodData}).\texttt{paymentIdentifier} = t_{dictkey} \wedge$$
$$\pi_1(preqEvent.\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver}$$

Using 20-23, 24, and 25 - 29, it follows that

$$\pi_1(preqEvent) = \texttt{PAYMENTREQUESTEVENT} \wedge$$
$$\pi_1(preqEvent.\texttt{methodData}).\texttt{paymentIdentifier} = t_{dictkey} \wedge$$
$$\pi_1(preqEvent.\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver} \wedge$$
$$preqEvent.\texttt{transactionId} = t.\texttt{transactionId} \wedge$$
$$preqEvent.\texttt{total} = t.\texttt{total}$$

As honest browser also enforces that a payment handler can only send requests to its origin (Line 26 of Algorithm 23), therefore the payment handler processing $preqEvent$ must be provided by pp. Hence, Lemma 10 is proven.

■

Lemma 11 (PaymentRequestEvent in browser's state implies SubmitPayment in browser's state). Given a WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every browser b honest in S, every $preqEvent \in^{\langle\rangle}$ S(b).events with $\pi_1(preqEvent) =$ PAYMENTREQUESTEVENT, it holds true that:

$$\exists\ spEvent.\ spEvent \in^{\langle\rangle} \text{S}(b).\texttt{events} \wedge$$
$$\pi_1(spEvent) = \texttt{SUBMITPAYMENT} \wedge$$
$$\pi_2(spEvent) = preqEvent.\texttt{paymentRequestNonce} \wedge$$
$$\pi_3(spEvent) = preqEvent.\texttt{handlerNonce}$$

Proof:

Let $preqEvent$ be an event such that $preqEvent \in^{\langle\rangle}$ S(b).events with $\pi_1(preqEvent) =$ PAYMENTREQUESTEVENT, for a browser b that is honest in S.

An honest browser b adds such an event only in Line 22 of Algorithm 22 to its events state entry.

This happens only if the function PROCESSEVENT (Algorithm 22) is called with an SUBMITPAYMENT event (see Line 5 of Algorithm 22). Let $spEvent$ be this event. The value of $preqEvent$.paymentRequestNonce is taken directly from $spEvent$ and is the second entry of $spEvent$, i.e., $\pi_2(spEvent) = preqEvent$.paymentRequestNonce (see Line 20 of Algorithm 22). The value of $preqEvent$.handlerNonce is the nonce of an handler. This nonce is chosen such that it is the same as the handler nonce contained in the $spEvent$ event (see Line 9 of Algorithm 22).

The function PROCESSEVENT (Algorithm 22) is called within the main function of the browser b (Line 49 of Algorithm 20).

The events passed to the function PROCESSEVENT are only taken from the pool of events stored in the browsers state (Line 47 of Algorithm 20). Therefore, $spEvent$ must have been stored in $S'(b)$.events for some state S' of a configuration prior to (S, E, N).

As an honest browser never removes events from its events state entry, we conclude that $spEvent \in^{\langle\rangle}$ S(b).events.

■

B. Intended Payments

Lemma 12 (Payment Request Event Implies Payment Intent).
For every WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every browser $b \in B$ that is honest in S, every payment request event (as defined in Definition 64) $preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$, it holds true that:

$$\begin{aligned}
&\exists payment_intent.\\
&payment_intent \in^{\langle\rangle} \mathrm{S}(b).\texttt{paymentIntents}[preqEvent.\texttt{transactionId}] \wedge\\
&payment_intent.\texttt{details.total} = preqEvent.\texttt{total} \wedge\\
&\pi_1(payment_intent.\texttt{methodData}).\texttt{receiver} = \pi_1(preqEvent.\texttt{methodData}).\texttt{receiver}
\end{aligned}$$

Proof:
Let $preqEvent$ be a payment request event such that $preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$, and b a browser honest in S. Initially, the `events` state entry of the browser is empty (per Definition 69). An honest browser adds payment requests events to its `events` state entry only in Line 22 of Algorithm 22. After this line is executed, the browser will execute Line 25 of the same algorithm (as there is no `stop` between these lines), in which the browser stores a payment intent pi. The dictionary key used for storing the intent is the transaction identifier that is also stored in $preqEvent$ i.e., $preqEvent.\texttt{transactionId}$ (see Line 20 and Line 25 of Algorithm 22).

Moreover, there exists a payment request $paymentReq$ such that $pi.\texttt{details.total} = paymentReq.\texttt{details.total}$ (Line 23 of Algorithm 22). The value of the total of $preqEvent$ is set to the value stored in $paymentReq$, i.e., $preqEvent.\texttt{total} = paymentReq.\texttt{details.total}$ (Line 10 and Line 20 of Algorithm 22). Therefore, we conclude that $pi.\texttt{details.total} = preqEvent.\texttt{total}$.

The `methodData` entry stored in the payment intent pi in Line 24 is the same that is stored in the payment request event $preqEvent$ in Line 20 of Algorithm 22, i.e., $\pi_1(preqEvent.\texttt{methodData}).\texttt{receiver} = \pi_1(pi.\texttt{methodData}).\texttt{receiver}$.

■

Lemma 13 (Intended Payments).
For every WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every payment provider server $pp \in PP$ honest in S, and every $t \in \mathrm{S}(pp).transactions$ it holds true that:
If $b := \texttt{ownerOfID}(t.\texttt{sender}) \in B$ is a browser honest in S, then $\exists i \in \mathbb{N}$ such that:

- $\pi_i(\mathrm{S}(b).\texttt{paymentIntents}[t.\texttt{transactionId}]).\texttt{details.total} = t.\texttt{total}$
- $\pi_1(\pi_i(\mathrm{S}(b).\texttt{paymentIntents}[t.\texttt{transactionId}]).\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver}$

Proof:
Let t be a transaction stored in the state of an honest payment provider pp, i.e., $t \in \mathrm{S}(pp).transactions$. We apply Lemma 10 and conclude that, if $b := \texttt{ownerOfID}(t.\texttt{sender}) \in B$ is a browser honest in S, then $\exists\ preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ such that:

$$\begin{aligned}
&\pi_1(preqEvent) = \texttt{PAYMENTREQUESTEVENT} \wedge\\
&preqEvent.\texttt{transactionId} = t.\texttt{transactionId} \wedge\\
&preqEvent.\texttt{total} = t.\texttt{total} \wedge\\
&\pi_1(preqEvent.\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver}.
\end{aligned}$$

Next, we apply Lemma 12, and conclude that there is a payment intent pi such that
$pi \in^{\langle\rangle} \mathrm{S}(b).\texttt{paymentIntents}[preqEvent.\texttt{transactionId}] \wedge$
$pi.\texttt{details.total} = preqEvent.\texttt{total} \wedge$
$\pi_1(pi.\texttt{methodData}).\texttt{receiver} = \pi_1(preqEvent.\texttt{methodData}).\texttt{receiver}$,
and, in particular,
$pi \in^{\langle\rangle} \mathrm{S}(b).\texttt{paymentIntents}[t.\texttt{transactionId}] \wedge$
$pi.\texttt{details.total} = t.\texttt{total} \wedge$
$\pi_1(pi.\texttt{methodData}).\texttt{receiver} = t.\texttt{receiver}$.

■

C. Uniqueness of Payments

Lemma 14 (Same payment request nonces if events have same transaction identifiers). Given a WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every browser b honest in S, for all payment request events (as defined in Definition 64) $preqEvent_1, preqEvent_2 \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ it holds true that:

$$preqEvent_1.\texttt{transactionId} = preqEvent_2.\texttt{transactionId} \implies preqEvent_1.\texttt{paymentRequestNonce} = preqEvent_2.\texttt{paymentRequestNonce}$$

Proof:

We prove Lemma 14 by contradiction. Assume that there exist payment request events (see Definition 64) $preqEvent_1, preqEvent_2 \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ with $preqEvent_1.\texttt{transactionId} = preqEvent_2.\texttt{transactionId}$ and $preqEvent_1.\texttt{paymentRequestNonce} \neq preqEvent_2.\texttt{paymentRequestNonce}$.

Because b is honest, all events have never been changed, replaced or removed after they were added into in $\mathrm{S}(b).\texttt{events}$. We trace back to the place where their transaction identifiers and payment request nonces were generated.

We can see from the model that payment request events can only be added into $\mathrm{S}(b).\texttt{events}$ by processing a `SubmitPayment` event (Line 20 of Algorithm 22). Its payment request nonce is the payment request nonce of the corresponding `SubmitPayment` event and its transaction identifier is the transaction identifier of the payment storage indexed by the payment request nonce (Line 19 of Algorithm 22).

Let $\texttt{spEvent}_1$ and $\texttt{spEvent}_2$ be two `SubmitPayment` events that were processed to generate $preqEvent_1$ and $preqEvent_2$, respectively. Let $(S_1, E_1, N_1), (S_2, E_2, N_2) \in \rho$ be the two output configurations of the processing steps in which $\texttt{spEvent}_1$ and $\texttt{spEvent}_2$ were processed. Because $preqEvent_1, preqEvent_2 \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$, the two configurations must be *previous* configurations of (S, E, N) in ρ. As b is honest in S, b was also honest in S_1 and S_2.

Let $PRN_1 = \texttt{spEvent}_1.\texttt{paymentRequestNonce}$ and $PRN_2 = \texttt{spEvent}_2.\texttt{paymentRequestNonce}$. By the assumption of the proof we have that

$$PRN_1 \neq PRN_2 \tag{30}$$

As reasoned above, we have that

$$preqEvent_1.\texttt{transactionId} = S_1(b).\texttt{paymentStorage}[PRN_1].\texttt{transactionId}$$

and

$$preqEvent_2.\texttt{transactionId} = S_2(b).\texttt{paymentStorage}[PRN_2].\texttt{transactionId}$$

Without loss of generality we assume that (S_2, E_2, N_2) is the *later* configuration in ρ. Because an entry of the payment storage in the state of an honest browser is never removed, and the transaction identifier of the entry is never updated, we have that $S_2(b).\texttt{paymentStorage}[PRN_1].\texttt{transactionId} = S_1(b).\texttt{paymentStorage}[PRN_1].\texttt{transactionId}$. By the assumption of the proof, we have that $preqEvent_1.\texttt{transactionId} = preqEvent_2.\texttt{transactionId}$. Therefore, we have

$$S_2(b).\texttt{paymentStorage}[PRN_1].\texttt{transactionId} = S_2(b).\texttt{paymentStorage}[PRN_2].\texttt{transactionId} \tag{31}$$

Lemma 8 states that for every payment storage in the state of an honest browser, there does not exist two different payment storage entries having the same transaction identifier value. From (31) we conclude that $PRN_1 = PRN_2$, which contradicts to (30). Hence, Lemma 14 is proven.

■

Lemma 15 (Same handler nonces if same payment request nonces). Given a WPA system in $\mathcal{WPAPI}$, for every run ρ of this system, every configuration (S, E, N) in ρ, every browser b honest in S, for all submit payment events (as defined in Definition 63) $spEvent_1, spEvent_2 \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$, it holds true that:

$$spEvent_1.\texttt{paymentRequestNonce} = spEvent_2.\texttt{paymentRequestNonce} \implies spEvent_1.\texttt{handlerNonce} = spEvent_2.\texttt{handlerNonce}$$

Proof:

We prove Lemma 15 by contradiction. Assume that there exist two submit payment events $spEvent_1, spEvent_2 \in^{\langle\rangle}$ S(b).events (see Definition 63) with $spEvent_1$.paymentRequestNonce $=$ $spEvent_2$.paymentRequestNonce and $spEvent_1$.handlerNonce $\neq$ $spEvent_2$.handlerNonce. Let $PRN :=$ $spEvent_1$.paymentRequestNonce. Furthermore, let $(S_1, E_1, N_1), (S_2, E_2, N_2) \in \rho$ be the two input configurations of the processing steps in which $spEvent_1$ and $spEvent_2$ were added into b's state. Without loss of generality we assume that (S_1, E_1, N_1) is a *previous* configuration of (S_2, E_2, N_2) in ρ. It is easy to see that both configurations are *previous* configurations of (S, E, N) in ρ. Because b is honest in S, b was also honest in S_1 and S_2.

An event of which SUBMITPAYMENT is the first field is only added into the state of an honest browser in the Algorithm 21, either by processing the command PR_SHOW (Line 122) or by processing the command PRES_RETRY (Line 153). In both cases the paymentRequestNonce of the command is used to set for the second field of the SubmitPayment event. We prove that $spEvent_2$ can only be added into b's state by processing the command PRES_RETRY.

Assume that $spEvent_2$ was added into b's state by processing PR_SHOW. $spEvent_2$ was added at Line 122 of Algorithm 21. That line can only be executed if the state of the payment request stored in the payment storage indexed by PRN was CR (Line 105 of Algorithm 21), i.e., we have that

$$S_2(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.state} = CR \tag{32}$$

Let (S_1', E_1', N_1') be the *right next* configuration state of (S_1, E_1, N_1). As (S_1, E_1, N_1) is the previous configuration of (S_2, E_2, N_2), we have that (S_1', E_1', N_1') is either (S_2, E_2, N_2) or a previous configuration of (S_2, E_2, N_2).

If $(S_1', E_1', N_1') \equiv (S_2, E_2, N_2)$, we have that $S_1'(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.state} = CR$. However, because $spEvent_2$ had been added into b's state, the state of corresponding paymentRequest must be set to IN (Lines 109 and 151 of Algorithm 21). Therefore, $S_1'(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.state} = IN$ (contraction). Thus, (S_1', E_1', N_1') must be a previous configuration of (S_2, E_2, N_2) in ρ. As reasoned above, we have that $S_1'(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.state} = IN$

The field paymentRequest.state of an entry in the payment storage stored in the state of an honest browser can only be set to CR at Line 98 of Algorithm 21. This leads to the creation of a new payment storage entry, where a fresh nonce is taken as the payment request nonce indexing such entry. If the fresh nonce is PRN, the configuration on which paymentRequest.state is set to CR must be a previous configuration of (S_1', E_1', N_1'). It means that $S_2(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.state} = IN$, contradicting to (32). Therefore, $spEvent_2$ can only be added into b's state by processing the command PRES_RETRY.

As $spEvent_2$ was added into b's state by processing the command PRES_RETRY, we have that $spEvent_2$.handlerNonce $= S_2(b).\texttt{paymentStorage}[PRN].\texttt{handlerNonce}$ (Line 152 of Algorithm 21). We show that $spEvent_1$.handlerNonce $= S_1'(b).\texttt{paymentStorage}[PRN].\texttt{handlerNonce}$. This is true because if $spEvent_1$ was added into b's state by processing a $PRES\_RETRY$ command, then $spEvent_1$.handlerNonce $= S_1(b).\texttt{paymentStorage}[PRN].\texttt{handlerNonce} = S_1'(b).\texttt{paymentStorage}[PRN].\texttt{handlerNonce}$. In case that $spEvent_1$ was added into b's state by processing a PR_SHOW command, both $spEvent_1$.handlerNonce and $S_1'(b).\texttt{paymentStorage}[PRN].\texttt{handlerNonce}$ were set to the nonce of the chosen handler (Lines 118 and 122 of Algorithm 21).

Because handlerNonce in an entry of the payment storage in the state of an honest browser never change, we have that $S_1'(b).\texttt{paymentStorage}[PRN].\texttt{handlerNonce} = S_2(b).\texttt{paymentStorage}[PRN].\texttt{handlerNonce}$, implying that $spEvent_1$.handlerNonce $=$ $spEvent_2$.handlerNonce. This contradicts to the proof assumption and therefore, Lemma 15 is proven.

■

Lemma 16 (All entries in method data of payment request have the same payment identifier). For every run ρ of a system in $\mathcal{WPAPI}$, every configurations $(S_1, E_1, N_1), (S_2, E_2, N_2)$ in ρ, every browser $b \in B$ honest in S_1 and S_2, every nonce $PRN \in \mathcal{N}$, it holds true that:

$$\begin{aligned}\forall\ & x_i \in^{\langle\rangle} S_1(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.methodData},\\ & x_j \in^{\langle\rangle} S_2(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.methodData}\ :\\ & x_i.\texttt{paymentIdentifier} = x_j.\texttt{paymentIdentifier}\end{aligned}$$

Proof:

Without loss of generality, we assume that (S_1, E_1, N_1) is a *previous* configuration of (S_2, E_2, N_2) in ρ. Because b is honest in S_2, b was honest in all previous global states. Therefore, the method data of the payment request stored in a payment storage entry in b's state never change. We consider the place where `paymentRequest.methodData` is initialized. Line 98 of Algorithm 21 shows that the method data of the payment request is taken from the input $methodData$ in the `PR_CREATE` command. Line 92 checks whether the input $methodData$ satisfies Definition 55, guaranteeing that all entries of the input $methodData$ have the same payment identifier. Thus, Lemma 16 is proven.

■

Lemma 17 (Same payment identifier if same transaction identifier). For every run ρ of a system in $\mathcal{WPAPI}$, every configuration (S, E, N) in ρ, every browser $b \in B$ honest in S, every payment request events (as defined in Definition 64) pre_1, $pre_2 \in^{\langle\rangle}$ S(b).`events`, it holds true that:

$$(pre_1.\texttt{transactionId} = pre_2.\texttt{transactionId}) \Rightarrow$$
$$(\pi_1(pre_1.\texttt{methodData}).\texttt{paymentIdentifier} = \pi_1(pre_2.\texttt{methodData}).\texttt{paymentIdentifier})$$

Proof:

Consider two arbitrary payment request events (see Definition 64) pre_1, $pre_2 \in^{\langle\rangle}$ S(b).`events` with pre_1.`transactionId` = pre_2.`transactionId`. We will prove that $(\pi_1(pre_1.\texttt{methodData}).\texttt{paymentIdentifier} = \pi_1(pre_2.\texttt{methodData}).\texttt{paymentIdentifier})$.

Similar to the proof for Lemma 14, there exist two `SubmitPayment` events $\texttt{spEvent}_1, \texttt{spEvent}_2 \in^{\langle\rangle}$ S(b).`events` that were processed to generate pre_1, pre_2, respectively. We also have from the proof for Lemma 14 that $\texttt{spEvent}_1.\texttt{paymentRequestNonce} = \texttt{spEvent}_2.\texttt{paymentRequestNonce}$. Let $PRN :=$ $\texttt{spEvent}_1.\texttt{paymentRequestNonce}$.

Let $(S_1, E_1, N_1), (S_2, E_2, N_2) \in \rho$ be the two output configurations of the processing steps in which $\texttt{spEvent}_1$ and $\texttt{spEvent}_2$ were processed to generate pre_1, pre_2, respectively. We showed in the proof of Lemma 14 that both (S_1, E_1, N_1) (S_2, E_2, N_2) are previous configuration of (S, E, N) in ρ, and that b was honest in both S_1 and S_2.

pre_1 and pre_2 were created and added into events by executing the PROCESSEVENT function for the case of `SubmitPayment` (Lines 20 and 22 of Algorithm 22). All entries of `methodData` of the created `PAYMENTREQUESTEVENT` are taken from the `methodData` field of the corresponding payment request stored in the payment storage (Lines 16 to 18 of Algorithm 22). Therefore, we have that

$$\pi_1(pre_1.\texttt{methodData}) \in^{\langle\rangle} S_1(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.methodData}$$

and

$$\pi_1(pre_2.\texttt{methodData}) \in^{\langle\rangle} S_2(b).\texttt{paymentStorage}[PRN].\texttt{paymentRequest.methodData}$$

Applying Lemma 16, we can conclude that $(\pi_1(pre_1.\texttt{methodData}).\texttt{paymentIdentifier} = \pi_1(pre_2.\texttt{methodData}).\texttt{paymentIdentifier})$. Lemma 17 is proven.

■

Lemma 18 (Uniqueness of Transaction). For every run ρ of a system in $\mathcal{WPAPI}$, every configuration (S, E, N) in ρ, every browser $b \in B$ honest in S, every $(txId, intents) \in$ S(b).`paymentIntents`, and with $PP_h = \{pp \in PP : \text{S}(pp).\texttt{isCorrupted} = \bot\}$ being the set of payment providers that are honest in S, it holds true that

$$\left| \bigcup_{pp \in PP_h} \left\{ t \in \text{S}(pp).\texttt{transactions} \mid txId = t.\texttt{transactionId} \wedge b = \texttt{ownerOfID}(\texttt{t.sender}) \right\} \right| \leq 1$$

Proof:

We prove Lemma 18 by contradiction. Without loss of generality, let b be an honest browser in S and $(txId, intents) \in$ S(b).`paymentIntents`. We assume that there exist two different transactions t, t' and two honest payment providers pp, pp' (pp and pp' can be the same) so that

$$t \in \text{S}(pp).\texttt{transactions} \ \wedge \ txId = t.\texttt{transactionId} \wedge b = \texttt{ownerOfID}(\texttt{t.sender}) \tag{33}$$

and

$$t' \in \text{S}(pp').\texttt{transactions} \ \wedge \ txId = t'.\texttt{transactionId} \wedge b = \texttt{ownerOfID}(\texttt{t}'\texttt{.sender}) \tag{34}$$

From (33) and Lemma 10, we have that: $\exists\ preqEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ such that:

$$\begin{aligned}&\pi_1(preqEvent) = \texttt{PAYMENTREQUESTEVENT} \wedge \\ &preqEvent.\texttt{transactionId} = t.\texttt{transactionId} = txId \wedge \\ &\mathrm{S}(pp).\texttt{transactions}[\langle \pi_1(preqEvent.\texttt{methodData}).\texttt{paymentIdentifier}\rangle] = t\end{aligned} \tag{35}$$

and $preqEvent$ has been delivered to a payment handler provided by pp to send an HTTPS request to pp generating t.

From (34) and Lemma 10, we have that $\exists\ preqEvent' \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ such that:

$$\begin{aligned}&\pi_1(preqEvent') = \texttt{PAYMENTREQUESTEVENT} \wedge \\ &preqEvent'.\texttt{transactionId} = t'.\texttt{transactionId} = txId \wedge \\ &\mathrm{S}(pp').\texttt{transactions}[\langle \pi_1(preqEvent'.\texttt{methodData}).\texttt{paymentIdentifier}\rangle] = t'\end{aligned} \tag{36}$$

and $preqEvent'$ has been delivered to a payment handler provided by pp' to send an HTTPS request to pp' generating t'.

As $preqEvent.\texttt{transactionId} = preqEvent'.\texttt{transactionId} = txId$, from Lemma 14 we have that

$$preqEvent.\texttt{paymentRequestNonce} = preqEvent'.\texttt{paymentRequestNonce} \tag{37}$$

We have from (35) and Lemma 11 that there exists $spEvent \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ such that

$$\begin{aligned}&\pi_1(spEvent) = \texttt{SUBMITPAYMENT} \wedge \pi_2(spEvent) = preqEvent.\texttt{paymentRequestNonce} \wedge \\ &\pi_3(spEvent) = preqEvent.\texttt{handlerNonce}\end{aligned} \tag{38}$$

Similarly, from (36) and Lemma 11 we have an event $spEvent' \in^{\langle\rangle} \mathrm{S}(b).\texttt{events}$ such that

$$\begin{aligned}&\pi_1(spEvent') = \texttt{SUBMITPAYMENT} \wedge \pi_2(spEvent') = preqEvent'.\texttt{paymentRequestNonce} \wedge \\ &\pi_3(spEvent') = preqEvent'.\texttt{handlerNonce}\end{aligned} \tag{39}$$

From (37), (38), and (39) we have:

$$\pi_2(spEvent) = \pi_2(spEvent') \tag{40}$$

From (40), we apply Lemma 15 and conclude that

$$\pi_3(spEvent) = \pi_3(spEvent') \tag{41}$$

From (38), (39), and (41) we have:

$$preqEvent.\texttt{handlerNonce} = preqEvent'.\texttt{handlerNonce}$$

The browser chooses the payment handler to process a `PaymentRequestEvent` using the handler nonce value of the event (Line 3 of Algorithm 26). Because the two `PaymentRequestEvent` events $preqEvent$ and $preqEvent'$ have the same handler nonce, we conclude that they are processed by the same payment handler. An honest browser also enforces that a payment handler can only send requests to its own origin (Line 26 of Algorithm 23), therefore, the payment handler, after processing $preqEvent$ and $preqEvent'$, can only send corresponding requests to the same /`pay` endpoint, implying that t and t' are stored in the same payment provider's state, i.e., $pp \equiv pp'$.

We have proven above that $preqEvent$ and $preqEvent'$ have the same transaction identifier value, and by applying Lemma 17, we conclude that

$$\begin{aligned}&\pi_1(preqEvent.\texttt{methodData}).\texttt{paymentIdentifier} = \\ &\qquad\qquad \pi_1(preqEvent'.\texttt{methodData}).\texttt{paymentIdentifier}\end{aligned}$$

thus, from (35) and (36) we can derive that $t = t'$. This contradicts the assumption that t and t' are different. Therefore, Lemma 18 holds.

■

Lemma 19 (Uniqueness of Payments). For every run ρ of a system in $\mathcal{WPAPI}$, every configuration (S, E, N) in ρ, every browser $b \in B$ honest in S, every $(txId, intents) \in \mathrm{S}(b).\texttt{paymentIntents}$, and

with $PP_h = \{pp \in PP : \mathrm{S}(pp).\texttt{isCorrupted} = \bot\}$ being the set of payment providers that are honest in S, it holds true that

$$\left| \bigcup_{pp \in PP_h} \left\{ t \in \mathrm{S}(pp).\texttt{transactions} \mid txId = t.\texttt{transactionId} \wedge b = \texttt{ownerOfID}(\texttt{t.sender}) \right\} \right| \leq 1$$

If such a t exists, there exists a $pi \in intents$ such that:
$t.\texttt{total} = pi.\texttt{details.total}$ and $t.\texttt{receiver} = \pi_1(pi.\texttt{methodData}).\texttt{receiver}$.

Proof:

Lemma 19 is proven by applying Lemma 18 and Lemma 13.

■

D. Proof of Theorem

Theorem 1 directly follows from Lemmas 13 and 19.

■