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 †Royal Holloway, University of London, Egham, Surrey, UK
{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.

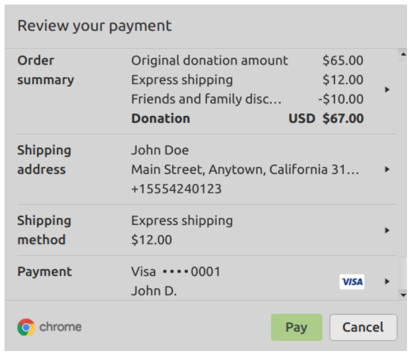
So far, there has been no in-depth security analysis of these new standards. In this paper, we present the first such analysis of the Web Payment APIs standards, a rigorous formal analysis. It is based on the *Web Infrastructure Model (WIM)*, the most comprehensive model of the web infrastructure to 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 [17], just to name a few. Also, customers are likely to abort a checkout process when facing a bad user experience [27].


Review your payment

Order summary	Original donation amount	$65.00
	Express shipping	$12.00
	Friends and family disc...	-$10.00
	Donation	**USD $67.00**
Shipping address	John Doe Main Street, Anytown, California 31... +15554240123	
Shipping method	Express shipping $12.00	
Payment	Visa ••••0001 John D.	VISA

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., [9, 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 [16], 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 [8] and [10], 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.
- During our analysis, we uncovered two previously unknown critical flaws which would allow a malicious

[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.

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. Further details are given in the appendix, with full details and proofs in our technical report [18].

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 [11, 12, 13, 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 [13], 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 [13]. 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 [11]. 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 [12]. 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 $\boxed{1}$, 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 $\boxed{2}$. 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` $\boxed{3}$.[3] After this initial selection, the browser triggers each of the selected payment handlers with a `CanMakePaymentEvent` $\boxed{4}$. 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`).

[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.

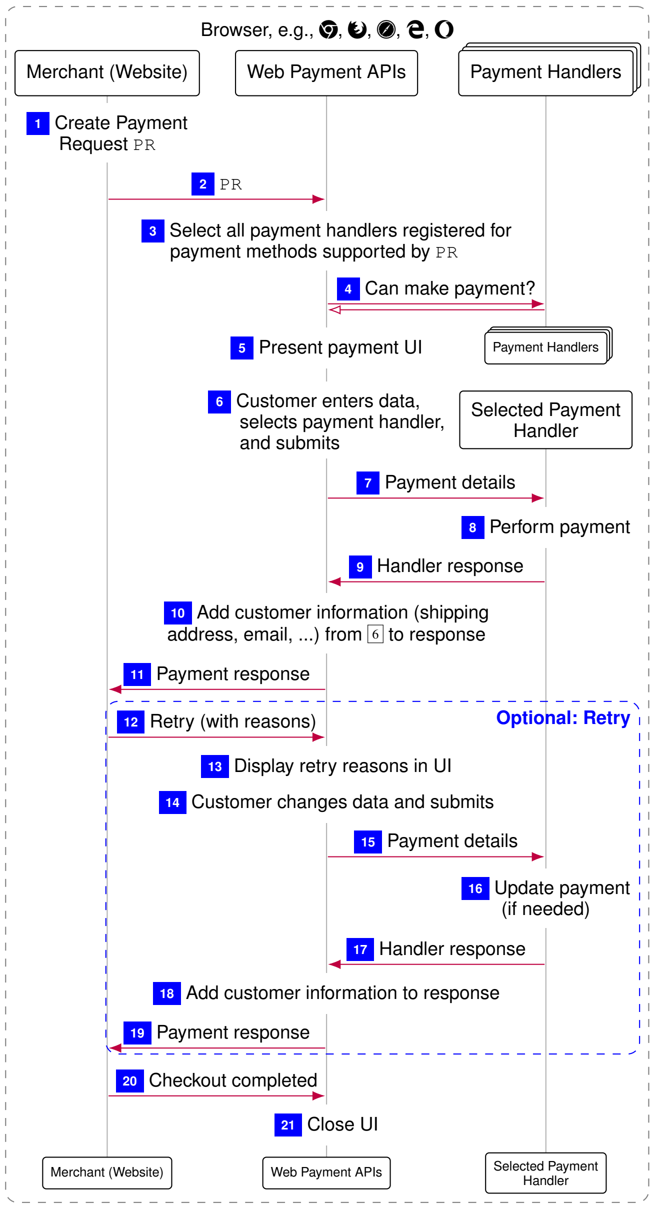


Figure 2 General flow of a payment with the WPA. Note that payment handlers are not part of the WPA implemented by the browser. They are web service workers – typically provided by the payment provider – which use the WPA to engage in the payment process. They may also send requests, e.g., to the payment 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 a website, but of the browser itself) called the *payment UI* $\boxed{5}$ (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 $\boxed{6}$. 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 $\boxed{7}$. 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 $\boxed{8}$. 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` $\boxed{9}$. 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 $\boxed{10}$. 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 $\boxed{11}$.

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 $\boxed{20}$ (in this case, the browser closes the payment UI $\boxed{21}$) or by indicating that there is a problem $\boxed{12}$. 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 $\boxed{13}$ and allows the user to revise the entered data $\boxed{14}$.[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.

[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.

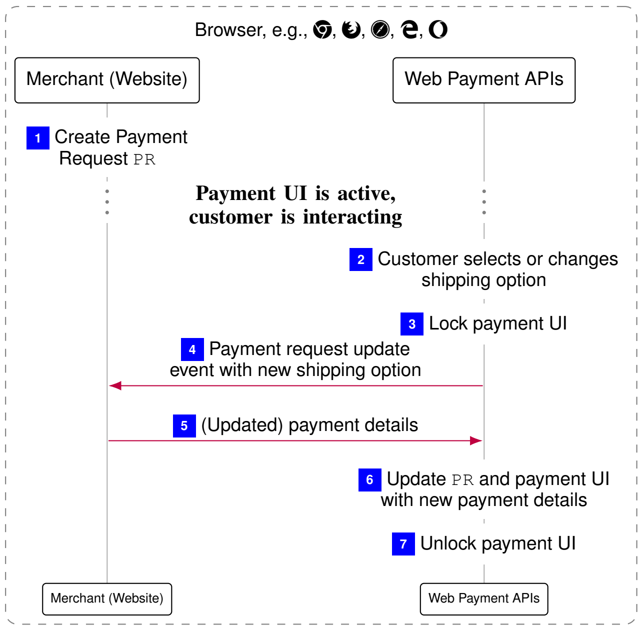

Browser, e.g.,
Merchant (Website)
Web Payment APIs
1 Create Payment Request PR
Payment UI is active, customer is interacting
2 Customer selects or changes shipping option
3 Lock payment UI
4 Payment request update event with new shipping option
5 (Updated) payment details
6 Update PR and payment UI with new payment details
7 Unlock payment UI
Merchant (Website)
Web Payment APIs

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

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 $\boxed{6}$ and $\boxed{14}$ of Figure 2). When the customer selects a shipping option $\boxed{2}$, the browser locks the payment UI $\boxed{3}$ 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 $\boxed{4}$. 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 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.

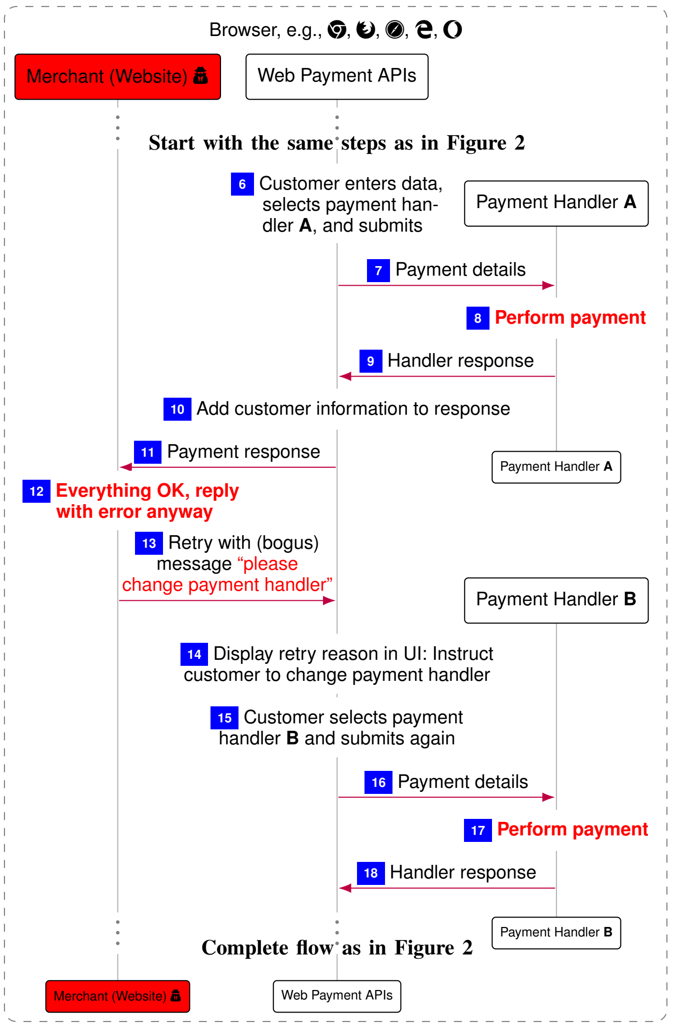
The mechanism to open a window is explicitly modeled in our browser and used by our generic payment handler to authenticate the customer (see Section V-B).

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 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



Browser, e.g.,
Merchant (Website)
Web Payment APIs
Start with the same steps as in Figure 2
6 Customer enters data, selects payment handler A, and submits
Payment Handler A
7 Payment details
8 Perform payment
9 Handler response
10 Add customer information to response
11 Payment response
Payment Handler A
12 Everything OK, reply with error anyway
13 Retry with (bogus) message "please change payment handler"
Payment Handler B
14 Display retry reason in UI: Instruct customer to change payment handler
15 Customer selects payment handler B and submits again
16 Payment details
17 Perform payment
18 Handler response
Complete flow as in Figure 2
Payment Handler B
Merchant (Website)
Web Payment APIs

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).

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 $\boxed{6}$ to $\boxed{11}$ as described in Section II-C.

Upon receiving the payment response, however, the merchant triggers a retry $\boxed{12}$. 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 $\boxed{13}$. The browser informs the customer by displaying the error message in its payment UI $\boxed{14}$. 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 $\boxed{15}$. The browser subsequently triggers the (new) payment handler $\boxed{16}$, which in turn performs the (second) payment $\boxed{17}$. 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.

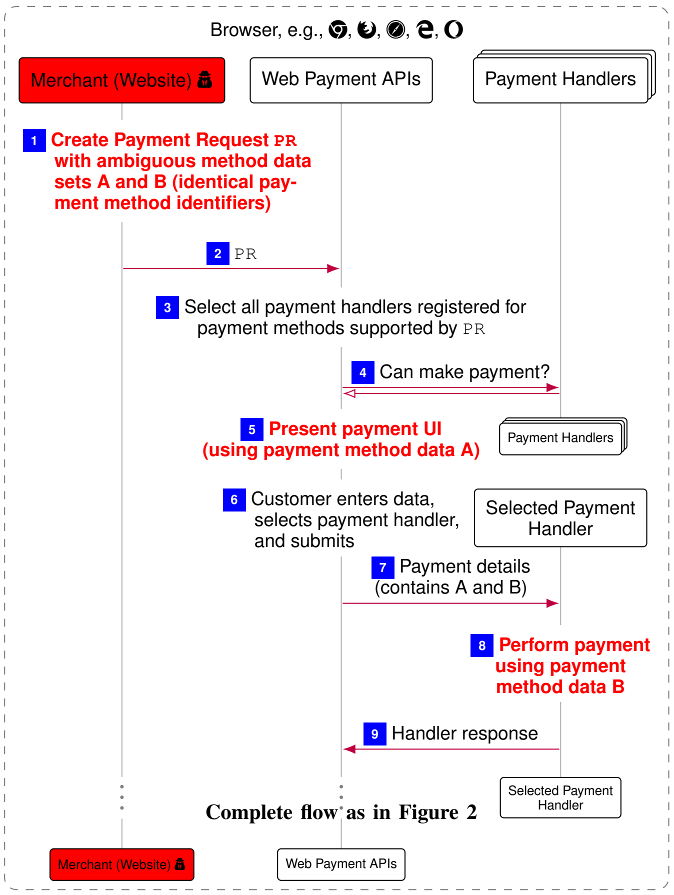
Implementation and Verification of the Attack. To test our attack, we implemented a dummy payment handler and merchant. At the time when we did the analysis and implementation, Google Chrome was the only browser available to us that already had full support for the WPA, thus we tested our attack against 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.

[6]We capture this property for the fixed model in Lemma 18 of our technical report [18].

[7]For the fixed WPA model (reflecting the fix in Line 152 of Algorithm 2), we show that a browser never hands the same payment request to two different payment handlers, see Lemma 15 of our technical report [18].



Browser, e.g.,
Merchant (Website)
Web Payment APIs
Payment Handlers
1 Create Payment Request PR with ambiguous method data sets A and B (identical payment method identifiers)
2 PR
3 Select all payment handlers registered for payment methods supported by PR
4 Can make payment?
5 Present payment UI (using payment method data A)
Payment Handlers
6 Customer enters data, selects payment handler, and submits
Selected Payment Handler
7 Payment details (contains A and B)
8 Perform payment using payment method data B
9 Handler response
Complete flow as in Figure 2
Selected Payment Handler
Merchant (Website)
Web Payment APIs

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

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 $\boxed{4}$. 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 $\boxed{6}$. 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 $\boxed{7}$. 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 $\boxed{8}$. The 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.

[8]For the fixed model (Lines 94 to 97 of Algorithm 1), we capture this property in Lemma 10 in our technical report [18].

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 [14]. 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 [15].

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.

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, $Q = (\mathrm{S}, \mathrm{E}, \mathrm{N}) \rightarrow (\mathrm{S}', \mathrm{E}', \mathrm{N}')$ 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 B.

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 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. We provide the remaining parts of the script API extension of the browser model in Appendix B, with full details in our technical report in [18].

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

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

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 $cookies$ := $\langle\{\langle c$.name, $c$.content.value$\rangle |$
        ↪ $c \in^{\langle\rangle} s'$.cookies $[s'.\overline{d}$.origin.host$] \wedge$
        ↪ $c$.content.httpOnly $= \bot \wedge$
        ↪ $(c$.content.secure $\implies$
          ↪ $(s'.\overline{d}$.origin.protocol $\equiv$ S$))\}\rangle$
4:    let $tlw \leftarrow s'$.windows such that
        ↪ $tlw$ is the top-level window containing $\overline{d}$
5:    let $sessionStorage$ :=
        ↪ $s'$.sessionStorage $[\langle s'.\overline{d}$.origin, $tlw$.nonce$\rangle]$
6:    let $localStorage$ := $s'$.localStorage $[s'.\overline{d}$.origin$]$
7:    let $secrets$ := $s'$.secrets $[s'.\overline{d}$.origin$]$
8:    let $R \leftarrow$ script$^{-1}(s'.\overline{d}$.script$)$
9:    let $in$ := $\langle tree$, $s'.\overline{d}$.nonce, $s'.\overline{d}$.scriptstate,
        ↪ $s'.\overline{d}$.scriptinputs, $cookies$,
        ↪ $localStorage$, $sessionStorage$, $s'$.ids, $secrets\rangle$
10:   let $state' \leftarrow \mathcal{T}_{\mathcal{N}}(V)$,
        ↪ $cookies' \leftarrow$ 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 $(in, out^{\lambda}) \in R$
11:   let $s'$.cookies $[s'.\overline{d}$.origin.host$]$ :=
        ↪ CookieMerge($s'$.cookies $[s'.\overline{d}$.origin.host$]$,
          ↪ $cookies'$)
12:   let $s'$.localStorage $[s'.\overline{d}$.origin$]$ := $localStorage'$
13:   let $s'$.sessionStorage $[\langle s'.\overline{d}$.origin, $tlw$.nonce$\rangle]$ :=
        ↪ $sessionStorage'$
14:   let $s'.\overline{d}$.scriptstate := $state'$
15:   switch $command$ do
      → In this excerpt, we focus on the extension for the WPA and
      refer to the technical report [18] for the full model. We use the
      same line numbers as in the technical report.
      ————— Extension with Payment APIs —————
91:     case ⟨PR_CREATE, $methodData$, $details$, $options\rangle$
92:       if $\neg(methodData \in$ 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 $paymentReq$ := ⟨PAYMENTREQUEST, $\nu_{14}$, $s'.\overline{d}$.nonce,
            ↪ $methodData$, $details$, $options$, $\langle\rangle$, $CR$, $\bot$, $\langle\rangle\rangle$
99:       let $transactionId$ := $\nu_{18}$
100:      let $s'$.paymentStorage$[\nu_{14}]$ :=
            ↪ $\langle paymentReq, \langle\rangle, \langle\rangle, transactionId\rangle$
101:      let $s'.\overline{d}$.scriptinputs
            ↪ := $s'.\overline{d}$.scriptinputs $+^{\langle\rangle}$ $paymentRequest$
102:      stop $\langle\rangle$, $s'$
103:    case ⟨PR_SHOW, $PRN$, $detailsUpdate\rangle$
104:      let $paymentReq$ :=
            ↪ $s'$.paymentStorage$[PRN]$.paymentReq
105:      if $paymentReq$.state $\neq CR$ then stop $\langle\rangle$, $s'$
106:      if $s'.\overline{w}.paymentRequestShowing = \top$ then
107:        let $s'$.paymentStorage$[PRN]$.paymentReq
              ↪ .state := $CL$
108:        stop $\langle\rangle$, $s'$
109:      let $s'$.paymentStorage$[PRN]$.paymentReq
            ↪ .state := $IN$
110:      let $s'.\overline{w}$.paymentRequestShowing := $\top$
111:      let $handlers$ := $\langle\rangle$
112:      for each $mds$ := $\langle pmi, receiver, paymentIdentifier\rangle \in$
            ↪ $paymentReq$.methodData do
113:        let $ph$ := GET_PAYMENT_HANDLERS($pmi$, $s'$)
114:        let $handlers$ := $handlers +^{\langle\rangle} ph$
115:        for each handler $\in ph$ do
116:          let $s'$.events := $s'$.events$+^{\langle\rangle}$
                ↪ ⟨CANMAKEPAYMENT, $handler$.nonce,
                ↪ $tlw$.origin, $s'.\overline{d}$.origin, $mds\rangle$
117:      let $handler \leftarrow handlers$
118:      let $s'$.paymentStorage$[PRN]$.handlerNonce :=
            ↪ $handler$.nonce
119:      if $detailsUpdate \neq \langle\rangle$ then
120:        let $s'$.paymentStorage$[PRN]$.paymentReq
              ↪ .updating := $\top$
121:        let $s'$.events := $s'$.events$+^{\langle\rangle}$
              ↪ ⟨PR_UPDATE_DETAILS, $PRN$, $detailsUpdate\rangle$
122:      let $s'$.events := $s'$.events$+^{\langle\rangle}$
            ↪ ⟨SUBMITPAYMENT, $PRN$, $handler$.nonce$\rangle$
123:      stop $\langle\rangle$, $s'$
      ————— ... —————
```

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.), with some more functions presented in Appendix B-B. We emphasize that our analysis, of course, covers the full definition of browsers, which is contained in our technical report in [18].

If the command created by the script is a PR_CREATE command (Lines 91ff.) with the arguments *methodData*, *details*,

[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).

and *options*, the browser creates a payment request with these values (see Steps $\boxed{1}$–$\boxed{2}$ in Section II-C and Section 3.1 of [13]). In this case, the browser verifies if *methodData* is well-formed (see Definition 13 in Appendix B-A) 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 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 ($s'.\overline{d}.\mathtt{nonce}$), the arguments of the command, as well as other state information of the payment request (see below and also Definition 5 in Appendix B-A). 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 $\boxed{2}$ in Section II-C and described in Section 3.3 of [13]).

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 our technical report [18] 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 $\boxed{7}$ 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 $\boxed{9}$ in Figure 2).

We note that our model of payment handlers does not send responses to `CanMakePaymentEvent`s. 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 [18]. 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.
- $\mathcal{S} = \{$script_merchant, script_payment_provider_index, script_default_payment_handler$\}$ contains the scripts of the model as sketched in Section V-B and formally defined in [18].
- script is a mapping from scripts to their string representation, where
  - script(script_merchant) = `script_merchant`
  - script(script_payment_provider_index) = `script_payment_provider_index`
  - script(script_default_payment_handler) = `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 $\mathit{paymentIntents}$. Each payment provider in turn stores (in its state) all performed transactions in a map called $\mathit{transactions}$. 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 $\mathit{transactionId}$ 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: $\mathsf{intents}(S, b, txId) := S(b).\mathtt{paymentIntents}[txId]$

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 $t \in S(pp).\mathtt{transactions}$ it holds true that:
If $b := \mathtt{ownerOfID}(t.\mathtt{sender}) \in B$ is a browser honest in S, then $\exists i \in \mathbb{N}$ such that $t.\mathtt{total} = \pi_i(\mathsf{intents}(S, b, t.\mathtt{txId})).\mathtt{details}.\mathtt{total} \wedge t.\mathtt{receiver} = \pi_1(\pi_i(\mathsf{intents}(S, b, t.\mathtt{txId})).\mathtt{methodData}).\mathtt{receiver}$.

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 $(txId, intents) \in S(b).\mathtt{paymentIntents}$, and with $PP_h = \{pp \in PP : S(pp).\mathtt{isCorrupted} = \bot\}$ being the set of payment providers that are honest in S, then $| \cup_{pp \in PP_h} \{t \in S(pp).\mathtt{transactions} \mid txId = t.\mathtt{txId} \wedge b = \mathsf{ownerOfID}(t.\mathtt{sender})\}| \leq 1$. If such a t exists, there exists a $pi \in intents$ such that $t.\mathtt{total} = pi.\mathtt{details}.\mathtt{total}$ and $t.\mathtt{receiver} = \pi_1(pi.\mathtt{methodData}).\mathtt{receiver}$.

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 the reader to our technical report [18] 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 the second lemma (Lemma 3) in Appendix A. To give an impression of the proof of Uniqueness of Payments, we provide a high-level proof sketch in Appendix C. Full proofs of all lemmas are provided in our technical report [18].

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 6) $preqEvent \in^{\langle\rangle} S(b).\mathtt{events}$, it holds true that $\exists pi$.

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

Proof: Let $preqEvent$ be a payment request event such that $preqEvent \in^{\langle\rangle} S(b).\mathtt{events}$, and b a browser honest in S. Initially, the `events` state entry of the browser is empty (per Definition 69 of [18]). An honest browser adds payment requests events to its `events` state entry only in Line 22 of Algorithm 3. 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., $preqEvent.\mathtt{txId}$ (see Line 20 and Line 25 of Algorithm 3).

Moreover, there exists a payment request $paymentReq$ such that $pi.\mathtt{details.total} = paymentReq.\mathtt{details.total}$ (Line 23 of Algorithm 3). The value of the total of $preqEvent$ is set to the value stored in $paymentReq$, i.e., $preqEvent.\mathtt{total} = paymentReq.\mathtt{details.total}$ (Line 10 and Line 20 of Algorithm 3). Therefore, we conclude that $pi.\mathtt{details.total} = preqEvent.\mathtt{total}$.

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 3, i.e., $\pi_1(preqEvent.\mathtt{methodData}).\mathtt{receiver} = \pi_1(pi.\mathtt{methodData}).\mathtt{receiver}$. ■

B. Proof of Intended Payments

The proof of the Intended Payments property uses Lemma 3, which we give in Appendix A. Informally, Lemma 3 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., $t \in S(pp).transactions$.

We apply Lemma 3 and conclude that, if $b := \mathtt{ownerOfID}(t.\mathtt{sender}) \in B$ is a browser honest in S, then $\exists\ preqEvent \in^{\langle\rangle} S(b).\mathtt{events}$ such that

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

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

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

and, in particular,

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

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 $\mathsf{DY}^\star$ [7], a new approach for the modular symbolic security analysis of protocol code written in the $\mathsf{F}^\star$ 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] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

[18] Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, N. Wenzler, and T. Würtele. *A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification*. Cryptology ePrint Archive, Report 2021/1012. https://ia.cr/2021/1012. 2021.

[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
Helper Lemmas

In this section, we provide basic lemmas that are necessary for proving the security theorem. Lemma 3 (see below) is directly used for proving the Intended Payments property (see Section VII-B). Lemma 4 is used for proving the Uniqueness of Payments property (see the high-level proof sketch in Appendix C). The remaining lemmas shown here (Lemma 5 and Lemma 6) capture basic properties needed for the proofs of both security properties.

We highlight that this is a small selection of the lemmas needed for the full proof and that we omit their proofs for brevity. We provide all lemmas with their full proofs in the technical report [18].

Transaction Values Originate from Payment Request Event. Lemma 3 relates transactions stored at an honest payment provider to payment request events stored at a browser. More precisely, the lemma states that for every run of a WPA system in $\mathcal{WPAPI}$ and every reachable configuration, and for every payment provider server pp honest in S, if pp stores a transaction t, then the browser that manages the identity of the sender of the transaction $t.\mathtt{sender}$ stores a payment request event that contains the values used for the transaction.

Lemma 3 (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 $t \in S(pp).\mathit{transactions}$ the following holds true:
If $b := \mathtt{ownerOfID}(t.\mathtt{sender}) \in B$ is a browser honest in S, then $\exists$ $\mathit{preqEvent} \in^{\langle\rangle} S(b).\mathtt{events}$ such that, with $pi = \pi_1(\mathit{preqEvent}.\mathtt{methodData}).\mathtt{paymentIdentifier}$, it holds true that

$$\pi_1(\mathit{preqEvent}) = \mathtt{PAYMENTREQUESTEVENT} \wedge$$
$$\mathit{preqEvent}.\mathtt{txId} = t.\mathtt{txId} \wedge$$
$$\mathit{preqEvent}.\mathtt{total} = t.\mathtt{total} \wedge$$
$$\pi_1(\mathit{preqEvent}.\mathtt{methodData}).\mathtt{receiver} = t.\mathtt{receiver} \wedge$$
$$S(pp).\mathtt{transactions}[\langle pi\rangle] = t$$

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.

Same Payment Identifier if Events have same Transaction Identifier. Lemma 4 captures that for every run of a WPA system in $\mathcal{WPAPI}$ and every reachable configuration, and for every browser b honest in S, for every two payment request events pre_1, pre_2 stored by the browser, if both events have the same transaction identifier, then they have the same payment identifier in the first method data element.

Lemma 4 (Same payment identifier if same transaction identifier). For every run ρ of a WPA system in $\mathcal{WPAPI}$, every configuration (S, E, N) in ρ, every browser $b \in B$ honest in S, every payment request events pre_1, pre_2 $\in^{\langle\rangle} S(b).\mathtt{events}$ with $\pi_1(pre_1) = \mathtt{PAYMENTREQUESTEVENT}$ and $\pi_1(pre_2) = \mathtt{PAYMENTREQUESTEVENT}$, it holds true that

$$(pre_1.\mathtt{txId} = pre_2.\mathtt{txId}) \Rightarrow$$
$$(\pi_1(pre_1.\mathtt{methodData}).\mathtt{paymentIdentifier} =$$
$$\pi_1(pre_2.\mathtt{methodData}).\mathtt{paymentIdentifier}).$$

Credentials do not leak. Lemma 5 states that for every run of a WPA system in $\mathcal{WPAPI}$ and every reachable configuration, every user identity of an honest payment provider (where the identity is managed by an honest browser), the user credentials of the identity (for authenticating at the payment provider) is only derivable by the browser and the payment provider, and by no other process, in particular, not by the network attacker.

Lemma 5 (Credentials do not leak). For every run ρ of a WPA system in $\mathcal{WPAPI}$, every configuration (S, E, N) in ρ, for every browsers $b \in B$ honest in S, every $id \in b.\mathtt{ids}$ with $pp = \mathtt{governor}(id)$ and pp honest in S, it holds true that $\forall p \in \mathcal{W}\setminus\{b, pp\} : \mathtt{secretOfID}(id) \notin d_\emptyset(S(p))$.[11]

Authorization Tokens do not leak. Lemma 6 states that for every run of a WPA system in $\mathcal{WPAPI}$ and every reachable configuration, a token stored in the state of an honest payment provider for an identity managed by an honest server cannot be derived by any other process except for the browser and the payment provider.

Lemma 6 (Authorization tokens do not leak). For every run ρ of a WPA system in $\mathcal{WPAPI}$, every configuration (S, E, N) in ρ, every payment provider server $pp \in PP$ honest in S, every $token \in S(pp).\mathtt{tokens}$ with id such that $S(pp).\mathtt{tokens}[id] = token$ and $b \in B$ such that $b = \mathtt{ownerOfID}(id)$ and b honest in S, it holds true that $\forall p \in \mathcal{W}\setminus\{b, pp\} : token \notin d_\emptyset(S(p))$.

Appendix B
Excerpt of the WPA Model

We introduce here some main features of the WPA model, including data structures of important events and objects (B-A), the remaining extension of the script API already presented in Algorithm 1 (B-B), and processing events (B-C) within the browser. The full model is provided in our technical report [18].

A. Data Structures

Payment Relevant Objects. In Definitions 5 to 8, we provide the data structures of simple objects used within the browser model.

[11] $d_\emptyset(M)$ denotes the set of all messages that can be derived from the set M of messages (in the usual Dolev-Yao style).

Definition 5 (Payment Request). A Payment Request is a term of the form $\langle$PAYMENTREQUEST, $\mathit{paymentRequestNonce}$, $\mathit{documentnonce}$, $\mathit{methodData}$, $\mathit{details}$, $\mathit{options}$, state, $\mathit{updating}\rangle$, where $\mathit{paymentRequestNonce} \in \mathcal{N}$, $\mathit{documentnonce} \in \mathcal{N}$, $\mathit{methodData} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{details} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{options} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{state} \in \{CR, IN, CL\}$ (corresponding to the states: **Cr**eated, **In**teractive and **Cl**osed), and $\mathit{updating} \in \{\top, \bot\}$.

Definition 6 (Payment Request Event). A Payment Request Event is a term of the form $\langle$PAYMENTREQUESTEVENT, $\mathit{preqEventNonce}$, $\mathit{paymentRequestNonce}$, $\mathit{handlerNonce}$, $\mathit{methodData}$, total, $\mathit{modifiers}$, $\mathit{instrumentKey}$, $\mathit{requestBillingAddress}$, $\mathit{transactionId}\rangle$, where $\mathit{preqEventNonce} \in \mathcal{N}$, $\mathit{paymentRequestNonce} \in \mathcal{N}$, $\mathit{handlerNonce} \in \mathcal{N}$, $\mathit{methodData} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{total} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{modifiers} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{instrumentKey} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{requestBillingAddress} \in \{\bot, \top\}$, and $\mathit{transactionId} \in \mathcal{N}$.

Definition 7 (Payment Handler Response). A Payment Handler Response is a term $\langle$PAYMENTHANDLERRESPONSE, $\mathit{paymentRequestNonce}$, $\mathit{handlerNonce}$, $\mathit{methodName}$, $\mathit{details}\rangle$, where $\mathit{paymentRequestNonce} \in \mathcal{N}$, $\mathit{handlerNonce} \in \mathcal{N}$, $\mathit{methodName} \in \mathcal{T}_{\mathcal{N}}$, and $\mathit{details} \in \mathcal{T}_{\mathcal{N}}$.

Definition 8 (Payment Response). A Payment Response is a term of the form $\langle$PAYMENTRESPONSE, $\mathit{paymentResponseNonce}$, $\mathit{paymentRequestNonce}$, $\mathit{handlerNonce}$, $\mathit{methodName}$, $\mathit{details}$, $\mathit{shippingAddress}$, $\mathit{shippingOption}$, $\mathit{payerInfo}$, $\mathit{complete}\rangle$, where $\mathit{paymentResponseNonce} \in \mathcal{N}$, $\mathit{paymentRequestNonce} \in \mathcal{N}$, $\mathit{handlerNonce} \in \mathcal{N}$, $\mathit{methodName} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{details} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{shippingAddress} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{shippingOption} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{payerInfo} \in \mathcal{T}_{\mathcal{N}}$, and $\mathit{complete} \in \{\bot, \top\}$.

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 9 (Payment Storage). A $\mathit{paymentStorage} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary, where for a nonce $PRN \in \mathcal{N}$, the term $\mathit{paymentStorage}[PRN]$ has the following structure: $\langle \mathit{paymentRequest}, \mathit{paymentRequestEvents}, \mathit{paymentResponses}, \mathit{transactionId}, \mathit{handlerNonce}\rangle$. $\mathit{paymentRequest} \in \mathcal{T}_{\mathcal{N}}$ is a term as defined in Definition 5. $\mathit{paymentRequestEvents} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary from nonces to payment request events (see also Definition 6). $\mathit{paymentResponses} \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary from nonces to payment responses (see also Definition 8). $\mathit{transactionId} \in \mathcal{N}$ and $\mathit{handlerNonce} \in \mathcal{N}$ are nonces.

Transactions. In our model, a payment provider server has a state entry for storing all transactions (more precisely, a sequence of transactions). In the following, we define the structure of this transaction state entry.

Definition 10 (Transaction). $\mathit{transactions} \in [\mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}]$ is a dictionary, where for a term $\mathit{key} \in \mathcal{T}_{\mathcal{N}}$, the transaction $\mathit{transactions}[\mathit{key}]$ has the following structure: $\langle \mathit{sender}, \mathit{receiver}, \mathit{total}, \mathit{transactionId}\rangle$, where $\mathit{sender}, \mathit{receiver}, \mathit{total} \in \mathcal{T}_{\mathcal{N}}$, and $\mathit{transactionId} \in \mathcal{N}$.

Service Workers. In the following, we give the definition of a service worker registration. We call the set of all Service Worker Registrations $\mathit{ServiceWorkerRegistrations}$.

Definition 11 (Service Worker Registration). A Service Worker Registration is defined through a term of the form: $\langle \mathit{nonce}, \mathit{scope}, \mathit{script}, \mathit{scriptinputs}, \mathit{scriptstate}, \mathit{paymentManager}, \mathit{trusted}\rangle$, where $\mathit{nonce} \in \mathcal{N}$, $\mathit{scope} \in$ URLs, $\mathit{script} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{scriptinputs} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{scriptstate} \in \mathcal{T}_{\mathcal{N}}$, $\mathit{paymentManager} \in \mathit{PaymentManagers}$, and $\mathit{trusted} \in \{\bot, \top\}$.

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

Definition 12 (Payment Method Identifier). A Payment Method Identifier is a URL for which it holds true that $\mathit{protocol} = S$. PaymentMethodIdentifiers is the set of all possible Payment Method Identifiers.

Definition 13 (Payment Method Data). A payment method data term is a term consisting of a sequence of terms $\langle x_1, x_2, \ldots\rangle$, where each x_i is a term $\langle \mathit{pmi}, \mathit{receiver}, \mathit{paymentIdentifier}\rangle$ with $\mathit{pmi} \in$ PaymentMethodIdentifiers, $\mathit{receiver} \in \mathcal{T}_{\mathcal{N}}$, and $\mathit{paymentIdentifier} \in \mathbb{S}$. Furthermore, we require for all x_i, x_j in the sequence that x_i.paymentIdentifier $=$ x_j.paymentIdentifier. The set of all possible payment method data terms is referenced by MethodDatas.

B. Script API Extension

Algorithm 2 continues some of the cases of the extension of the RUNSCRIPT function that are omitted in Algorithm 1. In particular, these cases show how the browser processes script commands for determining whether there is a payment handler that supports one of the payment methods of a payment request (PR_CANMAKEPAYMENT), aborting a payment (PR_ABORT), completing a payment (PRESS_COMPLETE), retrying a payment (PRESS_RETRY), and updating payment details (PR_UPDATE_DETAILS).

C. Event Processing

Algorithm 3 describes the PROCESSEVENT function that processes events within the browser. The function takes the current browser's state, the current active window and an event (which can be chosen from a pool of waiting events non-deterministically), and outputs a new state of the browser, which can contain new events. The events CanMakePayment, PaymentRequestEvent, and PaymentResponse are simply processed by transmitting the relevant event to the corresponding payment handler. The SubmitPayment event triggers a process to select the payment handler responsible to handle resulting PaymentRequestEvent. For the case of the event PaymentHandleResponse, information of corresponding payment request is integrated into a Payment Response object and the corresponding PaymentResponse event is submitted to the event set.

Algorithm 2 Web Browser Model: Execute a script.

```
1:   function RUNSCRIPT($\overline{w}$, $\overline{d}$, $s'$)
       → In this excerpt, we continue Algorithm 1, focussing on the
       extension for the WPA and refer to the technical report [18] for
       the full model. We use the same line numbering as in the
       technical report.
15:    switch $command$ do
     ———————————————— . . . ————————————————
124:       case ⟨PR_CANMAKEPAYMENT, $PRN$⟩
125:         let $paymentReq$ :=
               ↪ $s'$.paymentStorage[$PRN$].paymentReq
126:         if $paymentReq$.state $\neq CR$ then stop ⟨⟩, $s'$
127:         let $handlers$ := ⟨⟩
128:         for each ⟨$pmi$, $recv$, $paymentId$⟩ ∈
           ↪ $paymentReq$.methodData do
129:           let $handlers$ := $handlers +^{\langle\rangle}$
               ↪ GET_PAYMENT_HANDLERS($pmi$, $s'$)
130:         if $handlers \neq \langle\rangle$ then
131:           let $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs
               ↪ $+^{\langle\rangle}$ ⟨CANMAKEPAYMENTRESPONSE, $PRN$, $\top$⟩
               ↪    → Let script know that handler is available
132:         else
133:           let $s'.\overline{d}$.scriptinputs := $s'.\overline{d}$.scriptinputs
               ↪ $+^{\langle\rangle}$ ⟨CANMAKEPAYMENTRESPONSE, $PRN$, $\bot$⟩
               ↪    → Let script know that handler is not available
134:         stop ⟨⟩, $s'$
135:       case ⟨PR_ABORT, $PRN$⟩
136:         let $paymentReq$ :=
               ↪ $s'$.paymentStorage[$PRN$].paymentReq
137:         if $paymentReq$.state $\neq IN$ then stop ⟨⟩, $s'$
138:         if $s'$.paymentStorage[$PRN$]
               ↪ .paymentResponses $\neq$ ⟨⟩ then stop ⟨⟩, $s'$
139:         let $s'$.paymentStorage[$PRN$].paymentReq
               ↪ .state := $CL$
140:         let $s'.\overline{w}$.paymentRequestShowing := $\bot$
141:         stop ⟨⟩, $s'$
142:       case ⟨PRES_COMPLETE, $PRN$, $paymRespNonce$⟩
143:         let $paymentResp$ := ($s'$.paymentStorage[$PRN$]
               ↪ .paymentResponses)[$paymRespNonce$]
               ↪    → Retrieve payment response
144:         if $paymentResp.complete \equiv \top$ then stop ⟨⟩, $s'$
145:         let ($s'$.paymentStorage[$PRN$].paymentResponses)
               ↪ [$paymRespNonce$].complete := $\top$
146:         let $s'.\overline{w}.paymentRequestShowing := \bot$
147:         stop ⟨⟩, $s'$
148:       case ⟨PRES_RETRY, $PRN$, $paymRespNonce$, $errFields$⟩
149:         let $paymentResp$ := ($s'$.paymentStorage[$PRN$]
               ↪ .paymentResponses)[$paymRespNonce$]
               ↪    → Get response from state using the two nonces
150:         if $paymentResp$.complete $\equiv \top$ then stop ⟨⟩, $s'$
151:         let $s'$.paymentStorage[$PRN$].paymentReq
               ↪ .state := $IN$
152:         let $handlerNonce$ :=
               ↪ $s'$.paymentStorage[$PRN$].handlerNonce
               → Fix: No change of payment handler during retry
153:         let $s'$.events := $s'$.events$+^{\langle\rangle}$
               ↪ ⟨SUBMITPAYMENT, $PRN$, $handlerNonce$⟩
154:         stop ⟨⟩, $s'$
       → For brevity, we omit here the cases PR_GET_PREQ and
       PR_GET_PRESP
164:       case ⟨PR_UPDATE_DETAILS, $PRN$, $details$⟩ → According
       to the specification, updateDetails can only occur as a reaction
       to a PaymentRequestUpdateEvent. This is simplified in the
       model.
165:         let $paymentReq$ :=
               ↪ $s'$.paymentStorage[$PRN$].paymentReq
166:         if $paymentReq$.state $\neq IN$ then stop ⟨⟩, $s'$
167:         let $s'$.paymentStorage[$PRN$].paymentReq
               ↪ .updating := $\top$
168:         let $s'$.events := $s'$.events$+^{\langle\rangle}$
               ↪ ⟨PR_UPDATE_DETAILS, $PRN$, $details$⟩
169:         stop ⟨⟩, $s'$
170:       case else stop ⟨⟩, $s'$
```

APPENDIX C
UNIQUENESS OF PAYMENTS

In the following, we give a high-level overview of the proof of the Uniqueness of Payments property. We refer to the technical report in [18] for the formal proof.

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. In the full proof in [18], 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.

Moreover, 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 2). 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.

Algorithm 3 Web Browser Model: Process an event.

```
1: function PROCESSEVENT($e$, $\overline{w}$, $s'$)
2:   switch $e$ do
3:     case $\langle$CANMAKEPAYMENT, $handlerNonce$, $topOrigin$,
         ↪ $paymentRequestOrigin$, $methodData\rangle$
4:       call DELIVER_TO_DOC($handlerNonce, e, s'$)
         → The browser non-det. 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.
   ——— User accepts the payment request algorithm ———
5:     case $\langle$SUBMITPAYMENT, $PRN$, $handlerNonce\rangle$
6:       let $paymentReq$ :=
           ↪ $s$.paymentStorage[$PRN$].paymentReq
7:       if $paymentReq$.updating $= \top$ then stop $\langle\rangle$, $s'$
8:       if $paymentReq.state \neq IN$ then stop $\langle\rangle$, $s'$
9:       let $handler \leftarrow s'$.serviceWorkers such that
           ↪ $handler$.nonce $\equiv$ $handlerNonce$
             ↪ if possible; otherwise stop $\langle\rangle, s'$
10:      let $total$ := $paymentReq$.details.total
11:      let $modifiers$ := $paymentReq$.details.modifiers
           ↪   → Abstraction: pass all modifiers
12:      let $requestBillingAddress$ :=
           ↪ $paymentReq$.options.requestBillingAddress
13:      let $instrument \leftarrow handler$.paymentManager such that
           ↪ $\exists mds \in^{\langle\rangle}$ $paymentRequest$.methodData $\wedge$
           ↪ $mds$.pmi $=$ $instrument$.enabledMethods
           ↪ if possible; otherwise stop $\langle\rangle, s'$
14:      let $instrumentKey$ := $instrument$.instrumentKey
15:      let $methodData$ := $\langle\rangle$
16:      for each $mds := \langle pmi, recv, paymentId\rangle \in$
           ↪ $paymentReq$.methodData do
17:        if $pmi \equiv instrument$.enabledMethods then
18:          let $methodData$ := $methodData +^{\langle\rangle}$ $mds$
19:      let $transactionId$ :=
           ↪ $s'.paymentStorage[PRN]$.transactionId
20:      let $pre$ := $\langle$PAYMENTREQUESTEVENT, $\nu_{16}$, PRN,
           ↪ $handler$.nonce, $methodData$, $total$,
           ↪ $modifiers$, $instrumentKey$,
           ↪ $requestBillingAddress$, $transactionId\rangle$
21:      let $(s'.paymentStorage[PRN]$
           ↪ .paymentRequestEvents)[$\nu_{16}$] := $pre$
             ↪   → Make event available for later use
22:      let $s'$.events := $s'$.events $+^{\langle\rangle}$ $pre$
23:      let $paymentIntent$ := $paymentReq$
24:      let $paymentIntent$.methodData := $methodData$
25:      let $s'$.paymentIntents[$transactionId$] :=
           ↪ $s'$.paymentIntents[$transactionId$]$+^{\langle\rangle}$
           ↪ $paymentIntent$
           ↪   → New payment intent is added into the list of
                 payment intents indexed by $transactionId$
26:      stop $\langle\rangle$, $s'$
   —— Payment Handler is selected and processes request ——
27:    case
       $\langle$PAYMENTREQUESTEVENT, paymentRequestEventNonce,
         ↪ PRN, handlerNonce, $methodData$, $total$, $modifiers$,
         ↪ $instrumentKey$, $requestBillingAddr$, $transactionId\rangle$
28:      call DELIVER_TO_DOC($handlerNonce, e, s'$)
   — Payment handler's response is merged with relevant data —
29:    case $\langle$PAYMENTHANDLERRESPONSE,
         ↪ $paymentReqEventNonce$, $PRN$, $handlerNonce$,
         ↪ $methodName$, $details\rangle$
30:      let $paymentReq$ :=
           ↪ $s'.paymentStorage[PRN]$.paymentReq
31:      let $paymentRequestEvent$ := ($s'.paymentStorage$
           ↪ [$PRN$].paymentRequestEvents)
           ↪ [$paymentReqEventNonce$]
32:      if $paymentRequestEvent$
           ↪ .methodData $|$ $\langle methodName, *, *\rangle \equiv \langle\rangle$ then
33:        stop $\langle\rangle, s'$   → Method not accepted by merchant
34:      if $paymentReq.updating = \top$ then stop $\langle\rangle$, $s'$
35:      if $paymentReq.state \neq IN$ then stop $\langle\rangle$, $s'$
36:      let $shippingAddress$ := $\langle\rangle$
37:      let $shippingOption$ := $\langle\rangle$
38:      if $paymentReq$.options.requestShipping $= \top$ then
39:        let $shippingAddress$ := $\nu_{17}$
40:        let $shippingOption \leftarrow$
             ↪ $paymentReq$.details.shippingOptions
41:      let $payerInfo$ := $\langle\rangle$
42:      if $paymentReq$.options.requestPayerInfo $= \top$ then
43:        $payerInfo = \nu_{15}$   → Any payer specific info (phone,
                                    name, email)
44:      let $responseNonce$ := $\nu_{16}$
45:      let $response$ := $\langle$PAYMENTRESPONSE, $responseNonce$,
           ↪ PRN, $handlerNonce$, $methodName$, $details$,
           ↪ $shippingAddress$, $shippingOption$, $payerInfo$, $\perp\rangle$
46:      let $(s'.paymentStorage[PRN]$.paymentResponses)
           ↪ [$responseNonce$] := $response$
47:      let $s'$.events := $s'$.events $+^{\langle\rangle}$ $response$
           ↪   → Create PAYMENTRESPONSE Event
48:      stop $\langle\rangle$, $s'$
   —— Payment Response is submitted to script in user agent ——
49:    case
       $\langle$PAYMENTRESPONSE, $responseNonce$, $PRN$, $handlerNonce$,
         ↪ $methodName$, $details$, $shippingAddr$, $shippingOpt$,
         ↪ $payerInfo$, $complete\rangle$
50:      let $requestingWinNonce$
           ↪ := $s'.paymentStorage[PRN]$
           ↪ .paymentReq.docnonce
51:      call DELIVER_TO_DOC($requestingWinNonce, e, s'$)
   ——————— PR update details ———————
52:    case $\langle$PR_UPDATE_DETAILS, PRN, $details\rangle$
53:      let $s'.paymentStorage[PRN]$.paymentReq
           ↪ .details := $details$
54:      let $s'.paymentStorage[PRN]$.paymentReq
           ↪ .updating := $\perp$
55:      stop $\langle\rangle$, $s'$
```

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 4, 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.