USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

When Authorization Loses Its Meaning: Breaking and Fixing Third-Party Online Payments

Yongkang Xiao, Jing Chen, Min Shi, Kun He, Qiyi Deng, and Ruiying Du, *Wuhan University*

https://www.usenix.org/conference/usenixsecurity26/presentation/xiao

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

When Authorization Loses Its Meaning: Breaking and Fixing Third-Party Online Payments

Yongkang Xiao, Jing Chen*, Min Shi*, Kun He, Qiyi Deng, Ruiying Du
School of Cyber Science and Engineering, Wuhan University
{xiaoyongkang, chenjing, itachi, hekun, qiyideng, duraying}@whu.edu.cn

Abstract

Third-party online payment systems, such as Alipay and PSP-B, constitute critical infrastructure for modern e-commerce. However, their security rests on the unrealistic assumption of fully trusted communication channels. While prior studies have identified isolated vulnerabilities, a systematic formal analysis of payment protocol security remains absent. This paper presents formal security models for six third-party payment protocols, spanning three major payment scenarios and two dominant payment service providers. Our analysis reveals a fundamental design flaw: whenever channel integrity is compromised between the merchant client, merchant server, or payment system, order tampering attacks become feasible. We validate this threat on Android, where over 20% of tested merchant applications allow order tampering through implicit Intent hijacking. To mitigate this threat, we propose user-side order authentication, where per-user-merchant key pairs cryptographically bind consent to order semantics. Formal verification demonstrates its resilience against identified attacks under weak channel assumptions. By bridging formal methods and empirical analysis, this work offers actionable guidance for standardizing secure payment protocols.

1 Introduction

Third-party payment systems have become a core infrastructure of modern e-commerce and mobile payments. Particularly, third-party payment services in China have evolved into a foundational component of the digital economy, with both market scale and user penetration continuing to grow. According to the latest industry report by iResearch [3], the transaction volume of China's personal third-party payment market has reached 67 trillion Chinese yuan (approximately 9.6 trillion US dollars) in 2025, with online payments accounting for more than 50% of the total. Among all third-party payment platforms, Alipay [1] and PSP-B are the two dominant payment service providers in China, consistently occupying the top tier of the third-party payment market.

A typical third-party payment ecosystem comprises the User (*U*), the Merchant Frontend (*MF*), the Merchant backend Server (*MS*), and the Payment System (*PS*). This ecosystem supports multiple payment scenarios, including in-app payments, mobile web payments, and QR-code-based payments. Within this ecosystem, payment service providers can ensure the security of communications within their own payment systems, such as the communication between the payment application (*PS-App*) and the payment server (*PS-Server*). Additionally, they can protect the channel security between *MS* and *PS-Server* by restricting communication to HTTPS. However, payment service providers cannot guarantee the security of communications between external entities.

Unfortunately, numerous empirical studies have shown that insecure or misconfigured channels have led to various categories of security vulnerabilities in third-party payment workflows. Prior research has identifyed at least three types of attacks associated with untrusted channels: (1) *Order Injection and Tampering Over Insecure Channels*: Attackers can replace or replay order, causing users to unknowingly pay for incorrect orders [36, 41]; (2) *Misuse of Signatures and Parameter Structures*: Attackers can alter payment parameters without invalidating the signature by exploiting flawed signature designs [21]; (3) *Payment Result Forgery and UI Confusion*: Attackers can inject forged "payment success" messages or leverage UI overlay attacks, H5 redirection, and script injection to mislead merchants or users about the transaction status, enabling free purchases or misdirected payments [38, 41]. These studies reveal a critical issue: the cryptographic mechanisms employed by current third-party online payment protocols are insufficient to withstand threats arising from insecure external channels. From the perspective of payment service providers, *adopting weaker security assumptions for external channels reflects the relities of modern payment environments better*. This observation *motivates us to analyze the security of third-party online payment protocols under weakened external channel security assumptions*.

*Corresponding authors.

In this paper, we establish comprehensive symbolic models for third-party online payment protocols. Due to the absence of an industry-wide standard for online payments, we examined publicly available developer documentation of Alipay [2] and PSP-B, abstracting six representative payment protocol models that cover three core payment scenarios. Our models faithfully reflect the design details of actual payment processes and cover the complete workflow, from order placement to payment completion. From these six protocols, we derive two high-level design patterns and investigate their common security issues. To realistically capture the capabilities of adversaries, we explicitly model adversary's behaviors. Particularly, we grant the adversary legitimate access to merchant platforms, enabling them to create orders and obtain payment parameters.

We formally define a set of security properties for third-party online payment protocols across three dimensions: order, authorization, and result. We verify these properties in the six real-world payment protocols under weakened channel security assumptions using Tamarin-Prover [23]. Our analysis reveals a pervasive yet underexplored design anti-pattern: *mainstream high-level payment patterns generally lack mechanisms that bind user payment authorization to the intended order.* Consequently, once the communication channel between the user and the merchant is compromised, an adversary can induce the user to authorize any order.

Motivated by the findings from our formal analysis, we identify a new variant of the order tampering attack. We observe that many merchant applications on Android improperly use implicit intents to interact with payment applications. This misuse exposes the transmission of payment parameters to interception by malicious applications. We develop a proof-of-concept attack application that exploit this vulnerability and successfully perform order tampering attacks in real-world payment environments. To evaluate the prevalence and severity of this issue, we conduct a measurement study of 461 popular applications spanning multiple domains, including online shopping, online education, entertainment, and gaming. Our results show that more than 25% of the sampled applications are vulnerable. These applications collectively account for over 200 billion downloads, potentially affecting billions of users. We responsibly disclosed our findings to the security teams at Alipay and PSP-B. We received acknowledgment from them, and at the time of writing, remediation efforts were actively underway. Additionally, we notified the developers of 41 vulnerable merchant applications identified in our study and provided concrete security recommendations to help mitigate this risk.

To address these design flaws, we propose a deployable payment pattern which incorporates a user-side order authentication mechanism to establish a binding between user's authorization and the intended order. Under the weakened channel assumptions, we formally verify this pattern and show that it satisfies all the defined security properties.

Contributions. We make the following contributions:

1. **Symbolic verification of payment protocols.** We summarize the mainstream design patterns of third-party online payment protocols and analyze six representative real-world third-party online payment protocols using symbolic models. Our formal analysis results reveal that, under realistic adversary and channel assumptions, the high-level design of all analyzed protocols is flawed and vulnerable to order tampering attacks.

2. **Disclosure of a new order tampering attack.** Based on our formal analysis results, we identified a new variant of order tampering attacks and successfully demonstrated it in a real-world payment environment. Our further measurement study shows that this vulnerability affects more than 25% of real-world applications.

3. **Proposal of a formal verified countermeasure.** We propose a new payment protocol as a countermeasure, which introduces a user-side order authentication mechanism to address the identified design flaws. Formal verification results demonstrate that the proposed protocol can maintain security, even under weakened channel assumptions.

2 Background

In this section, we first introduce the typical ecosystem architecture and payment scenarios of third-party online payments. Then, we abstract two main high-level payment models, and finally introduce Tamarin-Prover [23].

2.1 Third-Party Online Payment Ecosystem

Currentlly, there is no universally adopted regulatory standard governing third-party online payments, and payment providers typically rely on proprietary protocols and implementations. By reviewing official documentation and developer guides [2], alongside an analysis of payment workflows, we summarize the typical ecosystem architecture of third-party online payment systems. A typical third-party online payment ecosystem involves four main participants:

1. ***User (U):*** An individual who initiates and authorizes payments for goods or services.

2. ***Merchant Frontend (MF):*** The interface through which the user interacts to select goods and initiate payments, typically implemented as a website or mobile application.

3. ***Merchant Server (MS):*** The server-side component of the merchant that processes orders and communicates with payment providers.

4. ***Third-Party Payment System (PS):*** A payment transaction handling system provided by a payment service provider.

The *PS* comprises a payment application (*PS-App*) and a payment server (*PS-Server*). Since the internal interaction mechanisms are proprietary and the communication occurs over private, secure channels, these two components cannot be individually inspected. Therefore, we treat the *PS* as a unified black-box system.

As illustrated in Figure 1, a third-party payment process can be abstracted into the following four phases:

i. ***Order Phase:*** The user *U* selects *goods* and chooses a payment provider on the *MF*. The *MF* sends the information of *goods* and the selected payment provider to the *MS*.

ii. ***Prepayment Phase:*** Upon receiving the information, the *MS* generates the corresponding payment parameters *params* and returns them to the *MF*, which subsequently forwards *params* to the corresponding *PS*.

iii. ***Authorization Phase:*** The *PS* displays the order information, including product description, payment amount, and payee's account, to *U* via the *PS-App* for confirmation. After verifying the order details, *U* enters the payment *password* in the *PS-App* to authorize the transaction.

iv. ***Notification Phase:*** Upon successful authorization, the *PS-Server* signs the payment result using its private key and sends the signed result to the *MS*. Simultaneously, the *PS-App* displays the payment result to *U*.

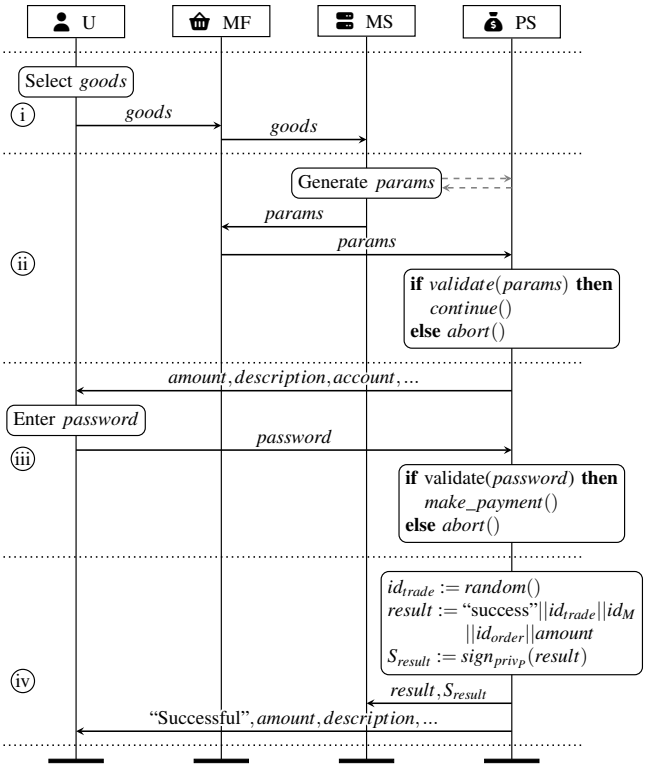


Figure 1: An Overview of a Typical Third-Party Online Payment. Dashed arrows indicate optional messages, depending on the specific payment pattern. Notation: $validate(p)$ denotes checking whether p is valid; $continue()$ denotes continuing the execution of the protocol; $abort()$ denotes terminating the protocol; $make\_payment()$ denotes the fund transfer operation; $random()$ denotes generating a random number; and $sign_k(m)$ denotes signing message m with key k.

2.2 Payment Scenarios

The *PS* obtains payment parameters from the *MF* via *PS-App* installed on the *U*'s mobile device. However, the communication methods between the *PS-App* and the *MF* differ depending on implementation of the *MF*, which may include native mobile applications, mobile web pages, and desktop web pages. These distinct forms result in different payment scenarios, prompting payment providers to define specific interaction methods between the *MF* and the *PS*. We introduce three common payment scenarios below:

In-app Payment. In this scenario, payment parameters are obtained within native mobile applications. After the user selects a good, the *MF* invokes the *PS-App* through the Software Development Kit (SDK) provided by the payment provider to transmit the required payment parameters to the *PS*.

Mobile Web Payment. In this scenario, the user interacts with the merchant through a mobile browser. Similar to in-app payment, the merchant's mobile web page initiates a payment request to the *PS-App* using a URL scheme.

QR Code Payment. QR code payment fundamentally differs from the previous two scenarios with respect to device interaction. In this scenario, the user browses the merchant's website using a desktop browser, while the payment is completed on a mobile device. Since the merchant's web page cannot directly redirect to a mobile payment application, the payment provider employs a QR code as an intermediary between the PC and mobile platforms. After placing an order, the merchant website generates a payment QR code, which the user scans with the *PS-App* to obtain the payment parameters.

2.3 Payment Patterns

During the *Prepayment* phase, the generation methods of payment parameters vary across different providers and scenarios. We introduce two high-level payment patterns adopted by the two leading payment providers, PSP-B and Alipay.

Payment Pattern A. In this pattern, the payment parameters are generated directly by the *MS*. Figure. 2 details the process by which the *MS* generates the *order information*. Based on the *goods* selected by the user, the *MS* calculates the *amount* to be paid and the order *description*, and generates a unique order identifier (id_{order}). These values, together with the merchant identifier (id_M), constitute the order information. The *MS* then signs the order information using its private key $priv_M$, producing a signature value S_{order}. The

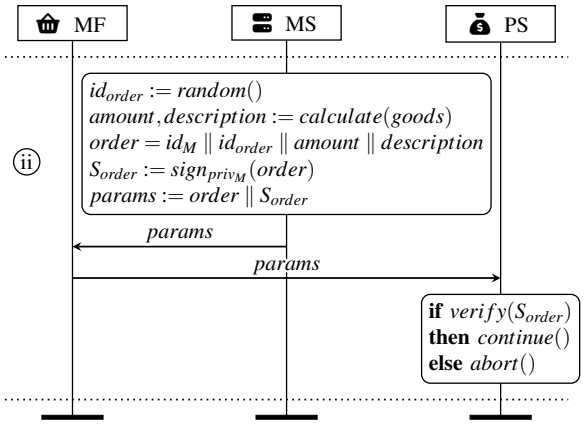


Figure 2: Payment Pattern A

order information and S_{order} are used directly as the payment parameters. The *PS* then verifies S_{order} to ensure the integrity of the order information. Payment pattern A is employed by Alipay in the in-app and mobile web payment scenarios.

Payment Pattern B. In this pattern, the payment parameters are generated by the *MS* with the involvement of the *PS*. As in pattern A, the *MS* must sign the critical order information. However, unlike pattern A, the order information and its signature, S_{order}, cannot be used directly as payment parameters. Instead, they are sent to the *PS* as a prepayment request. After verifying S_{order}, the *PS* generates a unique prepayment identifier, id_{prepay}, for the payment and returns it to the *MS*. The *MS* then concatenates id_M with id_{prepay} and signs the result using $priv_M$, producing the signature S_{prepay}. Finally, id_M, id_{prepay} and S_{prepay} are sent to the *PS* as the payment parameters. The *PS* verifies the validity of S_{prepay} to ensure the integrity of the payment request. Payment pattern B is adopted by Alipay in the QR Code Payment scenario and by PSP-B across all three payment scenarios.

2.4 Tamarin-Prover

Tamarin-Prover is a symbolic verification tool widely used for analyzing real-world security protocols, including EMV [7, 8, 29], TLS [13, 14, 19], Bluetooth [17, 34], and 5G-AKA [6, 12]. It employs a multiset rewriting system to model both protocol execution and adversary capabilities, providing a rich language for specifying security properties as first-order logic formulas over execution traces. Tamarin-Prover supports unbounded protocol sessions, a Dolev-Yao style adversary, and is particularly well-suited for analyzing stateful protocols due to its native handling of mutable and persistent state.

Terms and Facts. In Tamarin-Prover , protocol messages are represented as *terms*, which are constructed from variables, constants, and compound expressions formed by applying function symbols to subterms. An equational theory over terms specifies algebraic properties and reducibility relations,

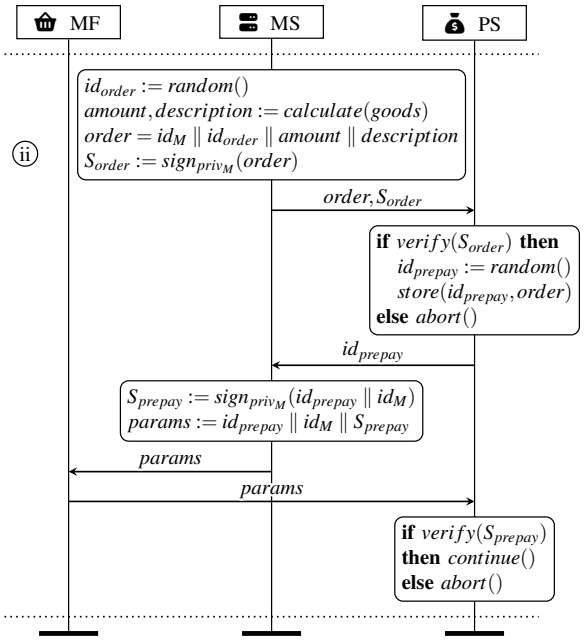


Figure 3: Payment Pattern B

thereby affecting pattern matching and substitution. For example, symmetric encryption is modeled by a encryption function `senc` and a decryption function `sdec` satisfying the equation $\texttt{sdec}(\texttt{senc}(m,k),k) = m$. The system state is captured using *facts*, which are predicates of the form $\texttt{F}(t_1,\ldots,t_n)$. Facts can be linear (consumed upon rule application), persistent (available throughout the execution, e.g., $!\texttt{PK}(A, Pk_A)$). The system execution can be observed using action facts, which record execution steps such as $\texttt{Running}(A,x)$ and $\texttt{Commit}(B,x)$ and are retained to form execution traces but do not constitute system states.

Multiset Rewriting Rules. Multiset rewriting rules formally describe protocol behavior. Each rule specifies a state transition by means of *left-hand side* premise facts (guards) and *right-hand side* conclusion facts (productions), and may also include action facts as observable events. Formally, a rule can be written as following:

$$[\, LF_1,\ldots,LF_m \,] - [\, AF(t_1,\ldots,t_k) \,] \rightarrowtail [\, RF_1,\ldots,RF_n \,]$$

Given a current state multiset S, the rule is enabled when $LF_1,\ldots,LF_m \subseteq S$. We use the function `linear` to denotes the linear facts from a set of facts. The application of the rule produces a new state $S' = (S \setminus \texttt{linear}(\{LF_1,\ldots,LF_m\})) \cup \{RF_1,\ldots,RF_n\}$, and appends the action facts $AF(t_1,\ldots,t_k)$ to the execution trace.

Trace and Formulas. In Tamarin-Prover , the global behavior of a protocol is captured by traces, which are sequences of action facts. Formally, a trace is represented as $\tau = \langle FA_1@t_1, FA_2@t_2, \ldots, FA_n@t_n \rangle$, where $FA_i@t_i$ denotes an observable event occurring at time point t_i. Based on such traces, Tamarin-Prover reasons about executions using first-

order logic formulas, where *restrictions* are used to specify global constraints over all possible executions. These constraints are also first-order logic formulas over events and time points. Within this restricted semantic space, security properties are likewise expressed as first-order temporal logic formulas (lemmas), such as secrecy, authentication, or consistency properties, and are required to hold for all valid traces. For example, key secrecy can be formalized as:

```
All k #i. Install(k)@#i ⟹ not(Ex #j. K(k)@#j)
```

This formula expresses that for any key k and time point i, if this key k is installed at time i, then there doesn't exist a time point j at which the key k is known by an adversary.

3 Threat Model and Assumptions

In this section, we introduce the threat model and security assumptions used in our formal analysis and attacks.

3.1 Threat Model

In this paper, we consider the following adversary model.

1. **Payment Parameter Acquisition.** We assume that adversary can obtain legitimate payment parameters for arbitrary goods from any merchant platform. This assumption is realistic, as most merchant platforms support public registration, allowing adversaries to create accounts and complete purchases to acquire payment parameters.

2. **Compromised Insecure Channels.** The adversary follows the standard Dolev-Yao model, where the adversary is able to fully control insecure communication channels, including eavesdropping, modifying, replaying, and injecting arbitrary messages. In third-party online payment ecosystems, *PS* cannot provide absolute security guarantees for all communication channels between external entities. Therefore, we adopt weak assumptions for such external channels and grant the adversary man-in-the-middle capabilities over these insecure channels. The specific channels considered insecure are discussed in Section 3.2.

3.2 Security Assumptions

Regarding the user, the payment system, and the communication channel, we make the following security assumptions:

User Assumptions. During the *Authorization* phase, users review the order information before authorizing the payment. We assume that users can correctly verify whether the payment *amount* and order *description* displayed by the *PS-App* match the goods selected on the *MF*. Another key element displayed on the payment interface is the payee's account name. However, since this name does not necessarily correspond to the merchant's brand name, *U* cannot reliably verify whether the displayed account belongs to the intended merchant. For instance, when purchasing a video game on Steam (a leading global digital game distribution and social platform), the payee's name shown is "Valve Corporation" (the parent company of Steam) rather than "Steam". Users without relevant background knowledge may find it difficult to determine whether "Valve Corporation" is indeed the payee account associated with "Steam". Therefore, we assume that users do not rely on the payee's account name when

Payment System Assumptions. The *PS* comprises two components: the *PS-App* and the *PS-Server*. We assume that the *PS-App* and the *PS-Server*, as well as the communication channel between them, are trusted. Unless otherwise specified, we treat them as a single *PS* entity in the subsequent analysis.

Channel Assumptions. According to developer manuals of various payment providers, the *MS* is recommended to be configured with security protocols such as HTTPS to ensure secure communication. This implies that channels where *MS* participates should be secure. However, we note that this assumption is overly strict, as a recommendation in the developer manual alone cannot provide a reliable guarantee of secure communication. Therefore, in our analysis, we relax the security assumptions for certain channels. We consider the following communication channels to be potentially insecure:

I. ***MF*** $\Longleftrightarrow$ ***MS*****:** We assume that the channel between *MF* and *MS* is insecure. This assumption is motivated by the fact that *MF* operates on the user's device, and users should not be assumed to possess a high level of security awareness or to be able to defend against diverse threats in real-world network environments. In addition, given the large number of merchants involved, payment service providers cannot guarantee that all front-end and back-end communications of merchants are free from security vulnerabilities.

II. ***MF*** $\Longleftrightarrow$ ***PS*****:** The communication between *MF* and *PS* occurs during the *Prepayment* phase, in which *MF* invokes the *PS-App* to send payment parameters. This interaction is vulnerable to hijacking attacks in practice, as evidenced by real-world exploits discussed in Section 6. Thus, we assume that this channel is insecure.

4 Payment Protocol Modeling

We formally model the payment protocols of PSP-B and Alipay in three payment scenarios, and the full models are available in an open-source repository [4]. In this section, we present the key details for modeling third-party payment protocols. First, we describe the user model, which allows us to capture users' evaluation of payment information and their authorization behavior within the symbolic model. Then, we introduce the oracle model for the adversary to obtain legitimate payment parameters.

4.1 User Model

Under our security assumptions, users are capable of accurately identifying the payment *amount* and *description*, thereby avoiding being misled into authorizing a payment. To model this capability in the symbolic model, we introduce two unary functions, `calc_amt` and `gen_desc`, which are used to compute the payment *amount* and *description* from *goods*, respectively. During the *Ordering* phase, users select goods. For the same goods (note that this does not refer to the same product model), a user can only purchase them at most once. Therefore, in our model, we use a fresh value as an abstract representation of the goods. The following rule models the user's goods ordering behavior:

$$[\,\texttt{Fr}(\sim goods)\,] \,-[\,]\!\rightarrow\, [\,\texttt{Order}(\sim goods), \texttt{SOut}(\ldots, \sim goods)\,]$$

This rule represents that the user U selects a product $\sim goods$ and sends it to *MF*. Here, $\texttt{Fr}(\sim goods)$ indicates that the value $\sim goods$ is freshly generated. $\texttt{Order}(\sim goods)$ indicates that the user has placed an order for the goods $\sim goods$. In *Prepayment* phase, *MS* computes the payment *amount* and *description* based on $\sim goods$ using `calc_amt` and `gen_desc`, as follows:

$$\begin{aligned} amount &= \texttt{calc\_amt}(\sim goods) \\ description &= \texttt{gen\_desc}(\sim goods) \end{aligned}$$

In *Authorization* phase, the user obtains the order information from the *PS-App* and confirms it. Due to pattern matching, the user authorizes a payment if only if the payment *amount* and *description* exactly match $\texttt{calc\_amt}(\sim goods)$ and $\texttt{gen\_desc}(\sim goods)$, respectively, and will not authorize any other values. The following rule models the user's payment authorization behavior:

$$\begin{aligned} &[\,\texttt{Order}(\sim goods), \\ &\texttt{SIn}(\ldots, \langle \texttt{calc\_amt}(\sim goods), \texttt{gen\_desc}(\sim goods)\rangle)\,] \\ &-[\,]\!\rightarrow\, [\,\texttt{UserAuthorizePayment}(\ldots)\,] \end{aligned}$$

Here, `UserAuthorizePayment` indicates that the user authorizes the payment. Notably, the above model implies that users can accurately distinguish between different items, even when they belong to the same product model. In practice, users do not possess such capabilities, nor do merchant platforms provide functionality that allows users to select items based on product serial numbers. Nevertheless, this idealized user model helps us focus on analyzing the security of the payment protocol itself, without conflating the analysis with errors that users may make.

4.2 Adversarial Capabilities Model

In our threat model, the adversary is capable of obtaining legitimate payment parameters. In practice, the adversary acquires such legitimate payment parameters by accessing merchant platforms in a normal manner and placing orders to purchase goods. To intuitively model this behavior, we introduce an *Order Oracle* within the symbolic framework, which is defined by the following rule:

$$\begin{aligned} &\texttt{Rule OrderOracle}: \\ &[\,\texttt{In}(\langle amt,\ desc\rangle), \texttt{!Merchant}(id_M,\ priv_M)\,] \,-[\,]\!\rightarrow \\ &[\,\texttt{Out}(\texttt{gen\_params}(amt,\ desc,\ id_M,\ priv_M))\,] \end{aligned}$$

By querying this oracle with an order amount *amt* and an order description *desc*, the adversary can obtain the corresponding legitimate payment parameters. The oracle is granted access to the merchant's identifier id_M and rivate key $priv_M$, enabling it to generate legitimate payment parameters that are accepted by *PS*. The function $\texttt{gen\_params}(amt,\ desc,\ id_M,\ priv_M)$ represents the process by which the oracle generates payment parameters from *amt* and *desc*. This process is consistent with the procedure used by merchants to generate payment parameters in both payment pattern A and payment pattern B. We use equations to flexibly define this process in symbolic models. In protocols conforming to Model A, the equation is defined as follows:

$$\begin{aligned} &\texttt{gen\_parmas}(amt, desc, id_M, priv_M) = \\ &\qquad amt \parallel desc \parallel id_M \parallel \texttt{sign}(amt \parallel desc \parallel id_M, priv_M) \end{aligned}$$

For protocols under Model B, the equation is as follows:

$$\begin{aligned} &\texttt{gen\_parmas}(amt, desc, id_M, priv_M, id_{prepay}) = \\ &\qquad id_{prepay} \parallel id_M \parallel \texttt{sign}(id_{prepay} \parallel id_M, priv_M) \end{aligned}$$

Here, our equation simplifies the actual generation process of payment parameters, such as the generation of id_{prepay}. In the modeling, we incorporate additional necessary parameters and computational steps according to the details of the actual protocol, ensuring the model's accuracy and completeness.

5 Security Properties

In this section, we formally define the security properties of third-party online payment protocols. The core security goal of protocols is to ensure that the user, the merchant, and the payment service provider reach a consistent view of a transaction. The protocol must guarantee that (a) all parties agree on the payment order being processed, (b) each payment is authorized by the user, and (c) the reported payment result faithfully reflects the actual fund transfer.

All properties considered in this section are instances of *injective agreement* [22]. Injective agreement strengthens standard agreement by requiring a one-to-one correspondence between protocol runs: each successful local completion must correspond to a unique execution of the peer, ruling out replay, duplication, or reuse of protocol messages. Two types of atomic facts are necessary to define injective agreement:

1. $\mathsf{Running}(A, B, x)$: records that party A is interacting with B in a concrete execution of the protocol and that the interaction involves data x (an objective execution fact);

2. Commit(A,B,x): denotes that party A subjectively believes that it has reached agreement with B on data x (e.g., after receiving and accepting a message).

This notion is particularly well-suited for online payment scenarios, where both consistency and uniqueness of transactions are fundamentally essential requirements. We structure the security properties according to the logical progression of a payment protocol, namely agreement on the *order*, agreement on *user authorization*, and agreement on the *result*.

5.1 Injective Agreement on Order

Injective agreement on the order ensures that all involved parties process the same payment order and that each order is handled at most once. Violations of this property indicate that an adversary may have tampered with the order information or caused inconsistent views among the participants. Since this property involves three principals, we separately consider the agreements between *PS* and *MS*, and between *U* and *MS*.

IA-O1 (*PS* ← *MS*): This lemma captures injective agreement on the order from the perspective of *PS*. If *PS* accepts a payment request for an order allegedly issued by *MS*, then *MS* must have previously initiated a corresponding request for that order. Moreover, no other payment request for the same order can be accepted by *PS*, ensuring uniqueness. The property is formalized as follows:

```
All ps ms order #i. PSCommit(ps, ms, order)@#i
  ⟹ Ex #j. MSRunning(ms, ps, order)@#j ∧
    ¬( Ex ps2 ms2 #i2. #i2 ≠ #i ∧
          PSCommit(ps2, ms2, order)@#i2)
```

Here, the *Commit* and *Running* facts follow the standard interpretation introduced in [22]. The fact `PSCommit(ps, ms, order)` denotes that *PS* has accepted a payment request for `order`, while `MSRunning(ms, ps, order)` records that *MS* has actually issued such a request. The term `order` denotes order information, including the merchant's order identifier, the payment amount, and the product description.

IA-O2 (*U* ← *MS*): From the user's perspective, injective agreement on the order ensures that the user and the merchant reach agreement on order information during the payment process, and that each order is processed exactly once. The formalization is as follows:

```
All u ms goods #i. UMCommit(u, ms, goods)@#i
  ⟹ Ex #j. MSRunning(ms, u, goods)@#j ∧
    ¬( Ex u2 ms2 #i2. #i2 ≠ #i ∧
          UMCommit(u2, ms2, goods)@#i2)
```

Unlike the order information considered in IA-O1, IA-O2 only requires agreement between the user and the merchant on the purchased goods, since the user is primarily concerned with what is being bought.

5.2 Injective Agreement on User Authorization

Injective agreement on user authorization ensures that each successful payment is backed by a unique and explicit authorization from the user. This property prevents both unauthorized payments and reuse of a single authorization to complete multiple transactions.

IA-UA (*PS* ← *U*): Specifically, if *PS* completes a payment based on a password hash, then the corresponding password must have been input by *U*, and each password input can authorize at most one payment:

```
All u p s #i. PaymentComplete(u, h(p), s)@#i
  ⟹ Ex #j. UserInputPwd(u, p, s)@#j ∧
    ¬( Ex u2 p2 #i2. #i2 ≠ #i ∧
          PaymentComplete(u2, h(p2), s)@#i2)
```

In this lemma, the fact `UserInputPwd(u, p, s)` represents that the user `u` inputs the password `p` in a session `s` to authorize a payment. The fact `PaymentComplete(u, h(p), s)` denotes that *PS* has received the hash value `h(p)` of the password `p` sent by `u`, and has completed the payment.

5.3 Injective Agreement on Result

Finally, injective agreement on the payment result guarantees that successful payment notifications cannot be forged, replayed, or duplicated. A violation of this property would allow an adversary to falsely convince the user or the merchant that a payment has been completed, even though no actual fund transfer has occurred. We again consider the perspectives of *MS* and *U* separately.

IA-R1 (*MS* ← *PS*): From the merchant's view, if *MS* receives a successful payment notification for an order, then this notification must have been generated by *PS*. Also, *MS* must not receive multiple successful notifications for the same order. This is formalized in the lemma:

```
All ms ps order #i. MSCommit(ms, ps, order)@#i
  ⟹ Ex #j. PSRunning(ps, ms, order)@#j ∧
    ¬( Ex ms2 ps2 #i2. #i2 ≠ #i ∧
          MSCommit(ms2, ps2, order)@#i2)
```

IA-R2 (*U* ← *PS*): For the user, if *U* observes a successful payment notification in the application, then *PS* must have sent a corresponding notification. Each order produces at most one notification. Formally,

```
All u ps goods #i. UPCommit(u, ps, goods)@#i
  ⟹ Ex #j. PSRunning(ps, u, goods)@#j ∧
    ¬( Ex u2 ps2 #i2. #i2 ≠ #i ∧
          UPCommit(u2, ps2, goods)@#i2)
```

Table 1: **Verification Results of Third-Party Online Payment Protocols.** Each protocol is evaluated under three channel security assumptions. IA-O1/O2, IA-UA, IA-R1/R2 denote the security properties defined in Section 5. ✓ indicates that the security property is satisfied, while ✗ indicates a violation. Time reports the total analysis time.

Assumption	Provider	Scenario	Model	IA-O1	IA-O2	IA-UA	IA-R1	IA-R2	Time (s)
Base	Alipay	In-App	A	✗	✗	✓	✓	✓	265.8
Base	Alipay	Mobile-Web	A	✗	✗	✓	✓	✓	266.22
Base	Alipay	QR-Code	B	✓	✗	✓	✓	✓	606.59
Base	PSP-B	In-App	B	✓	✗	✓	✓	✓	562.21
Base	PSP-B	Mobile-Web	B	✓	✗	✓	✓	✓	88.18
Base	PSP-B	QR-Code	B	✓	✗	✓	✓	✓	123.89
SecureMS	Alipay	In-App	A	✗	✗	✓	✓	✓	13.94
SecureMS	Alipay	Mobile-Web	A	✗	✗	✓	✓	✓	14.39
SecureMS	Alipay	QR-Code	B	✓	✗	✓	✓	✓	73.57
SecureMS	PSP-B	In-App	B	✓	✗	✓	✓	✓	19.96
SecureMS	PSP-B	Mobile-Web	B	✓	✗	✓	✓	✓	7.73
SecureMS	PSP-B	QR-Code	B	✓	✗	✓	✓	✓	7.22
SecurePS	Alipay	In-App	A	✗	✗	✓	✓	✓	19.55
SecurePS	Alipay	Mobile-Web	A	✗	✗	✓	✓	✓	18.2
SecurePS	Alipay	QR-Code	B	✓	✗	✓	✓	✓	155.14
SecurePS	PSP-B	In-App	B	✓	✗	✓	✓	✓	159.02
SecurePS	PSP-B	Mobile-Web	B	✓	✗	✓	✓	✓	20.66
SecurePS	PSP-B	QR-Code	B	✓	✗	✓	✓	✓	26.77

5.4 Analysis Results

In Section 3.2, we provide a detailed rationale for adopting weak security assumptions for the two communication channels $MF \iff MS$ and $MF \iff PS$. Based on different combinations of these two weak-channel assumptions, we construct three channel security assumptions for our analysis:

1. **Base:** both $MF \iff MS$ and $MF \iff PS$ are insecure;
2. **SecureMS:** only $MF \iff PS$ is insecure;
3. **SecurePS:** only $MF \iff MS$ is insecure.

Under these three channel security assumptions, we conduct a comprehensive automated security analysis of six third-party online payment protocols, covering three payment scenarios for each of Alipay and PSP-B. All analyses are performed using Tamarin-Prover version 1.10.0 and are executed on a server equipped with a 24-core, 32-thread Intel Core i9-14900K processor and 192,GB of memory, running Ubuntu 24.04 LTS. This section presents and discusses the verification results of the above analyses in detail.

5.4.1 Results

Table 1 summarizes our analysis results. We find that two protocols following payment pattern A fail to satisfy the IA-O1 property under all three channel assumptions. With respect to IA-O2, none of the six analyzed protocols we analysed satisfies this property under our channel assumptions.

We further observe that the analysis results remain consistent across all three channel assumptions. This indicates that as long as any insecure channel exists, an adversary can successfully carry out an attack, causing the user and the merchant to hold inconsistent views of the order information and thereby violating the IA-O1 and IA-O2 properties.

Moreover, across all payment scenarios, protocols adopting payment pattern A fail to satisfy both IA-O1 and IA-O2, while protocols following payment pattern B fail to satisfy IA-O2. This demonstrates that protocol security is closely tied to the adopted payment pattern, whereas variations in payment scenarios have a limited impact on security. All counterexamples violating IA-O1 and IA-O2 shows that each of them arises from an order tampering attack.

5.4.2 Order Tampering Attack

The order tampering attack was first identified by [41] in the in-app payment scenario. Our formal analysis reveals that *this attack is pervasive across all payment scenarios and all payment patterns*. As long as any insecure channel exists, an adversary can successfully launch this attack. Through an analysis of counterexamples violating IA-O1 and IA-O2, we identify two variants of the order tampering attack under different channel assumptions. Figure 4 illustrates these

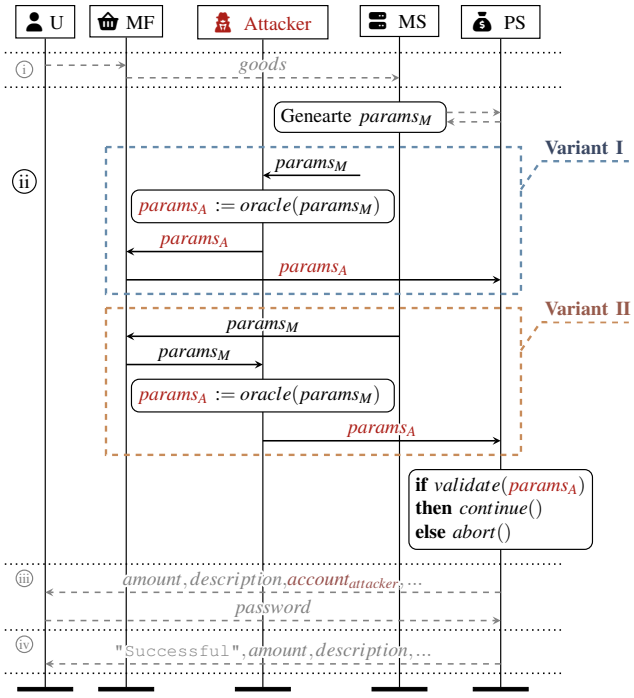


Figure 4: Order Tampering Attack Variants. Notation: $params_M$ denotes the original payment parameters generated by the merchant; $params_A$ denotes the payment parameters after being tampered with by the adversary; $oracle(params_M)$ denotes the process by which the adversary generates payment parameters such that the order information remains fully consistent with $params_M$.

two variants. In Variant I, the adversary targets the channel $MF \Longleftrightarrow MS$, causing U to ultimately authorize an attacker-specified order. In Variant II, the adversary tampers with the order on the channel $MF \Longleftrightarrow PS$.

In prior work [21, 40, 41], discussions of order tampering attacks primarily focus on Variant I. Under the channel assumptions of SecurePS, all counterexamples that violate the security properties originate from order tampering attacks on Variant I. However, our analysis under the channel assumptions of SecureMS indicates that Variant II also has the potential to be successfully exploited in real-world settings. In Section 6, we present a widely prevalent vulnerability that we discovered, which can be exploited to launch a Variant II order tampering attack.

6 Real-World Attacks

In this section, we present a real-world attack case that was identified inspired by our analysis results and successfully carried out in practice. We also introduce our measurement study to assess the scope and practical extent of the vulnerability's impact across affected systems. We further discuss the responsible vulnerability disclosure process.

6.1 A Real-World Vulnerability

Inspired by the Variant II attack, we identified an exploitable vulnerability on the Android platform. When initiating a payment request, merchant's Android applications use the Android Intent mechanism to transmit payment parameters to the payment service application. As a core inter-component communication mechanism provided by the Android system, Intent serves as the primary means by which the PSP-B and Alipay receive payment parameters from merchant applications. A merchant application typically constructs an Intent containing payment parameters and uses it to launch the corresponding payment application.

It should be noted that Intents can be classified into two categories, explicit Intents and implicit Intents. An explicit Intent specifies the fully qualified package name of the target application, whereas an implicit Intent only declares the action, category, and data, allowing the system to match and select a responding component based on Intent Filters. The security risk primarily arises from the misuse of implicit Intents.

When requesting a payment, a merchant application should use an explicit Intent to explicitly designate the PSP-B or Alipay applications as the target, thereby ensuring that the payment request is correctly routed to a trusted payment application. However, we find that some merchant applications use implicit Intents in practice. This means that an attacker can hijack payment requests originally sent to a payment application by deploying a malicious application that declares the same Intent handling capability.

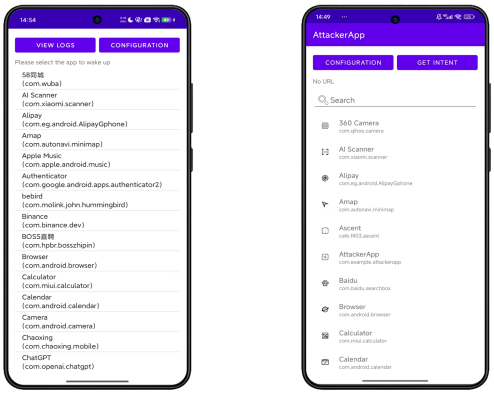
Intent hijacking is not specific to payment scenarios, but rather a systemic security risk inherent in the Android component communication mechanism. However, existing third-party online payment protocols are designed under the assumption that the communication channel between the merchant client and the payment application is trusted. This channel assumption is unrealistic in real-world deployments.

6.2 Attack Implementation

We developed two Android applications, a malicious application and an attacker application, to exploit the aforementioned vulnerability. Fig. 5 illustrates the devices used in the attack, serving as the victim device and the attacker device, respectively, with the corresponding applications installed. The malicious application on device 5a is responsible for capturing and replaying payment parameters. The attacker application on device 5b places orders on the merchant platform and returns payment parameters to the malicious application.

Fig. 6 illustrates the workflow of the new order tampering attack. The attack proceeds in four phases:

1. A benign merchant application initiates a payment request by dispatching an implicit Intent containing payment parameters to the payment application;



(a) Victim Device (b) Attacker Device

Figure 5: Attack Setup. Device 5a denotes the victim device, which installs the malicious application, a vulnerable merchant application, and a payment application. Device 5b denotes the attacker device, which installs the attacker application. The screenshots correspond to the real interfaces of the malicious and attacker applications, respectively.

2. By hijacking the Intent resolution process, the malicious application intercepts the original payment parameters and extracts the embedded order information;

3. Using the extracted information, the attacker application places a new order under an attacker-controlled account on the merchant platform and obtains a fresh set of payment parameters that preserve the original order information while being bound to the attacker's account;

4. The tampered payment parameters are then relayed to the payment application, which processes the request without detecting the modification.

As a result, the attacker successfully tampers with the order, causing the user to complete a payment specified by the attacker rather than the original merchant.

6.3 Impact Assessment

To assess the real-world impact of the identified vulnerability, we conducted a manual measurement study on Android applications and games that integrate the PSP-B or Alipay SDK. Our dataset consists of 150 popular Android applications and 50 games, covering up to the three most recent versions of each application and the latest version of each game. For some applications, only one or two versions were available. In total, we collected 461 application versions.

Among these versions, 419 support PSP-B and 401 support Alipay. Our analysis reveals that 64 PSP-B–enabled versions and 72 Alipay-enabled versions are vulnerable. Out of 53 applications, a total of 116 versions representing more than 25% of the analyzed dataset support at least one vulnerable

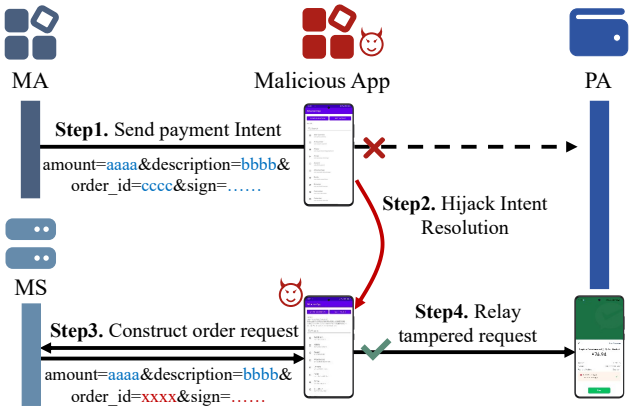


Figure 6: Attack Flow. MA denotes the merchant application; Malicious Application denotes the malicious application we developed; PA denotes the payment application; MS denotes the merchant server.

payment method. These vulnerable applications span a wide range of categories, including e-commerce, online education, and entertainment. Their cumulative download count reaches 265.85 billion, indicating that the vulnerability potentially affects a very large user base and poses a significant security risk in practice.

6.4 Responsible Disclosure

Following responsible disclosure guidelines, we reported the vulnerability to the security teams of PSP-B and Alipay in December 2025. At the time of writing, both providers had acknowledged the issue and confirmed that remediation efforts were underway. We identified contact information for 41 affected Android applications and issued security notifications to their developers, detailing the underlying risks and corresponding mitigation strategies. Among these, 17 developers responded; one explicitly committed to fixing the vulnerability, while the remaining vendors attributed the issue to the Alipay and PSP-B SDKs. We did not notify the Google Play Store, as Chinese apps are distributed mainly through vendor stores and the root cause lies in the Alipay and PSP-B SDKs, which a single update can fix for all integrating apps. Other major systems (Apple Pay, Stripe, etc.) use architectures not exposed to this attack.

7 Countermeasure

In this section, we analyze the causes of order tampering attacks and introduce our countermeasure. Then, we verify our countermeasure using symbolic methods and present the results. Finally, we discuss our countermeasure.

7.1 Cause Analysis

Superficial Causes. From a superficial perspective, order tampering attacks arise from the existence of insecure communication channels between *MS* and *PS-App*. An attacker can intercept and tamper with order information along this channel, thereby misleading the user into authorizing an incorrect payment. At first glance, addressing this issue appears straightforward. It suffices to ensure that the communication channels between *MS* and *PS-App* are secure. However, for payment service providers, this seemingly simple solution is difficult to realize in practice. The security of the communication path from merchant backends to *PS-App* is influenced by both merchants and end users, and is largely beyond the control of payment service providers. Payment service providers serve a large number of merchants, each with diverse backend implementations, making it infeasible to ensure that all merchants correctly secure their communication channels. Similarly, merchants serve a vast user base with heterogeneous device environments, and payment service providers cannot guarantee that all users operate within secure network conditions. Therefore, defending against order tampering attacks solely by securing communication channels is, in practice, highly challenging. Consequently, mitigation is often reactive: once an insecure communication channel is identified, merchants and users are notified to apply patches.

Root Causes. The root cause of such attacks lies in the lack of a user-side authentication mechanism for order information in existing payment patterns. We examine the typical actions performed by users during the payment process: entering a payment password to authorize the transaction, and possibly verifying the payment amount and description. However, the user does not actually authenticate the order. Information such as the payment amount and item description merely represents the content of the current payment, and does not correspond to the original order that the user intends to confirm. In other words, during the *Authentication* phase, the user performs no form of authentication of the order data. The user's authorization is applied only to the act of payment transaction, rather than to a specific order. Once an attacker is able to tamper with order parameters in transit, the user can be induced to authorize an incorrect order. This observation reveals a design flaw in current payment patterns.

7.2 Countermeasure Design

To fundamentally address order tampering attacks, we design and propose a new third-party payment protocol that introduces a user-side order authentication mechanism to effectively prevent such attacks. Our payment protocol distinguishes between *initial* and *regular* payments, whose workflows are illustrated in Fig. 7 and Fig. 8, respectively. For a given merchant and user, the first payment initiated by the user is defined as an *initial* payment, while all later payments

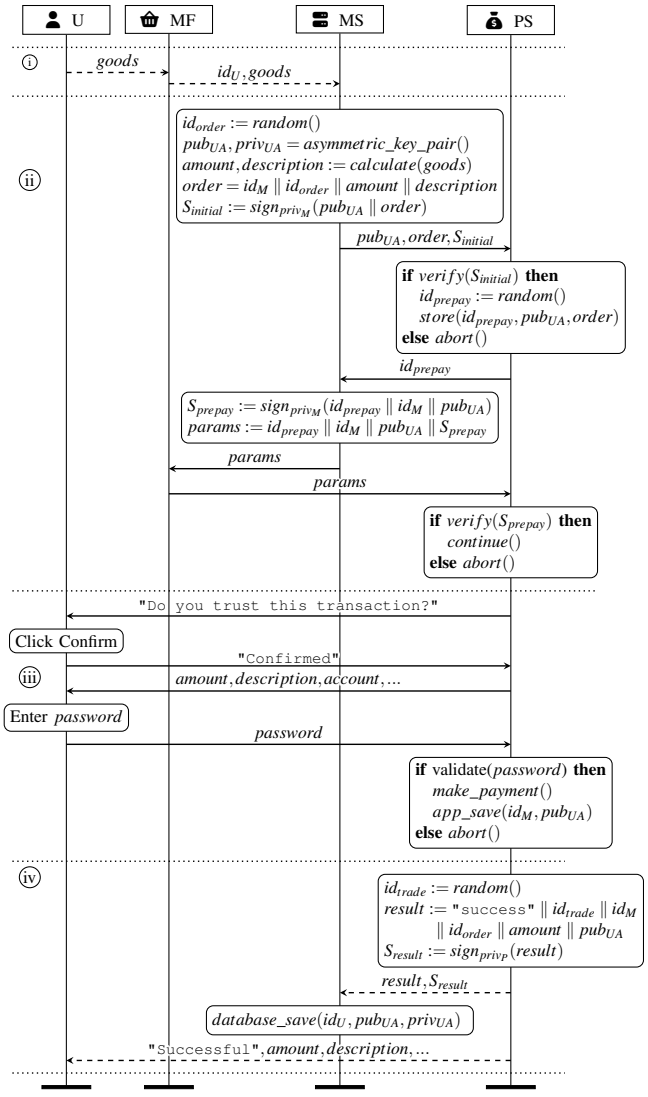


Figure 7: Initial Payment

are defined as *regular* payments.

At the beginning of the *Prepayment* phase of a user's initial payment, *MS* generates a public–private key pair $(priv_{UA}, pub_{UA})$ for *U*, and then uses the merchant private key $priv_M$ to sign pub_{UA} together with the order information before sending them to *PS*. Then, *PS* temporarily stores pub_{UA} together with the order information and returns the prepay identifier (id_{prepay}) to the merchant. Upon receiving id_{prepay}, *MS* generates two signatures over the payment parameters using pub_M and pub_{UA}, respectively, and sends signatures and id_{prepay} to *MF* During the *Authentication* phase, *PS-Server* send the order information together with pub_{UA} to *PS-App*. At this point, *PS-App* notifies the user that this is the first payment to the merchant and asks whether this merchant should be trusted. If the user chooses to trust the merchant, *PS-App* verifies both signatures and, upon successful verifi-

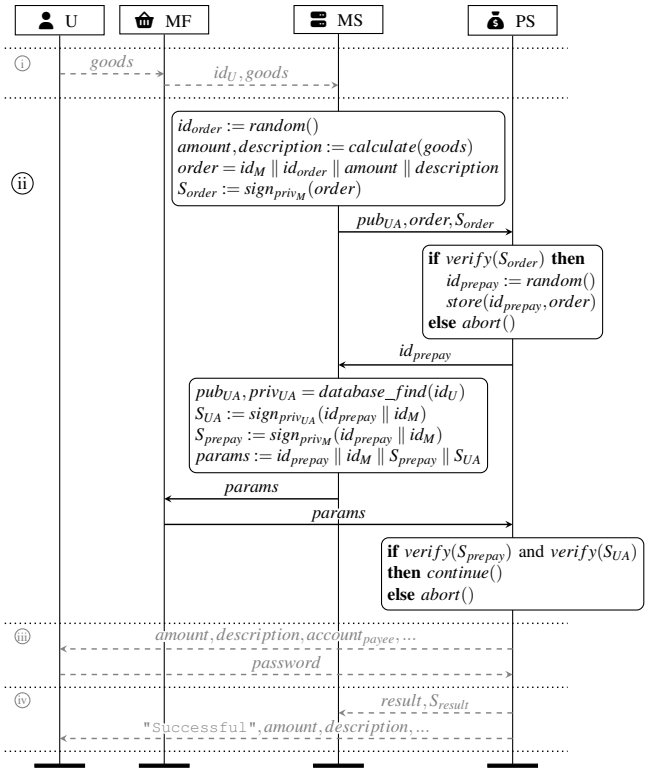


Figure 8: Regular Payment

cation, binds pub_{UA} to the merchant identifier and stores it locally. Otherwise, the payment process is aborted.

In regular payments, the *Prepayment* phase largely follows payment pattern B. The only difference is that *MS* additionally signs id_{prepay} and id_M using $priv_{UA}$. Then, *PS-App* first checks whether a public key pub_{UA} has already been bound locally to the merchant identifier. If such a binding exists, *PS-App* verifies both signatures before prompting the user to confirm the order information and complete the payment. Otherwise, the application informs the user that the merchant is not trusted and asks whether to proceed with the payment.

7.3 Symbolic Verification

We formally modeled and verified the initial and regular payments, and the results are summarized in Table 2. The analysis shows that under all three channel assumptions, our regular payment effectively defends against order tampering attacks and satisfies all security properties. This demonstrates that by introducing an order authentication mechanism on the user side, we can fundamentally address the problem of order tampering attacks. Although our initial payment does not satisfy all security properties under certain channel assumptions, which is consistent with the behavior of Model B, the initial payment occurs only once and thus minimizes the likelihood of successful attacks.

Table 2: Verification Results of Countermeasure. ✓ means satisfy, while ✗ indicates a violation.

Stage	IA-O1	IA-O2	IA-UA	IA-R1	IA-R2	Time(s)
Initial	✓	✗	✓	✓	✓	477.86
Regular	✓	✓	✓	✓	✓	916.97

7.4 Discussion

Advantages. From a protocol design perspective, our payment pattern exhibits three key advantages.

1. **Minimal trust assumptions.** At the protocol design level, we minimize reliance on trusted channels. The protocol is formally proven to guarantee payment security under weakened external channel assumptions.

2. **Formally provable security.** Our formal analysis results demonstrate that our payment pattern effectively prevents order tampering attacks during regular payments and satisfies all security properties defined in this paper.

Limitations. We honestly present the limitations and trade-offs of our payment pattern.

1. **Signature and storage overhead.** Our payment pattern introduces one additional signature operation, and requires both merchants and users to locally store extra keys, which incurs additional computational and implementation overhead. However, no new cryptographic primitives are introduced, and we consider the resulting overhead to be acceptable in practice.

2. **Additional user interaction during the initial payment.** During the initial payment, users are required to explicitly confirm their trust in the merchant, introducing additional user interaction. For regular payments, no extra interaction is required, and the overall user experience remains comparable to that of existing payment patterns.

3. **No security improvement for the initial payment.** Our proposed model does not provide additional security guarantees for the initial payment. Instead, our design follows the widely adopted Trust-On-First-Use (TOFU) model, under which the initial interaction may be vulnerable to active network attacks. Once a legitimate payment context is established, however, any subsequent deviation can be reliably detected. Consequently, while an adversary may interfere with the initial payment, our scheme guarantees the integrity of subsequent payments and enables attack detection even if the first payment was compromised.

8 Related Work

8.1 Third-Party Payment System

In recent years, the research community has devoted substantial effort to the security of third-party payment systems. Overall, prior work has largely focused on specific payment scenarios or concrete system implementations, with empirical analyses of third-party payment workflows used to uncover security flaws in real-world systems.

In in-app payment scenarios, early studies uncovered multiple security vulnerabilities in different payment platforms. Early work by [25, 32] identified security flaws in Google's in-app billing service, which could be exploited to bypass payment verification and enable unauthorized free purchases. Focusing on the Indian market, [18] conducted an in-depth analysis of the Unified Payments Interface (UPI) and uncovered design flaws in its multi-factor authentication mechanism. [40,41] performed an empirical study of in-app payment services in China. They proposed a set of security rules, and identified widespread rule violations in real-world systems, revealing pervasive risks of payment bypass and payment fraud. More recently, [24] conducted a large-scale measurement on payment libraries in over 10,000 Android apps. They found 71.7% of apps rely on outdated or insecure payment SDKs, exposing hundreds of leaked private keys and tens of thousands of high-risk vulnerabilities.

In web-based payment scenarios, prior research has centered on the interaction workflows between merchant websites and third-party payment platforms. [39] studied multiple online shopping platforms that integrate payment services such as PayPal and revealed severe logic vulnerabilities in these systems. Later work, including [28, 35, 36], used symbolic execution and black-box analysis to identify payment logic flaws across many web applications.

Beyond analyses of concrete implementations, several studies have investigated third-party payment systems from alternative perspectives. [11] applied NLP techniques to developer documentation of payment services and found multiple exploitable logic vulnerabilities. From a design and defense perspective, [26] proposed a privacy-preserving mobile payment protocol and formally verified its security using ProVerif. [21] investigated personal payment systems built atop third-party payment infrastructures and identified multiple vulnerabilities at both the protocol and implementation levels.

8.2 Banking Payment Systems

Early research primarily focused on uncovering security vulnerabilities in mobile banking systems through empirical and systematic analyses. [27] revealed security flaws in mobile banking systems from an application design perspective, while [5, 16] further highlighted pervasive privacy and security issues in real-world deployments. Building on this line of work, [30, 31] conducted a systematic analysis of transaction workflows in branchless banking applications, revealing widespread transaction integrity vulnerabilities.

As payment protocols have grown in complexity, researchers have increasingly employed formal methods to model and verify bank-based payment systems, particularly for credit card and EMV protocols. De Ruiter et al. [33] conducted a formal analysis of a variant of the EMV protocol using ProVerif, marking the first systematic application of formal methods to EMV protocol analysis. Subsequently, more precise and comprehensive models were proposed. Basin et al. [8] modeled the complete EMV protocol using the Tamarin-Prover . Formal analysis identified and explained multiple real-world attacks, including the PIN bypass vulnerability [9] and the card brand mixup attack [7].

8.3 Vulnerabilities in Implementations

In mobile ecosystems, payment applications suffer from a wide range of implementation-level vulnerabilities arising from insecure inter-application communication and input handling. Prior work shows that improperly exported Android components enable launch hijacking and intent spoofing, allowing malicious applications to intercept or impersonate legitimate payment components [10, 37]. Similarly, implicit URI schemes and deep links are often insufficiently validated, enabling attackers to hijack payment flows and steal sensitive parameters [20]. Moreover, embedded QR code scanning functionalities further expand the attack surface, as malicious QR codes can trigger sensitive interfaces or tamper with [15].

9 Conclusion

This work presents a systematic security analysis of third-party payment protocols and uncovers a fundamental design anti-pattern: user authorization is never bound to order semantics, leaving major payment ecosystems open to order tampering. By formally modeling six widely deployed protocols from major Chinese providers and verifying them with Tamarin-Prover under weakened channel assumptions, we discover a novel variant of this attack and establish a security property framework spanning three dimensions.

We further validate the threat in practice: a proof-of-concept attack on Android tampers with real payment orders for both Alipay and PSP-B, and a measurement study of hundreds of popular applications finds over 25% vulnerable. Following our responsible disclosure, both providers have officially acknowledged the vulnerability.

Finally, we propose a deployable countermeasure that binds user authorization to the order through user-side order authentication. Formal verification confirms that it satisfies all security properties even under weakened channel assumptions, offering a practical path toward more secure third-party payments.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their constructive feedback. This research was supported in part by the National Cryptologic Science Fund of China under grant No. 2025NCSF02027, the National Natural Science Foundation of China under grants No. 62302343 and No. 62472323, the Key R&D Program of Hubei Province under grant No. 2024BAB018, and the Wuhan Scientific and Technical Achievements Project under grant No. 2024030803010172. The corresponding authors are Jing Chen and Min Shi.

Ethical Considerations

The parties affected by this research are the users of Alipay and PSP-B, the two providers themselves, and the merchants whose applications integrate their SDKs. Across these parties, the principal risk is the potential misuse of the disclosed technique, which we mitigate through the safeguards below, whereas the offsetting benefit is that it enables the providers and developers to remediate a fundamental design flaw before it is exploited in the wild. All experiments were performed using accounts fully controlled by the authors, without involving real users or accessing real user funds, and all transactions were limited to test environments or self-owned accounts. The Android measurement study focused solely on whether payment requests could be intercepted or hijacked at the communication level; no order tampering or manipulation of payment amounts or outcomes was performed, and no financial or operational impact was imposed on application developers or merchants. To reduce the risk of misuse, affected applications are not individually identified and are reported only in aggregated form. The protocol models presented in this paper are reconstructed solely from public developer documentation, rather than from reverse engineering or leaked internal specifications. The identified issues were responsibly disclosed to Alipay and PSP-B, as well as to affected application developers, prior to publication, enabling mitigation before public disclosure; to prevent opportunistic exploitation during the remediation window, we do not release the attack implementation or the list of affected applications.

Open Science

To support transparency and reproducibility, we release an artifact repository [4] containing the formal models of six third-party payment protocols and our countermeasure, the specifications of the proposed security properties, and scripts for reproducing the formal verification results. For security and ethical reasons, we do not release exploit code for real merchant applications or data that could directly identify vulnerable apps or merchants.

References

[1] Alipay. https://www.alipay.com/, 2025.

[2] Alipay developer documentation. https://opendocs.alipay.com/open/00a0ut?pathHash=b19b288a, 2025.

[3] China's third-party payment industry research report (2025). https://report.iresearch.cn/report/202601/4780.shtml, 2026.

[4] Formal models of third-party online payment protocols. https://doi.org/10.5281/zenodo.20303820, 2026.

[5] Gilberto Marins de Almeida. M-payments in Brazil: Notes on how a country's background may determine timing and design of a regulatory model. *Wash. JL Tech. & Arts*, 8:347, 2012.

[6] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proc. of CCS*, 2018.

[7] David Basin, Ralf Sasse, and Jorge Toro-Pozo. Card brand mixup attack: Bypassing the PIN in Visa cards by using them for Visa transactions. In *Proc. of USENIX Security*, 2021.

[8] David Basin, Ralf Sasse, and Jorge Toro-Pozo. The EMV standard: Break, fix, verify. In *Proc. of IEEE S&P*, 2021.

[9] David Basin, Patrick Schaller, and Jorge Toro-Pozo. Inducing authentication failures to bypass credit card PINs. In *Proc. of USENIX Security*, 2023.

[10] Shweta Bhandari, Wafa Ben Jaballah, Vineeta Jain, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur, Mohamed Mosbah, and Mauro Conti. Android inter-app communication threats and detection techniques. *Computers & Security*, 70:392–421, 2017.

[11] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. Devils in the guidance: Predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *Proc. of USENIX Security*, 2019.

[12] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. In *Proc. of NDSS*, 2019.

[13] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla Van Der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proc. of CCS*, 2017.

[14] Cas Cremers, Marko Horvat, Sam Scott, and Thyla Van Der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *Proc. of IEEE S&P*, 2016.

[15] Xing Han, Yuheng Zhang, Xue Zhang, Zeyuan Chen, Mingzhe Wang, Yiwei Zhang, Siqi Ma, Yu Yu, Elisa Bertino, and Juanru Li. Medusa attack: Exploring security hazards of In-App QR code scanning. In *Proc. of USENIX Security*, 2023.

[16] Andrew Harris, Seymour Goodman, and Patrick Traynor. Privacy and security concerns associated with mobile money applications in africa. *Wash. JL Tech. & Arts*, 8:245, 2012.

[17] Mohit Kumar Jangid, Yue Zhang, and Zhiqiang Lin. Extrapolating formal analysis to uncover attacks in Bluetooth passkey entry pairing. In *Proc. of NDSS*, 2023.

[18] Renuka Kumar, Sreesh Kishore, Hao Lu, and Atul Prakash. Security analysis of Unified Payments Interface and payment apps in india. In *Proc. of USENIX Security*, 2020.

[19] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to make TLS middlebox-aware? In *Proc. of NDSS*, 2019.

[20] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. Measuring the insecurity of mobile deep links of Android. In *Proc. of USENIX Security*, 2017.

[21] Jiadong Lou, Xu Yuan, and Ning Zhang. Messy states of wiring: Vulnerabilities in emerging personal payment systems. In *Proc. of USENIX Security*, 2021.

[22] Gavin Lowe. A hierarchy of authentication specifications. In *Proc. of 10th Computer Security Foundations Workshop*, 1997.

[23] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Proc. of CAV*, 2013.

[24] Fadi Mohsen, Manar Alohaly, Usman Rauf, Dominic Therattil, and Loran Oosterhaven. PayScan: Detection and security analysis of payment libraries in Android apps. *International Journal of Information Security*, 24(4):189, 2025.

[25] Collin Mulliner, William Robertson, and Engin Kirda. VirtualSwindle: An automated attack against in-app billing on Android. In *Proc. of ASIA CCS*, 2014.

[26] Jeyamohan Neera, Xiaomin Chen, Nauman Aslam, and Biju Issac. A trustworthy and untraceable centralised payment protocol for mobile payment. *ACM Transactions on Privacy and Security*, 28(2):1–29, 2025.

[27] Michael Paik. Stragglers of the herd get eaten: Security concerns for GSM mobile banking applications. In *Proc. of ACM HotMobile*, 2010.

[28] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In *Proc. of NDSS*, 2014.

[29] Andreea-Ina Radu, Tom Chothia, Christopher J.P. Newton, Ioana Boureanu, and Liqun Chen. Practical EMV relay protection. In *Proc. of IEEE S&P*, 2022.

[30] Bradley Reaves, Jasmine Bowers, Nolen Scaife, Adam Bates, Arnav Bhartiya, Patrick Traynor, and Kevin RB Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications. *ACM Transactions on Privacy and Security*, 20(3):1–31, 2017.

[31] Bradley Reaves, Nolen Scaife, Adam Bates, Patrick Traynor, and Kevin R. B. Butler. Mo(bile) money, mo(bile) problems: Analysis of branchless banking applications in the developing world. In *Proc. of USENIX Security*, 2015.

[32] Daniel Reynaud, Dawn Xiaodong Song, Thomas R. Magrino, Edward XueJun Wu, and Eui Chul Richard Shin. FreeMarket: Shopping for free in Android applications. In *Proc. of NDSS*, 2012.

[33] Joeri De Ruiter and Erik Poll. Formal analysis of the EMV protocol suite. In *Proc. of Joint Workshop on Theory of Security and Applications*, 2011.

[34] Min Shi, Jing Chen, Kun He, Haoran Zhao, Meng Jia, and Ruiying Du. Formal analysis and patching of BLE-SC pairing. In *Proc. of USENIX Security*, 2023.

[35] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, and Luca Compagna. Attack patterns for black-box security testing of multi-party web applications. In *Proc. of NDSS*, 2016.

[36] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting logic vulnerabilities in e-commerce applications. In *Proc. of NDSS*, 2014.

[37] Pu Sun, Sen Chen, Lingling Fan, Pengfei Gao, Fu Song, and Min Yang. VenomAttack: Automated and adaptive activity hijacking in Android. *Frontiers of Computer Science*, 17(1):171801, 2023.

[38] Enis Ulqinaku, Julinda Stefa, and Alessandro Mei. Scan-and-pay on Android is dangerous. In *Proc. of IEEE INFOCOM*, 2019.

[39] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *Proc. of IEEE S&P*, 2011.

[40] Wenbo Yang, Juanru Li, Yuanyuan Zhang, and Dawu Gu. Security analysis of third-party in-app payment in mobile applications. *Journal of Information Security and Applications*, 48:102358, 2019.

[41] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in Android apps. In *Proc. of NDSS*, 2017.

A Payment Scenarios Illustrations

Three third-party online payment scenarios were discussed in Section 2.2. We provide detailed workflow diagrams for each of these payment scenarios, as shown in Figures 9–11.

In in-app payment, as shown in Figure 9, the workflow is as follows:

Step 1. The user selects goods in the merchant application and initiates a purchase request.

Step 2. The merchant application communicates with the merchant server to create the order and obtain the payment-relevant information required for the subsequent payment process.

Step 3. The merchant application invokes the payment provider's SDK to launch *PS-App* and delivers the payment information to *PS-App*.

Step 4. The user authorizes the payment within *PS-App* by entering the payment password.

Step 5. *PS-App* submits the payment request to *PS-Server*, which validates the request and proceeds with payment execution.

Step 6. The *PS-Server* performs the funds transfer to complete the transaction.

Step 7. The *PS-Server* sends a callback notification to the merchant server, enabling the merchant server to update the order status accordingly.

In mobile web payment, as shown in Figure 10, the workflow is mostly similar to that of in-app payment. The main difference lies in **Step 3**: the mobile web page launches *PS-App* via a dedicated URL scheme registered by *PS-App* at the operating system level.

In QR code payment, as shown in Fig. 11, the workflow differs from the previous two scenarios mainly in **Steps 1–3**,

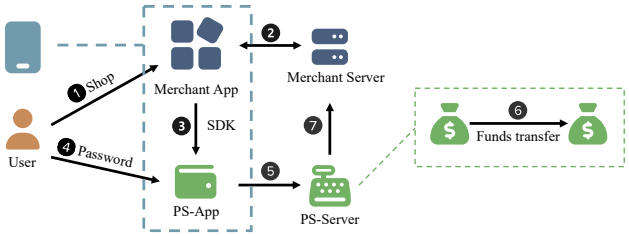


Figure 9: Payment Scenario I – In-app Payment

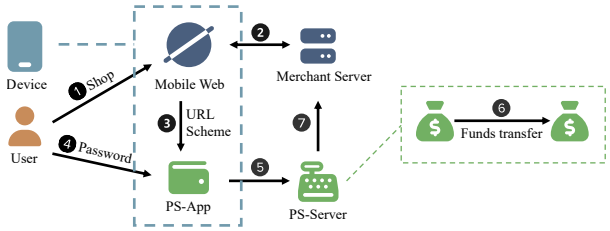


Figure 10: Payment Scenario II – Mobile Web Payment

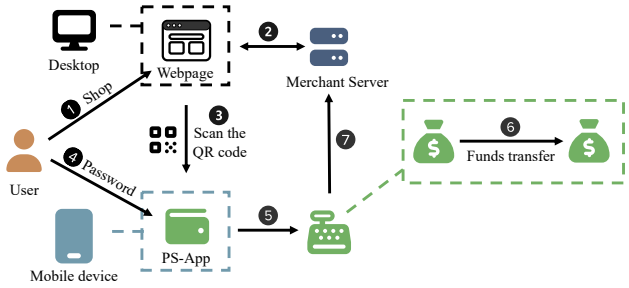


Figure 11: Payment Scenario III – QR Code Payment

which are described below. The remaining steps follow the in-app payment workflow.

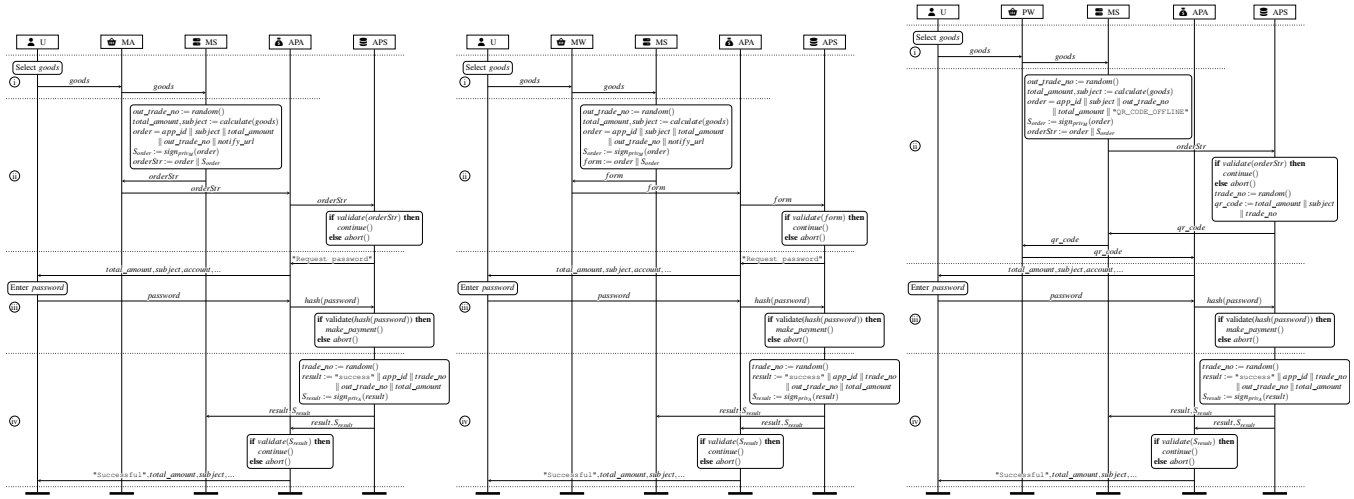
Step 1. The user selects goods in the merchant webpage and initiates a purchase request.

Step 2. The merchant webpage communicates with the merchant server to create the order and obtain the payment-relevant information required for the subsequent payment process.

Step 3. The merchant webpage displays a QR code encoding the payment information. The user scans the QR code on the mobile device, which launches *PS-App* and transfers the payment information to *PS-App*.

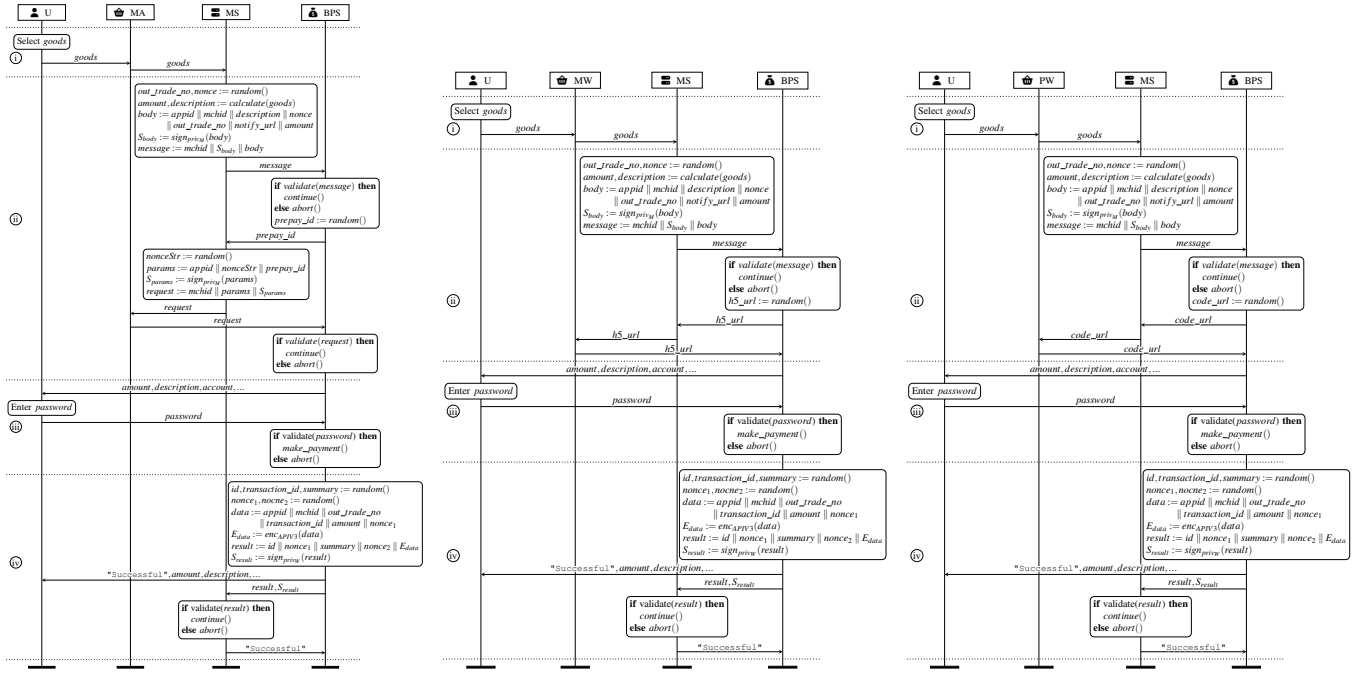
B Alipay & PSP-B Protocols

We summarize the Alipay and PSP-B protocols, reconstructed from their official developer documentation, across the three payment scenarios in Figures 12 and 13.



(a) Alipay In-App Payment (b) Alipay Mobile Web Payment (c) Alipay QR Code Payment

Figure 12: Alipay Payment Methods. Notations: *APA*: the Alipay payment application. *APS*: the Alipay payment server. out_trade_no: the order number of the *MS*. $total\_amount$: the payment amount. $subject$: the order title describing the purchased goods, which is displayed to the user during payment. app_id: the merchant application identifier. $notify\_url$: the merchant callback endpoint. $orderStr$: the signed order string generated by the merchant server. $trade\_no$: the Alipay transaction number. $form$: the HTML form. qr_code: the QR code. $priv_A$: the private key of *APA*.



(a) PSP-B In-App Payment (b) PSP-B Mobile Web Payment (c) PSP-B QR Code Payment

Figure 13: PSP-B Payment Methods. Notations: *BPS*: the PSP-B system, abstracting its externally visible payment functionality. $appid$: the merchant application identifier. *mchid*: the merchant identifier. $description$: the order description, which is displayed to the user during payment. $prepay\_id$: the prepayment token. id: the number of callback notification. $transaction\_id$: the PSP-B transaction number. $summary$: the summary notes for callback content. $nonce/nonce_1/nonce_2$: a fresh nonce for replay protection. $enc_k(m)$: encrypting message m under key k using AES symmetric encryption. $APIV3$: the symmetric key shared between *MS* and *BPS*. $priv_W$: the private key of *BPS*. $h5\_url$: an intermediate URL the merchant H5 page accesses to launch the PSP-B cashier and complete the payment. $code\_url$: the URL to be encoded into the QR code.

C Artifact Appendix

This artifact appendix provides a roadmap for evaluating the artifact associated with our paper, "When Authorization Loses Its Meaning: Breaking and Fixing Third-Party Online Payments".

C.1 Abstract

This artifact provides the formal models and verification framework used to analyze the security of third-party online payment protocols.

C.2 Description & Requirements

This section describes the steps and requirements needed to set up the artifact and reproduce the experiments.

C.2.1 Security, privacy, and ethical concerns

None. The formal models in this artifact are derived exclusively from publicly available developer documentation of the studied payment platforms. No reverse engineering, no inspection of proprietary or internal documents, and no testing against live accounts or production systems was performed.

C.2.2 How to access

The artifact is archived on Zenodo and can be accessed through the following DOI: https://doi.org/10.5281/zenodo.20303820.

C.2.3 Hardware dependencies

An x86-64 machine with at least 8 CPU cores, 32 GB of RAM, and 10 GB of free disk space.

C.2.4 Software dependencies

Ubuntu 24.04 with Docker installed.

C.2.5 Benchmarks

None

C.3 Set-up

This section describes the installation and basic test steps required to prepare the environment for evaluating the artifact.

C.3.1 Installation

1. Pull our pre-built Docker image.

   ```
   $ docker pull ghcr.io/luojiazhishu\
   /tamarin-docker/toolbox:latest
   ```

2. Download and extract the artifact archive from Zenodo (https://doi.org/10.5281/zenodo.20303820). After extraction, the directory structure is as follows:

   ```
   .
   |-- models
   |   |-- base
   |   |-- countermeasures
   |   |-- secure_ms_channel
   |   `-- secure_ps_channel
   |-- results
   |   |-- base
   |   |-- countermeasures
   |   |-- secure_ms_channel
   |   `-- secure_ps_channel
   |-- scripts
   |   |-- fix.py
   |   |-- parse.py
   |   `-- run.py
   |-- Makefile
   `-- README.md
   ```

   The top-level entries are organized as follows:

   - `./models/`: Tamarin formal models. The subdirectories `secure_ms_channel/`, `secure_ps_channel/`, and `base/` correspond to three channel security assumptions, each containing six payment-protocol models (two payment providers × three payment scenarios) — 18 protocol models in total. The formal models of our proposed countermeasures are under `countermeasures/`.
   - `./scripts/`: Helper scripts for running Tamarin and aggregating its output into JSON/CSV summaries.
   - `./results/`: Verification results produced by Tamarin, mirroring the subdirectory layout of `./models/`.
   - `./Makefile`: Entry point that orchestrates the full evaluation pipeline.

C.3.2 Basic Test

To verify the environment, run:

```
$ docker run --rm ghcr.io/luojiazhishu\
/tamarin-docker/toolbox:latest \
  tamarin-prover test
```

If the installation is successful, the command output contains the following test summary:

```
*** TEST SUMMARY ***
All tests successful.
The tamarin-prover should work as intended.

           :-) happy proving (-:
```

C.4 Evaluation workflow

The evaluation workflow involves formal model verification using the Tamarin Prover.

C.4.1 Major Claims

(C1): The verification results of six third-party online payment protocols under three channel assumptions reproduce Table 1 in the paper. Supported by experiment **(E1)**.

(C2): The verification results of our countermeasures reproduce Table 2 in the paper. Supported by experiment **(E1)**.

C.4.2 Experiments

(E1): Verification [3 human-minutes + 1 compute-hour]: Use the Tamarin Prover to check the security lemmas of every formal model in `./models/`. The resulting *verified/falsified* outcomes should match Table 1 and Table 2 of the paper.

How to: Run the verification in three steps.

Preparation: With the installation in Section A.3 completed, launch the Tamarin toolbox container from the **artifact root directory**:

```
$ docker run -d \
  --name tamarin-toolbox \
  -p 3001:3001 \
  -v "$(pwd):/root" \
  -w /root \
  ghcr.io/luojiazhishu\
/tamarin-docker/toolbox:latest
```

Execution: Enter the container, clear any pre-existing results, and run the verification pipeline:

```
$ docker exec -it -w /root \
  tamarin-toolbox bash
$ make clean
$ make all JOBS=1
```

On the reference hardware (8 cores, 32 GB RAM), `JOBS=1` completes the full verification in about one hour. On a more powerful machine (e.g., 32 cores and 64 GB RAM), increase parallelism with `make all JOBS=8` to reduce the total time to roughly 30 minutes.

Results: The top-level file `./results/results.csv` aggregates the verification outcome of every lemma across all models; its rows reproduce Table 1 and Table 2 of the paper. Each subdirectory of `./results/` corresponds to one channel security assumption (or to the countermeasures) and contains one `.spthy` proof file per protocol plus a `result.json` summarising that subdirectory. To inspect a proof graph, launch Tamarin's web interface inside the container; for example, to view the proofs under `./results/base/`, run:

```
$ tamarin-prover interactive \
  --interface=0.0.0.0 \
  --derivcheck-timeout=0 \
  ./results/base
```

Then open `http://127.0.0.1:3001` in a browser on the host machine.

The Tamarin lemma names that appear in `results.csv` and in each `result.json` correspond to the security properties defined in the paper as follows:

- `PS_MS_Order_Injective_Agreement`: IA-O1.
- `U_MS_Order_Injective_Agreement`: IA-O2.
- `MS_PS_Response_Injective_Agreement`: IA-R1.
- `U_PS_Response_Injective_Agreement`: IA-R2.
- `No_Payment_Without_Password_Input`: IA-UA.

C.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.