Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis

Yi Chen[1,3], *Luyi Xing*[2], *Yue Qin*[2], *Xiaojing Liao*[2], *XiaoFeng Wang*[2], *Kai Chen*[1,3], *Wei Zou*[1,3]
[1]*{CAS-KLONAT, BKLONSPT, SKLOIS},Institute of Information Engineering, CAS,* [2]*Indiana University Bloomington,*
[3]*School of Cyber Security, University of Chinese Academy of Sciences*
{luyixing, qinyue, xliao, xw7}@indiana.edu, {chenyi, chenkai, zouwei}@iie.ac.cn

Abstract

Finding logic flaws today relies on the program analysis that leverages the functionality information reported in the program's documentation. Our research, however, shows that the documentation alone may already contain information for *predicting* the presence of some logic flaws, even before the code is analyzed. Our first step on this direction focuses on emerging syndication services that facilitate integration of multiple payment services (e.g., Alipay, Wechat Pay, PayPal, etc.) into merchant systems. We look at whether a syndication service will cause some security requirements (e.g., checking payment against price) to become unenforceable due to losing visibility of some key parameters (e.g., payment, price) to the parties involved in the syndication, or bring in implementation errors when required security checks fail to be communicated to the developer. For this purpose, we developed a suite of Natural Language Processing techniques that enables automatic inspection of the syndication developer's guide, based upon the payment models and security requirements from the payment service. Our approach is found to be effective in identifying these potential problems from the guide, and leads to the discovery of 5 new security-critical flaws in popular Chinese merchant systems that can cause circumvention of payment once exploited.

1 Introduction

Logic vulnerabilities are a category of security defects caused by faulty program logic, which have long been known to be hard to analyze, due to their close ties to specific functionalities of a system. Finding these defects relies on evaluating the target system's behaviors, at the code level, against a set of invariants describing its function-related security properties (e.g., expected authentication and authorization operations). A great source for such invariants is the system's documentation, which states its security goals and has been leveraged by prior research for the purposes such as model checking on the system's code [27]. In the meantime, the documentation also provides detailed accounts of how the system is designed to achieve the goals and how it should be used to remain secure. Such information can be valuable for *predicting* whether security-critical logic flaws are present in the system or applications that use the system: for example, prior research shows that logic vulnerabilities could be related to low-quality documentations, such as those missing explanations about implicit assumptions for secure integration of authentication SDKs [49]; also a conflict between the system's operations, as described in its documentation, and its security goals may indicate the existence of a design flaw. However, never before has any effort been made to dig into the details contained in a myriad of software documentations to understand their security implications, not to mention any attempt to exploit their full value for logic flaw detection.

Logic vulnerabilities in payment syndication. In this paper, we present preliminary evidence that system documentations indeed carry abundant vulnerability-related indicators, which can help identify logic flaws *even before the code of the system has been analyzed*. Further we show that this documentation-based approach can be automated, enabling more effective vulnerability detection through providing guidance to program analysis such as fuzzing. This first step has been made possible by a study on *syndication services* that facilitate integration of various payment services (e.g., Alipay [2], WeChat Pay [17], PayPal [25]) in mobile apps. These payment services have different APIs and SDKs, and an app often needs to use all of them to offer its customers different payment options. To simplify integration of these options, a syndication service encapsulates each payment service with a *wrapper* that exposes to the developer a uniform interface. This, however, injects the syndicator as a proxy into the already complicated payment interactions among the payer (the app user), the payee (the merchant) and the payment service provider (e.g., Paypal). Logic flaws can therefore be induced

*Work was done when the first author was at Indiana University Bloomington.
†Key Laboratory of Network Assessment Technology, CAS.
‡Beijing Key Laboratory of Network Security and Protection Technology
§State Key Laboratory of Information Security, IIE, CAS

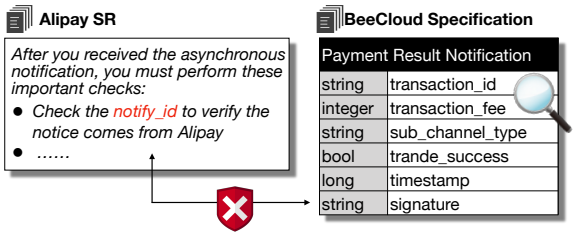


Figure 1: Incosistency example.

at the design level, when the wrapper causes some security checks hard to proceed (e.g., verification of payment against price), or at the implementation level, when the developer fails to perform security checks correctly, due to incomplete instructions given by the syndicator.

These payment syndication services today become increasingly popular: according to the reports [35, 36], their total transactions in year 2018 have reached 21.1 trillion yuan (3.15 trillion dollars). Any vulnerability inside these services, once exploited, will have significant impacts, affecting 251 million syndication users world-wide. We believe that their documentations, developer's guide in particular, contain information that can help predict or even detect the logic flaws in their customers' systems. As an example, Figure 1 demonstrates the inconsistency discovered when comparing an Alipay's security requirement, which asks for inspecting `notify_id`, with the payment notification issued by BeeCloud [6] (a popular service syndicating Alipay and other payment services), which does *not* include this information, as discovered from its developer's guide; therefore, the merchant server receiving the notification will *not* be able to perform the required check. This logic problem has been *confirmed* in our research. A question here is how to systematically identify such logic vulnerabilities from documentation. Also, developer's guides are typically long and complicated, including a lot of irrelevant information (e.g., instructions for using the syndicator's tools like dashboard). For example, Ping++ has 1093 KB text documents online with at least 278 KB related technical content [12]. Inspecting all the content manually is both time-consuming and error-prone. Automated techniques therefore need to be developed to help vulnerability discovery.

Document analysis for flaw detection. In our research, we developed *Dilution* (Documentation Inspector for Logic Vulnerability Prediction), a new technique that automatically analyzes the developer's guide of a payment syndication to infer potential security flaws in the merchant systems integrating the service. Dilution is designed to predict missing security checks in the integration, which is caused by either improper encapsulation of a payment service that renders its security checks impossible to perform through the wrapper, or failure in communicating the necessary checks to the developer through the guide. For this purpose, we utilize natural language processing (NLP) to automatically recover semantic information from the wrapper's integration instructions documented by the developer's guide and compared it against the finite state machine (FSM) of the payment service encapsulated. Note that this payment FSM was manually extracted but considered as one-time efforts (Section 3.1). Our analysis automatically infers the relation between the syndication payment process and the payment FSM, maps important payment states to the related instructions in the guide and further recovers the parameters for required security checks (e.g., Alipay key for signature verification, price) at the state from the text. By analyzing the merchant or syndicator's visibility of the parameters, we can determine whether these checks can still be performed by the merchant or the syndicator. Also from related descriptions in the guide, our approach can automatically find out whether the developer is informed about these security requirements when integrating the wrapper. Missing such instructions indicates the possible absence of security checks in the merchant's code.

We implemented Diluation using a suite of NLP techniques, including dependency parsing and word embedding, and evaluated it on labeled content extracted from the developer's guides of real-world syndication services. Our study shows that Dilution accurately caught logic flaws and went through 182 KB text document content within 3.18 seconds, averagely.

Finding. Further, we ran Dilution on the documentations of eight popular syndication services, including Ping++ [12], Paymax [11], BeeCloud [6], etc., which have tens of thousands of merchants each and power the apps with millions of users. From 1,456 KB documentations, our approach automatically predicted totally 41 potential issues, including 11 highly likely to be logic flaws from five syndication services: Fuqianla [8], BeeCloud, TrPay [15], UMF Pay [16] and 66zhifu [1]. All the issues reported were found to be accurate by our manual inspection of the documentations. Despite the challenges in finding the apps integrating these services, due to the obfuscations these services suggest [20, 23, 26], we collected 17 popular Chinese apps using two of these syndication services. Through a black-box testing on these apps and their merchant systems, we concluded that all 5 logic flaws related to these syndication services predicted by Dilution are indeed present in either the syndicators' systems or their customers' code. All such confirmed flaws are security-critical, and once exploited, will have serious consequences, allowing the adversary to shop at a lower price or even for free. We reported our findings to the providers of the syndications and the merchants who are affected, and they all acknowledged the importance of the problems we discovered. Now we are in the process of helping them fix these vulnerabilities. Video demos of our attacks are posted online [4].

Contributions. The contributions of the paper are outlined as follows:

• *New direction*. We explore the potential to predict the presence of logic vulnerabilities in a software system from its documentation. Our preliminary study on payment syndica-

tion services shows that this is indeed feasible. Research along this line could bring in a new perspective to software security analysis, enabling more effective and intelligent vulnerability detection and helping enhance software security quality.

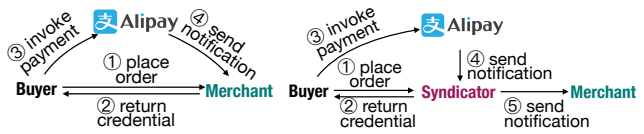
• *New techniques*. We developed Dilution, the first semantics-based documentation analyzer, to automatically inspect the developer's guide and infer possible security fallacies in the merchant's integration of the syndication service. Our approach includes a suite of NLP techniques tuned towards software documentation, which are found to be effective and efficient, as demonstrated by our evaluation.

• *New findings*. We analyzed the developer's guides of 8 most popular syndication services using Dilution and discovered potential security issues. Among these we can validate, we confirmed that all 5 logic flaws predicted by our approach are indeed present in syndicator or merchant systems. These vulnerabilities, once exploited, allows the adversary to shop at an arbitrarily low price he set or completely for free, affecting millions of users. We are working with affected syndicators and merchants to fix these problems.

2 Background

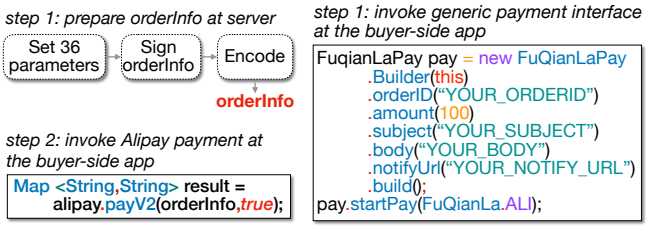
Payment and syndication service. A third-party payment service (aka. a *payment processor*) like PayPal [25] is an Internet service to help handle transactions between the buyer (the payer) and the seller (the payee) [43]. Such a service simplifies transaction managements on both the payer and the payee sides and therefore plays an important role in e-commerce. Figure 2(a) shows how the service works using Alipay [2], the largest online payment processor with over 1 billion users around the world [47], as an example. The buyer first places an order through the app (①) and receives a payment-related `credential` from the merchant (②), including order ID, price, seller account and others, and then forwards it to Alipay (③). After the order is paid by the buyer, Alipay issues a notification to inform the merchant the completion of the transaction (④). The interfaces exposed by these payment services tend to be complicated. Figure 3(a) further details the process the app developer (working for the merchant) is supposed to do for using Alipay : the merchant server generates an Alipay specific argument `orderInfo` (part of `credential`) with 36 entries once the buyer places an order; then the buyer-side app of the merchant invokes Alipay's payment service.

To simplify this integration process, syndication services emerge to wrap different payment processors into a uniform interface for the developer to conveniently incorporate them into her app. Figure 2(b) and 3(b) shows a common syndication payment process for Alipay. Instead of asking merchant developer to implement anything specific to Alipay, the syndicator receives an order from the buyer on behalf of the merchant (①), construct the `credential` (②) and then invokes Alipay payment service from the buyer's app (③). Note that,



(a) Payment process (b) Syndication process

Figure 2: Examples of payment and syndication process.



(a) To integrate Alipay (b) To integrate a syndicator Fuqianla

Figure 3: The implementation examples.

③ only requires the app to invoke a generic payment interface provided by the syndicator's SDK in the app, and the target payment processor, i.e., Alipay in this case, will be invoked by the syndicator's SDK. Once the payment is done, the syndicator receives the notification from the the payment processor (④) and restructures the message to a uniform format before forwarding it to the merchant server (⑤). In this way, a simple integration of a single syndication service on both the app end and the merchant server end will allow the merchant to work with multiple payment processors supported by the syndicator. The developer is only supposed to follow a single set of instructions from the syndicator to ensure the secure payment process.

Security requirements. Online payment is a security-critical process, so it is safeguarded by various security checks performed both by the payment processor and by the merchant, as required by the payment processor on its developer's guide. We call these checks *security requirements* (SRs)[1] throughout the paper. For instance, most third-party payments ask their merchants to verify the payment amount on the notification against the price of the purchase, the seller account information to ensure that the merchants are intended payees, etc. In Section 4.2, we present more examples for common security requirements. In the case that the payment syndication is used, we expect that either the developer or the syndicator is still at the position to perform these required checks, and also in the former case, the developer should be properly informed through the guide provided by the syndicator.

Natural language processing. In our research, we utilized two NLP techniques to automatically analyze the documentation of the payment syndication service: *dependency parsing* and *word embedding*, as explained below.

[1] All the abbreviation's explanation are summarized in Table 5 at Appendix for convenience.

Table 1: Examples of the relations between linguistic units

Example sentence: *She gave me a very happy smile and a hug.*

Abbreviation	Description	Relation example
SBV	Subject-verb	She <- gave
VOB	Verb-object	gave ->smile, gave ->hug
IOB	Indirect object	gave ->her
ATT	Attribute	happy <- smile
ADV	Adverbial	very <- happy
COO	Coordinate	smile ->hug

Dependency parsing is an NLP technique to reveal the syntactic structure of a sentence by analyzing the grammatical relations between linguistic units such as words. Examples of such relations include subject-verb(SBV), verb-object(VOB), indirect object(IOB), attribute(ATT), adverbial(ADV), coordinate(COO) and others (see detailed explanation and examples of these relations in Table 1). The result of the dependency parsing is represented as a rooted parsing tree. At the center of the tree is the verb of a clause structure, which is linked, directly or indirectly, by other linguistic units. The state-of-the-art dependency parser (e.g., Stanford parser [31]) can achieve a 92.2% accuracy in grammatical relation discovery from a sentence. In our study, we leveraged the parsing tree generated from sentences in a developer's guide to locate the parties involved in a payment process and the content transmitted between them.

Word Embedding is a set of language modeling and feature learning techniques that map text (words or phrases) from a vocabulary to high-dimensional vectors of real numbers. Such a mapping can be implemented in different ways. The state-of-the-art word embedding tool, *Word2vec* [37], initializes word representations by random values and uses as its input a joint probability distribution of words' context by applying a continuous Bag-of-Words or a skip-gram model. This distribution is then utilized during the training of a neural network, in which word vectors are continuously updated to maximize the joint probability. The outcome of the training ensures that related words are given approximate vectors for their similar contexts while irrelevant words are mapped into different vectors. In our study, we leveraged *Word2vec* to generate a semantic vector for each word and measured their semantic difference by cosine distance between vectors.

Threat model. In our research, we consider a malicious buyer who intends to get a product for free or at a lower price by exploiting the vulnerabilities in the payment process, particularly the logic flaws caused by incorrect or inadequate security checks on the merchant or the syndicator side. This adversary has the capability to modify or forge the messages delivered to both the merchant server and the syndicator.

3 Dilution: Design

3.1 Overview

We believe that a critical security goal of a payment syndication is to ensure that all the security requirements made by the payment processor it wraps are met through proper

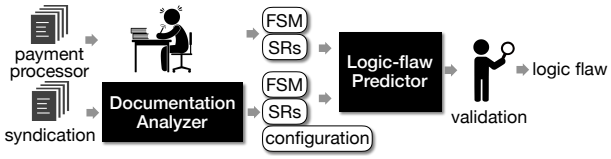


Figure 4: Architecture of our approach.

security checks, either by the merchant integrating the service or by the sydnicator itself. The purpose of Dilution, therefore, is to identify from the developer's guide of the syndication *indicators* that some SRs may fall through the cracks.

To this end, our approach is designed to compare the information observed by the syndicator and the merchant (including its app) with the SRs expected at individual states of the original payment process (e.g., that of Alipay), to determine whether either of the syndicator or the merchant has enough information to fulfill these requirements at the states. Also analyzed is whether these SRs have been explicitly stated in the developer's guide provided by the syndicator, when related checks need to be done on the merchant's side. More specifically, we first extract an FSM from the payment process, with some of its states associated with SRs and the parameters they are predicated on. This FSM is then extended to include the operations performed by the wrapper, based upon the information automatically recovered from the guide. Further using the text content related to each state of the extended FSM, we evaluate whether all parameters of each SR at the state are still visible to the merchant or the syndicator. Finally, our approach automatically analyzes the content of the guide to determine whether the SRs are explained to the developer.

Architecture. Figure 4 illustrates the architecture of Dilution, including a preprocessing component, *Documentation Analyzer* (DoA), *Logic-flaw Predictor* (LfP) and a validation component. The preprocessing step is done manually in our current system, which involves extraction of the FSM and label of SRs for a payment processor. Since our focus is the syndication service, so we consider this step as a one-time effort: for each third-party payment, the information identified can be used to automatically analyze tens of syndication services, each with a development guide containing hundreds of thousands of words. Such documentations are inspected by DoA, which utilizes NLP techniques to extend the payment FSM with the states representing the wrapper's actions, and further recover the description for each SR from the guide. The extended FSM and the parameters are then analyzed by the LfP to identify the SRs that cannot be fulfilled by the merchant and the syndicator, and those that have not been properly explained to the developer. Also, the logic flaws predicted by LfP are validated on the merchant's integration of the syndicator (the merchant's app and its server-side component) to confirm the presence of these vulnerabilities.

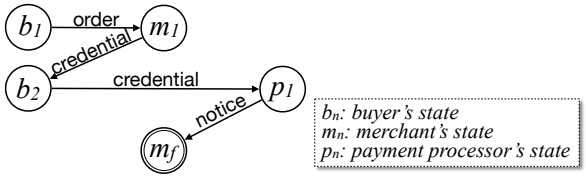
Example. Here, we use an example to describe how our approach works. Figure 5(a) shows Alipay's FSM, with one of

its SRs at state m_f being verification of the payment amount against the price of an item. Analyzing the developer's guide from Fuqianla (a popular syndicator), DoA discovers that the order has been sent to the syndicator, instead of the merchant ("The merchant client sends a payment query to the Fuqianla server"), and the notification from Alipay is delivered to the syndicator ("The Fuqianla server will receive a payment notification from the payment service"), before it is forwarded to the merchant . The semantics recovered from the text is then used to replace state m_1 in Figure 5(a) with state w_1 and add state w_2 to the FSM, converting it to the syndication FSM as shown in Figure 5(b). This extended FSM and all the parameters it carries is then further inspected by LfP. Here let us look at the aforementioned SR at m_f. At this state, LfP concludes that the syndicator cannot check the payment amount, as it gets the price information from the buyer not the merchant, which cannot be trusted. On the other hand, though the merchant is at the position to make the security check, LfP cannot find a sentence, right after the description of the last communication (from w_2 to m_f), informing the developer about the SR, through a dependency analysis on all the sentences involving terms related to the subject (the merchant), the object (price and payment amount) and the expected action of the security check. This leads to the conclusion that the SR may not be communicated to the developer, and so may not be fulfilled on the merchant side.

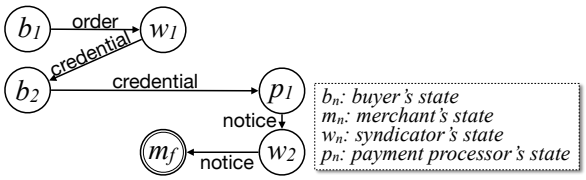
3.2 Preprocessing

As mentioned earlier, our approach involves a one-time preprocessing step in which the FSMs of major payment processors (Alipay, Wechat, PayPal, etc.) are constructed and the SRs for different states are identified. Such information can typically be found from these payment services' integration guides. For example, Figure 11 at Appendix shows the excerpts from Alipay's documents. Its payment process is clearly described through a diagram, which can be easily converted into an FSM. From the figure, we can also see the content for SRs, their parameters and the relations with the payment process. Following we present the FSM that models the payment process and the SRs. From such content, we extract the payment process model and security checks, as formally described below.

Payment model. The FSM for a payment process is described as a 5-tuple: ($\boldsymbol{S}$, $\boldsymbol{D}$, $\boldsymbol{E}$, b_1, m_f). Here $\boldsymbol{S}$ is a set of payment states, in each of which an actor (buyer b, merchant m or syndicator w) can send out a message $d \in D$; $E : S \times D \to S$ is a function that drives the transition from one payment state to the next, given a specific message $d \in D$ sent out from the former; b_1 is the initial state in which the buyer places an order to start the whole payment transaction, and m_f is the final state that the merchant receives the last message and the transaction is complete. For example, Figure 5(a) illustrates such an FSM for Alipay.



(a) The FSM of the payment processor using Alipay



(b) The FSM of the syndication using Fuqianla

Figure 5: FSM examples of the payment processor and syndication.

Security requirement. A security requirement SR describes a security check that needs to be performed at a certain state: for example, at m_f of Alipay's FSM (Figure 5(a)), the merchant is supposed to verify *payment* = *price*. As we can see from the example, central to the SR are the state (m_f), subject (*merchant*), object (*payment*), a verification function (the *equal function* in the example) and additional parameters for the verification (*price*). However, in the context of the payment FSM, each state is bound to an actor who is actually the subject performing a security check. Further since all we care is the feasibility of fulfilling the SR at a given state, we just need to know whether all the inputs of the verification function (object and other parameters) are visible to the subject at the state, not the function itself. Therefore, we can simply model an SR as a 3-tuple: (SR_{state}, SR_{obj}, SR_{para}), to represent its state, object and other parameters for the expected security check. Note that the object and parameters here are *types of information*, i.e., the *key* part of a key-value pair. This is because all we want to know is whether the merchant at a state can see these keys (*payment* and *price*) so as to perform the required check; the outcome of the check, which depends on their specific values, is not important for our purpose.

3.3 Syndication FSM Discovery

With the FSM and SRs collected from a payment processor, we can run DoA to automatically analyze the developer's guides of different syndication services that wrap the processor. Most important here is to discover the syndicated payment process through extending the payment FSM, which further allows us to find out whether the parameters of required security checks are still visible to the new states supposed to perform the checks, and how these security requirments are explained to the developer (Section 3.4).

FSM extension structures. Fundamentally, a syndication service is meant to support a merchant's interactions with the payment service, in terms of generating the input for the

service and converting its output to a unified form easy for the merchant to interpret. This observation allows us to come up with a set of possible *extension structures*, which are then confirmed by the evidence extracted from the syndication's document (its developer's guide).

Specifically, given a payment processor's FSM, we consider two types of states[2] that the syndicator can help: m where the merchant generates an input for the payment service, and m_f when the merchant receives the final notification from the payment service. More specifically, the operations at the state m can be assisted by a syndication state for input construction; payment notification the merchant finally receives at m_f can be converted by a syndication state first. Following this line of thinking, we can identify all possible extensions of m and m_f, which are present in Figure 6. As we can see here, for m, the extensions include the situations when either the merchant or the syndicator produces the full output of m, that is, the input for the payment service (see a.1 and a.4), and those when they jointly create the output (see a.2, a.3, a.5 and a.6). The latter can be further broken down into the cases when the same party receives and issues the messages related to m (see a.2 and a.5), or when different parties do (see a.3 and a.6). For m_f, since the payment process must end at this merchant state, only three situations exist: no extension (see b.1), the merchant getting the input (see b.2) and the syndicator receiving the original input (see b.3). DoA automatically inspects every merchant state and evaluates the consistency between each possible extension structure and the evidence discovered from the developer's guide through NLP, to find out how the syndication indeed happens.

Taking the syndicator Fuqianla as an example (see Figure 5(b)), through inspecting it documentation, DoA determines that Fuqianla uses the extension structures a.4 and b.3 to wrap the payment processor. Specifically, its state w_1 replaces the original state m_1 to generate credential, which is an input for invoking the payment processor; the syndicator at state w_2 receives the payment notification from the merchant and converts it to a unified form (across different payment services it supports) before forwarding it to the merchant at state m_f.

Extension discovery from document. The evidence collected from the syndication developer's guide is the sentence that describes the message transferred from a sender to a receiver in the FSM (the buyer, the merchant, the payment provider and the syndicator). For example, from the sentence "the merchant client sends a payment query to the Fuqianla server", we know that the payment query from the buyer has been sent to the syndication server, not the merchant, which confirms the existence of a transition from b to w in the extension structure a.5 and a.6 (Figure 6). Our idea is to find all

[2]Note that we are only interested in these states because per payment services' guides [2,17,25], the inputs they accept are supposed to be generated by the merchant server and the outcome of a transaction will be delivered to the server.

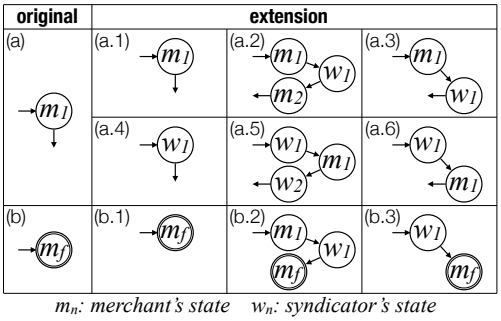


Figure 6: FSM possible extension structures.

such descriptions and extract their transition-related *semantic information*, in the form of (*Sender*, *Receiver*, *Content*), to identify all transitions introduced by the syndication and further determine the way merchant states have been extended.

For this purpose, we utilized a suite of NLP techniques to first find out all sentences related to transmission activities (e.g., including predicates like "send", "receive", etc.), then performs a syntactic analysis on each sentence and further converts detected syntactic elements to a *semantic triplet* describing the parties involved in a transition as well as the message sent (*Sender*, *Receiver*, *Content*). The challenges here come from the ambiguity of the descriptions and diversity of sentence structures in the developer's guide. Particularly, we found that a variety of terms are used to describe message delivery and reception: not only common synonyms like "transmit", "dispatch", etc., but also those specific to the integration domains such as "call" (a remote function) and "invoke" (a remote client).

To identify those synonymous terms, we leverage the observation that such expressions, no matter how diverse they are, all share the similar context. For example, from the sentences "call the merchant API to place an order" and "send the order to the merchant", we know that "call" and "send" are semantically close given their relations with 'merchant' and 'order'. So in our research, we trained a *word embedding* model [18] over the documentations of two syndicators, Ping++ [12] and Fuqianla [8], and two payment service providers, Alipay [2] and Wechat Pay [17]. The model maps each word to a vector that represents its context. So the cosine distance between the vectors quantifies their semantic similarity. In our research we first manually collected a small set of "seeds", words semantically related to "send" and "receive", such as "call" and "invoke", and then built a synonym list for these words with the embedding model we trained over the aforementioned documentations, using LTP [19] to segment Chinese words. These lists are utilized to identify sentences in a developer's guide involving these transmission-related terms.

On each sentence discovered, DoA needs to extract its semantics – the triplet. For this purpose, we come up with a unique technique that utilizes dependency parsing (LTP [19]) to first identify a sentence's syntactic elements (subject, object, etc.) and then determine their semantics (*Sender*, *Receiver*,

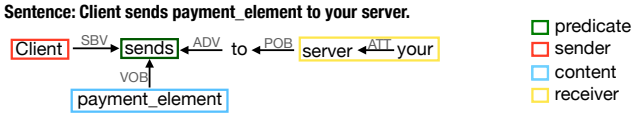


Figure 7: Entities in data-transmission related sentence.

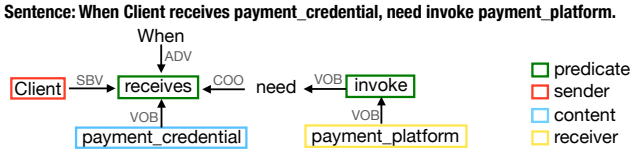


Figure 8: Entities in complex sentence.

Content). This is quite intuitive for a simple sentence. For example, Figure 7[3] shows the dependency relations between the predicate "send" and other words or phrases. As we can see here, the subject of the predicate ("client") is *Sender*, direct object ("payment_element") is *Content* (the message delivered), and indirect object ("your server") is the receiver. These elements can then be mapped to a transition on the FSM, based upon their semantic similarity with payment parties, as measured by the distances between their vectors. However, the semantics of the syntactic elements become more difficult to determine in the presence of more complicated sentence structures. For example, Figure 8 shows a complex sentence with multiple clauses, including both "When Client receives payment_credential" and "(Client) needs to invoke payment_platform". In this case, the subject of "receives" ("Client") becomes *Sender*, the object of "invoke" ("payment_platform") is *Receiver* and the *Content* in the sentence is found to be the object of "receives" ("payment_credential"). To address this challenge, we trained an SVM classifier on a labeled dataset with transmission related sentences discovered from payment documentations and syndication documentations. The model uses the predicates and their relations (e.g., order) as features to predict their subject, object and indirect object's semantic class labels (*Sender*, *Receiver* or *Content*).

Using the triplets recovered from the sentences, DoA continues to inspect each merchant state to determine whether and how it is extended by a syndicator. Specifically, for each transition in the extension structures described in Figure 6, denoted by $s' = E(s,d)$, we try to align *Sender* to the name of the actor at s (e.g., "merchant"), *Receiver* to the actor of s' (e.g., "syndicator") and *Content* to d (representing message *name* here, such as "order"). Note that in the case any of these entities is described by a phrase, instead of a word (e.g., "payment element"), we calculate its *phase vector* as the average of its individual word vectors (e.g., those of "payment" and "element"). We consider that an alignment succeeds when all these elements are found to be similar to their counterparts on the transition. When this happens, we believe that this transition exists in the extended FSM. To discover more transitions, our approach also leverages partial information collected, when only two elements of the triplet has been recovered. If one of these elements is *Content*, DoA still compares them against the transitions and confirms the presence of a transition if an alignment is found.

Based upon all the transitions discovered, our approach further determines the extension structure used by a syndication: the one contains all these transitions is selected. When there are more than one such structures, we consider that all such extensions are possible and predict the presence of potential logic flaws if one of them is found to be problematic.

3.4 SR Information Discovery

The syndication FSM discovered tells us how a payment transaction proceeds among the buyer, the merchant and the payment service provider, in the presence of the syndicator. To find out whether all security checks required by the payment service can be performed on this new FSM, we need to take a further look at the information visible to the states that need to fulfill these security requirements. Also to be understood is how the SRs are presented to the developers who are supposed to integrate these checks in merchant-side code. All such information has been automatically recovered from the syndication developer's guide, as elaborated below.

API parameter discovery. At a syndication or merchant state a security check is expected to happen (SR_{state}), the information required for the check (SR_{obj}, SR_{para}) either *comes from the message it receives (e.g., payment_result) or is already in the possession of the syndicator or the merchant*. In the former case, the communication with the syndicator always goes through its APIs, as integrated in the merchant's client or server side code. These APIs are documented in the developer's guide and their attributes describe the information passed by a message. A question is how to determine which API is used in a transition. Such an API is explicitly mentioned in some sentences, such as "call the Creating Charge to invoke a payment processor". In our research, we utilized a SVM model for labeling syntactic elements to detect the API names, which serve as the object of "call".

However this approach turns out to be inadequate, since more often than not, a transition-related sentence in the developer's guide does not include any API name: for example,"initiate payment". In this case, we found that the semantics of the message name (d in the transition and its corresponding *Content*) is always highly related to the name of the API to be used. In the above example, an API "initiate payment" is responsible for sending the message of payment requirement. This is understandable since the guide is supposed to inform the developer how to establish communication with the syndicator. If this has not been done explicitly, using semantically related API names is an implicit way to do so. Our DoA automatically identifies such APIs using our

[3]Figure 7, Figure 8 and Figure 10 show Chinese grammatical relations between words. The words shown in the figure were translated from Chinese.

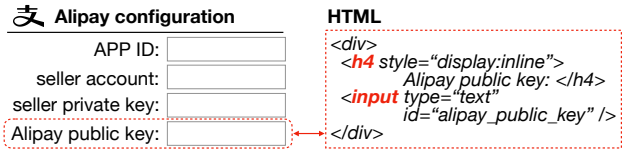


Figure 9: An excerpt of configuration HTML.

word-embedding model to compare the phrase similarity between each API name and d and *Content*. Again, here we utilized LTP [19] for Chinese word segmentation and word extraction from API names. Whenever an API is found to be semantically similar to either d or its related *Content*, our approach collects all its attributes and consider that their values have been exposed to the actor at the state receiving d.

Configuration information extraction. As mentioned earlier, the information for a security check can also be provided to the actor in a certain state before a transaction happens. For example, the merchant has her private key for signing a payment credential and she can also delegate this task to the syndicator by configuring her account on the syndication website. Such preconfigured information is documented by the developer's guide. However, the details are often included in images, together with those irrelevant ones for explaining the payment or syndication service. Content extraction from these images using OCR [39] did not work well in our study. So our DoA is designed to discover the configuration data directly from the syndication website, the one from which we collect the developer documentation.

Specifically, our approach first searches for the entry link labeled with "Alipay configuration", "Wechat Pay management", "PayPal setup", etc., the standard names for the configuration page on a syndication site, using named entity recognition and keywords (e.g., Alipay) together with synonyms for "configuration", etc. On such a page, we inspect its HTML tags, looking for the input type – the entry item for the merchant to enter her data, and its inline header, which is the *key* for the data. Figure 9 shows part of the configuration page of Ping++ [12]. From the text entry identified, we can recover its header "Alipay public key" under the tag *h4*. Also as we can see from the example, other keys that can be found from the configuration page includes APP ID, seller account, etc. The information is gathered for comparing with SR parameters for a given payment service such as Alipay.

SR description recovery. Also important to logic vulnerability discovery is to find out whether required security checks have been properly explained to the developer. For this purpose, DoA has also been made to search for the description of the SRs for a given payment service. Such an SR is typically presented in a sentence: e.g., "The merchant should verify whether the payment amount equals to the order price." From the sentence, we know that the merchant is the party responsible for this SR, so the check is supposed to happen on a merchant state (SR_{state}), payment amount is the object

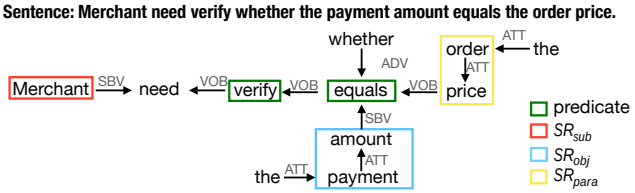


Figure 10: Entities in sentences describing SR.

(SR_{obj}) and the order price is the additional parameter for the security check (SR_{para}). Our idea is to automatically discover all SR-related sentences based upon the actions to be taken, as we did in the FSM extension discovery (see Section 3.3), and then perform a syntactic analysis to discover SR_{obj} and SR_{para}, before finally determining the state of the potential check (SR_{state}).

Specifically, to find all SR-related sentences, our approach first utilizes a small set of *seed action terms* including "check", "match", "verify", etc., and runs our word-embedding model on the training documents (guides for syndication and payment services) to extend these seeds with their synonyms. Then DoA inspects a given document to collect all the sentences containing the term(s) on the list. Each of them is analyzed using dependency parsing to label the subject, object and indirect object of the action term ("verify", "check", etc.), as demonstrated by the example in Figure 10. Given the fact that the developer's guide is meant to explain implementation details to the developer working for the merchant, we expect that an SR-related subject should be "merchant", "developer", "you", and their synonyms (e.g, "seller","verdor", etc.). Also we consider the merchant to be the subject of all imperative sentences, e.g., "please make sure that the payment amount equals to the order price". From these sentences, we further identify the object and indirect object (if exists) of the predicate, and label them as potential SR_{obj} and SR_{para}.

Before we can report possible SR-related description, we also need to determine the state for the potential security check (SR_{state}). Our approach is to look at where the sentence is found: intuitively a reminder of a security check should appear under the context of state transition. For example, the sentence "The merchant needs to verify whether the payment amount matches the order price." comes right after "The syndicator will send a Webhook request to the merchant server. " in Beecloud [6]. So for each potential SR-related sentence, DoA tries to locate a transition-related sentence in the same paragraph, before the SR sentence. Once the transition is found, we further check whether its destination is a *merchant* state, so the merchant is supposed to perform the security inspection mentioned in the sentence. In this case, we set SR_{state} to that merchant state.

The only other place where security check description can be found is the specification of the API used for the state transition. For example, under the API "Transaction-result-notification" (iAppPay [9]), there is a note that reminds the developer to inspect payment: "Please verify the transaction

payment is the same as the product price". So when the SR-related sentence is discovered in such a specification, we look at the state the API leads to, and make it SR_{state} if it is a merchant state.

3.5 Logic-flaw Prediction and Validation

From the FSM and the SR information discovered from the syndication documentation, LfP infers the possible presence of logic flaws: the SRs expected by the payment processor (e.g., Alipay) that cannot be fulfilled in the syndication FSM, and those that have not been explained to the developer. Following we explicate how to determine the states expected to perform the required security checks, how to evaluate whether the checks can take place, and how to capture the SRs that have not been properly communicated to the developer.

Security goals. Consider a syndicator W that wraps a payment service P. We believe that W needs to achieve the following two security goals:

• *Secure Design (SD):* for any security requirement SR to be enforced at a state s of P, there exists an *enforceable* SR' at the state s' of W such that SR and SR' are equivalent except their states, and s' is a state in the extension structures for s (Figure 6). Intuitively, this means that every payment SR should still be fulfilled after syndication.

• *Secure Implementation (SI):* every SR of W is correctly implemented by either the merchant or the syndicator.

To achieve SD, for every SR of the state s in the payment FSM, LfP first identifies all extended states of s and then inspects each of them s' to determine whether the state has the *visibility* of the object (SR_{obj}) and other parameters (SR_{para}) of SR. As mentioned earlier, s' is identified by the extension structures (Figure 6). Any state among these replacing s is considered to be a possible location for enforcing SR. As an example, consider *b.3*, which is an extension of b, and $SR = (m_f, payment, price)$, which checks $payment = price$. This inspection can happen at either m_f or w, when at least one of them can observe both *payment* from the payment service (e.g., Alipay) and *price* from the merchant.

For SI, without looking at the code of the extended FSM, LfP goes through the developer's guide for the indicators that could lead to implementation flaws. The most important one used in our current design is the absence of the explanations about SRs, a clear signal that the related security checks might not be implemented by the uninformed developer. Also we are concerned about the SRs that can only be enforced by the syndication state, since they are out of the merchant's control. So in both cases, LfP will predict potential logic flaws and suggests a code-level validation.

Design flaws identification. When none of the states in the extension structure of a payment state can observe SR_{obj} and SR_{para} for a security requirement SR, we can conclude that the syndication service contains a design flaw. To detect such a flaw, LfP needs to analyze visibility of data at each extension state. As mentioned earlier, such data either comes from the message a given state receives, whose content is described by all the attributes of the API used for transmitting the message (Section 3.4), or preconfigured by the merchant in her syndication account, with all data attributes (APP ID, public key, seller account, etc.) discovered by DoA. Our approach directly compares these attributes with the SR information. Note that *only the data delivered through a secure channel and from a trusted source can be used in a security check*: for example, the payment amount should be signed by the payment provider and the price should come from the merchant (through preconfiguration, signed message or local storage).

Specifically, at a given extension state, let TA be a set of *trusted* message attributes (e.g., signed by the payment provider) as collected from related API specifications, and when the state is controlled by the syndicator, TC be a set of collected attributes for preconfigured merchant information. Also, we abuse notations a little bit, using SR_{obj} and SR_{para} to represent the *sets* for the object and for the additional parameters, respectively, of a given security requirement SR at that state. The objective of LfP is to find out whether there exists an extension state such that for a given SR on the corresponding payment state, $SR_{obj} \cup SR_{para} \subseteq TA \cup TC$ for the TA and TC of the extension state (TC includes all the local data for a merchant state). If none of such an extension state can be found, LfP reports that the SR can no longer be enforced and therefore a design flaw is detected.

For a data attribute (e.g., "price", "public_key", etc.) $a \in SR_{obj} \cup SR_{para}$, it is *nontrivial* to determine whether it is also in $TA \cup TC$, simply because the attribute names of the SR collected from a payment service (e.g., Alipay) may not match those included in the message API and the configuration web page. Our solution here, again, is using our word-embedding model to product a semantic vector for a and then find whether there is an attribute in $TA \cup TC$ whose vector is sufficiently close to that of a. When every attribute of the SR can find its counterpart in TA or TC, we consider that the SR is enforceable at the current extension state.

Implementation flaws prediction. As mentioned earlier, implementation flaws can also be predicted when a required security check has not been properly communicated to the developer. LfP is designed to inspect the SR descriptions recovered by DoA to identify such missing security guidance.

Specifically, for each SR enforced at the state s in the payment FSM, LfP searches across all security requirements discovered from the guide by DoA for those associated with the extension states of s. Let SR' be such a requirement. Our approach tries to determine whether $SR_{obj} = SR'_{obj}$ and $SR_{para} = SR'_{para}$. Again, here we need to deal with the inconsistency in attribute names during the comparisons, which has been addressed in our implementation using our word-embedding model and distance measurement between semantic vectors. If SR' here indeed matches SR, we have reason to believe that the developer knows the security check required

by the payment provider. If such SR' cannot be found, then we know that such information has not been conveyed to the developer. When the SR can only be enforced by the merchant, LfP will raise the alarm since we doubt that an uninformed developer can make the protection right.

Even when all the SRs are found to be enforceable on the syndication FSM and all merchant-side SRs are properly mentioned in the developer's guide, we are still concerned about the security checks that can only be performed by the syndicator, who does not mention that this has actually been implemented. Given the fact that the merchant essentially loses the control of these security checks that they could do without the syndication, we believe that these SRs should be evaluated to ensure that they have been put in place. So our current implementation of LfP also reports all such SRs, which are evaluated during the validation step.

Validation. Fully automated verification of our predicted flaws is possible but nontrivial, due to the requirements of entering user credentials (password, fingerprint) to trigger a payment process and handling diverse user interfaces in different mobile apps for entering purchase information (product, quantity, address, etc.). Although existing GUI testing tools could be enhanced to serve this purpose and likely industry-grade fuzzers can already support these operations, building such techniques are outside the scope of our research. So we manually validated all the flaws predicted by Dilution. Specifically, based upon the specific security requirement that we consider hard to enforce, we acted as a malicious buyer to adjust the payment parameters to find out whether the predicted flaw can indeed be exploited. As an example, consider the payment process in Figure 5(b). Dilution predicts an implementation flaw that the merchant does not check the payment amount. In our research, we set a lower price in the *orderInfo* given to an app. This transaction got through (Section 5.2), which confirmed the presence of the flaw we predicted.

4 Implementation and Evaluation

4.1 Implementation

Dilution is implemented in a prototype. In the DoA, we employ Language Technology Platform (LTP) [19], for word segmenting, POS tagging and dependency parsing to analyze the sentences. To adopt the open domain toolkit to payment document analysis, we craft external dictionaries containing 48 domain-specific terms (e.g., payment element and payment credential) in the payment process to improve the performance of word segmenting and embedding. Taking the results of dependency parsing as features we further implement the classifier for entity recognition with LIBSVM [30] in version 3.23. To map words into vectors we utilize the word2vec model in Gensim library [18] in 3.7.1 version. Moreover, we ran the crawler *Scrapy* [44] in version 1.6 to crawl all the web pages of syndications' official websites, and then utilize the *BeautifulSoup* [42] in version 4.4.0 to parse webpages and extract the developer's guides and configuration. For the LfP, we implement with 404 lines of Python code for interring the supposed SRs and inspecting each of them to predict flaws. We are going to release the source code of Dilution online [14].

4.2 Experiment Settings

Dataset. In our research, we utilized four datasets for model training and evaluation:

• *Groundtruth set*. The groundtruth set was used for logic flaw detection, entity recognition and phrase alignment.

For logic flaw detection, the groundtruth set includes two syndication documents (Ping++ [12] and Fuqianla [8]) and their corresponding 17 potential logic flaws. In particular, we manually analyzed the documents and identified 11 implementation flaws in the documentations of syndication Ping++, 1 design flaw and 5 implementation flaws in the documentation of syndication Fuqianla, as elaborated in Section 5.

The groundtruth set for entity recognition in the payment process consists of 574 entities (148 *Sender*, 175 *Receiver* and 251 *Content*) from 242 sentences describing data transmission. These sentences were collected from a training corpus including documents of two payment services Alipay and Wechat and two syndicators Ping++ and Fuqianla. We implemented a 2-fold cross validation with half of the data as the training set each time.

We manually labelled the groundtruth set for phrase alignment upon the syndication documents of Ping++ and Fuqianla, which contain 14 data-transmission sentences, 103 APIs and 1,986 parameters in total. The groundtruth set includes 63 and 203 positive pairs, and 45 and 40,866 negative pairs in transition mapping for extension discovery and API parameter discovery, respectively.

• *Unknown syndication documents*. To evaluate *Dilution*, we ran our prototype on the developer's guides of six syndications, including Paymax [11], BeeCloud [6], iAppPay [9], Trpay [15], UMF Payment [16] and 66zhifu [1]. These documents consist of 3,613 sentences and 46,098 words in total. They are all publicly available, well-written documentations including the description of data transmission in payment processes, API parameter explanations, SRs and configuration information. Note that those services are popular, serving tens of thousands apps and millions users.

• *Third-party payment documents*. The third-party payment documents we manually analyzed to extract FSMs and SRs come from Alipay, WeChat Pay and PayPal, which are the three most popular mobile payment services in the world [45]. We read all the related documentations about the payment process and searched keywords related to SRs including "security", "requirement", "check", "inspect", "compare" to collect all the SRs. Three experts spent 2 days to finish the extraction and the SRs are validated across all three experts. The detailed SRs are summarized in Table 2.

Table 2: Security requirements of payment service

$notify_{id}$: The id of a notification, $seller_{id}$: The id of a seller, txn_{id}: The id of a transaction,
$receiver_{email}$: The account of the merchant, mc_{gross}: The amount of a payment, $mc_{currency}$: The currency of a payment.

Payment Service	No.	SR description	SR
Alipay	SR1	Check the signature in the notification.	$(m2, notification, key)$
	SR2	Check the $notify_{id}$ to verify the message comes from Alipay.	$(m2, notify_{id}, \emptyset)$
	SR3	Check the price in the notification is the same with the amount in the order.	$(m2, payment, price)$
	SR4	Check the $seller_{id}$ represents the supposed merchant.	$(m2, seller_{id}, merchant)$
WeChat Pay	SR5	Verify the signature in the payment notification message.	$(m2, notification, key)$
	SR6	Check the price in the notification equals the price in the order.	$(m2, payment, price)$
PayPal	SR7	Verify the message came from PayPal.	$(m2, message, \emptyset)$
	SR8	Check the txn_{id} against the previous PayPal transaction that you processed to ensure the IPN message it not duplicate.	$(m2, txn_{id}, previous\ txn_{id})$
	SR9	Check that the $receiver_{email}$ is the email address registered in your account.	$(m2, receiver_{email}, registered\ email)$
	SR10	Check that the price carried in mc_{gross} are correct for the item.	$(m2, payment, price)$
	SR11	Check that the currency carried in $mc_{currency}$ are correct for the item.	$(m2, receipt\ currency, supposed\ currency)$

• *Payment corpora for word embedding model training*. For training the word embedding model, we built a corpus for payment service by combining two payment documentations (Alipay, Wechat) and two syndication documentations (Ping++, Fuqianla), which were crawled from the corresponding websites. After word segmenting, the training corpus contains 1715.2 KB text with 23,576 sentences and 306,680 words.

Parameters. The parameters for our implementation are set as follows:

• *Entity classifier*. We implemented the classifier for entity recognition with LIBSVM [30]. The classifier was trained with the following settings: c=8.0, g=0.5 and default settings.

• *Word2vec*. We utilized skip-gram with negative sampling as the framework of the word2vec model, which was trained with the following parameters: sg=1, size=100, sample=0.0001, window=10, iter=5, min_count=1, negative=20 and other default settings.

• *Threshold*. We utilized phrase similarity to find out whether two phrases are semantically close. For payment-related expressions (e.g.,payment_element, payment_credential), the threshold used in transition mapping, API name matching and parameter matching were set to 0.91, 0.91, and 0.97 respectively. As for other phrases, the threshold in three tasks were set to 0.87, 0.87, and 0.96, respectively.

Platform. All the experiments in our study were conducted on the macOS with 2.3GHz CPU, 16GB memory and 512GB hard drivers using a single process.

4.3 Effectiveness

We first evaluated the overall effectiveness of our prototype in predicting potential logic flaws from documentations. Running on both the groundtruth set (Ping++ and Fuqinala) and the unknown dataset containing six syndicators, Dilution achieved 100% accurate predictions. More specifically, on the six unknown documentations, our system predicted 1 design flaw and 16 potential implementation flaws. We manually verified each of them and found that all reports were correct (based upon the descriptions in the documentations).

Further, we evaluated the two internal modules of DoA: entity recognition and API parameter discovery. The evaluation for entity recognition was run on the groundtruth set for entity recognition (242 data-transmission related sentences with 574 entities including 148 *Sender*, 175 *Receiver* and 251 *Content*). Under the two-fold cross validation, our model achieved a precision of 89.38%, 93.28%, 94.57% and a recall of 96.88%, 97.72%, 96.81% when taking *Sender*, *Receiver*, *Content* as the positive class, respectively. The effectiveness of our model is acceptable since our algorithm for discovering the state extension is capable of addressing the false positives and false negatives induced by entity recognition through alignment with the transitions in the extension structures (Section 3.3). As for the API parameter discovery, the experiment results on the guides of all eight syndications show that all phrase pairs aligned by DoA are accurate.

4.4 Performance

We ran Dilution on the developer's guide of 8 syndications (1,456KB) to predict the presence of logic flaws. Averagely, our system spent merely 3,177.8 ms to go through the whole process on one syndication. The time of the analysis ranges from 2,743.2ms to 3,873.8 ms, with the medium being 3,099.1 ms. More specifically, DoA spent 2,738.8 ms to 3,840.0 ms with an average of 3,166.2 ms. LfP took 2.9 ms to 33.8 ms with an average of 11.6 ms. This result offers strong evidence that Dilution can easily scale to the level expected for processing a large amount of documentation.

5 Discoveries in the Wild

In this section, we report the logic flaws predicted by Dilution from the developer's guides of real-world syndication services and the end-to-end exploits to validate the predictions through popular merchant apps. We show that our document-only predictions are indeed accurate, leading to the discovery of security-critical vulnerabilities.

5.1 Finding from Documentations

There are more than 30 syndication services, with the number continuing to grow. However, most of them provide developer's guides to paid users only. Actually we found that just

Table 3: Summary of predictions by Dilution

(DF: design flow, IF: implementation flow, CI: cases of interest)

(a) design & implementation

Syndication	Type	SR No.
Fuqianla	DF	1
	IF	3, 6
BeeCloud	DF	1
	IF	11
TrPay	IF	3, 6
UMF Pay	IF	3, 6
66zhifu	IF	3, 6
Total	DF	**2**
	IF	**9**

(b) cases of interest

Syndication	Type	SR No.
Ping++	CI	1 - 11
Fuqianla	CI	2, 4, 5
Paymax	CI	1 - 6
BeeCloud	CI	2, 4, 5, 7, 8, 9
iAppPay	CI	2
TrPay	CI	2
UMF Pay	CI	2
66zhifu	CI	2
Total	CI	**30**

8 of them have well-documented guides publicly available. These syndicators are all popular, with hundreds of millions of users. In our research, we ran Dilution on all of their guides (over 1.4 MB), which reported its findings in a few seconds.

Landscape. Table 3 shows all the syndications we analyzed and the logic flaws predicted. Specifically, Dilution reported 41 potential issues from all the syndications. Among them, 11 are highly likely to be logic flaws, including 2 design flaws (in BeeCloud and Fuqianla), in which required security checks cannot be done, and 9 likely implementation flaws, with critical security checks missed in the guides. In addition, the remaining 30 are "cases of interest", since their SRs can only be or should be fulfilled by syndicators if merchants cannot achieve them or do not be told. Therefore they are considered to be risky and need to be validated.

Design flaws. Among all the syndication services, BeeCloud and Fuqianla are found to contain a design flaw each (SR1 in Table 2). Specifically, Dilution reported that these syndicators receive payment notifications from Alipay on behalf of their merchants, helping them finish the final security checks before informing them of the completion of the transactions. The problem is that the merchants of these services cannot configure their Alipay's verification keys[4] to the syndicators. As a result, the syndicators cannot check the authenticity of the messages, nor can their merchants, since they do not get the signed notifications. We further found that the practice of processing payment messages for the merchants without forwarding them the original messages is very common across all syndications we studied. This is because the syndication aims to unify the merchant-side interfaces with different payment services to reduce the complexity in integrating them, which, however, makes the syndicator-side operations complicated and error-prone. Although only two design flaws were revealed by Dilution, due to the small number of syndicators we evaluated, we believe that the practice likely brings in design lapses to other syndication services.

Implementation flaws. Dilution predicted 9 potential implementation flaws by the merchant developers in 5 syndication services. Specifically, our approach found that the syndicators Fuqianla, TrPay, UMP Pay and 66zhifu cannot verify the payment amount since they do not have access to the price of a purchase.In the meantime, they fail to remind their merchant developers of enforcing the security requirements (SR3 and SR6) through verifying the amount. Similarly, BeeCloud encapsulates PayPal but does not tell its merchant that the currency type for a purchased item needs to be checked. As a result, we believe that very likely the required security checks will fall through the cracks.

[4]Each merchant has a unique key-pair for verifying the messages from Alipay.

5.2 Attacks on Real-World Systems

Challenges in validating predicted flaws. To find out whether the predicted logic flaws are indeed present in syndicators or the merchant-side code, we need to validate them through merchant apps. This attempt, however, faces two challenges. First, finding the apps integrating a given syndication is difficult. Even though these services are popular (e.g., at least 25K apps using Ping++), with tens of millions of users according to their websites, rarely do they provide a list of the merchants that use their services. Actually, most syndicators ask their merchant developers to obfuscate their code, possibly for the purpose of IP protection [29]. Second, even given an app integrating a syndication, exploiting its logic flaws may need additional resources we do not have and some of the flaws may not even be exploitable in the absence of other flaws. Particularly, in 7 out of the 30 cases of interest reported by Dilution, we need to produce Alipay's signature on the payment notification to confirm whether the syndicator (e.g., Paymax) indeed fails to perform a security check (e.g., on the payment amount), since the syndicator may still verify Alipay's signature. The exploit can only be executed with the help of a merchant under our control: we can make a purchase from our own merchant and use the notification to determine whether the syndicatior indeed verifies its attributes (e.g., payment amount). We manually analyzed our findings and believe that if the predicted flaws are there, we can exploit them in this way. However, merchant registration (with Alipay) is complicated, which we did not do in our research.

Despite the challenges, still we were able to find 17 apps to confirm 5 logic flaws across 2 syndication services and their merchants. These 17 apps were found from over 50K apps we analyzed. Most importantly, *for every merchant app that could be analyzed, every single logic flaw or case of interest predicted by Dilution has been confirmed*. Specifically, we randomly crawled over 50K apps from the Baidu Market, a top Chinese app market [5], and ran Apktool [3] to reserve-engineer them. The 17 were found because the names of their syndication SDKs or the domains of syndication servers have not been obfuscated. Among them, 16 use BeeCloud [6] and 1 uses TrPay [15]. Both the syndication services and the apps are very popular. As shown in Table 4 in Appendix, BeeCloud claims to have tens of thousands merchants, and these apps have hundreds of millions of users. In our experiment, we ran a proxy called Burp Suite [7] and a network API testing tool called Postman [13] to modify or forge the messages

delivered from the apps or our site to the merchant or the syndicator server.

Attacks on design flaws. As mentioned earlier (Section 5.1), Dilution reported two design flaws, one for BeeCloud and the other for Fuqianla, in which the syndicator cannot verify Alipay's signature on a payment notification due to its lack of the verification key. In our research, we could not find the app using Fuqianla possibly due to the obfuscation it suggests to its users [23]. The 16 apps using BeeCloud, however, were predicted to all have the *same* logic flaw. Therefore, we just randomly picked one of them, a Chinese education app called Chuangyebang [21], for the validation. Specifically, we placed an order for an online class provided by the app without payment, then used Postman to forge an unsigned Alipay's notification and delivered to the BeeCloud server. As predicted, the syndicator accepted the fake message and we successfully got the digital product for free [5]. This demonstrates that the design flaw is real and exploitable, which has a serious consequence given the fact that BeeCloud is serving tens of thousands of merchants [28]. Actually, the 16 apps we collected have 82.8 million downloads in total, selling products ranging from 0.1 dollars to 2,000 dollars. Even Changyebang is reported 490,000 installs.

Attacks on implementation flaws. Also we were able to find a Chinese tool app called OffPhone [24] (with 830,000 downloads) that integrates TrPay, another popular syndication service [15] wrapping WeChat and Alipay payment services. This enabled us to validate 2 potential implementation flaws that Dilution predicted. Both problems happen at the final state m_f (one for WeChat and the other for Alipay), where the merchant is supposed to check the payment amount, which the developer's guide of TrPay fails to mention. Since in both cases, the syndicator generates the credential (including the price of the order) for payment at WeChat or Alipay, based upon the order issued through the app, the adversary can control the app to provide wrong price information to mislead TrPay into producing a credential with a lower price. If the payment amount reported by WeChat Pay or Alipay has not been verified by the merchant (OffPhone) at m_f, the adversary can get the item with the price she set. In our experiment, we modified the order placed through the OffPhone app to reduce the price of a VIP membership from 5 dollars to just 1 cent, causing TrPay to send a credential to the payment services. In the end, both transactions went through, which indicates that indeed the payment amount has *not* been checked by the merchant, as Dilution predicted.

Furthermore, Dilution also reported 2 "cases of interest" in BeeCloud, where the syndicator is supposed to perform some security checks on behalf of the merchant. Specifically, we found that on receiving the payment notification from Alipay, BeeCloud only passes some of the notification content to the merchant. An attribute not being forwarded is *notify_id*, the merchant's identity issued by Alipay for finding out whether the merchant is the right recipient of the notification. Also, although another attribute, *seller_id*, serving the same purpose is indeed sent to the merchant, BeeCloud fails to mention in its developer's guide that the attribute should be verified. Both attributes are *required* to be inspected by Alipay [2]. Here, we tried to find out whether the checks have been done by BeeCloud on the merchant's behalf. For this purpose, we randomly selected another app from the 16 apps, called Clean [22], a memory cleaner with 850,000 downloads, to find out whether the SR2 and SR4 in Table 2 have been enforced. In the experiment, we ran Postman to fake a payment notification including a random *notify_id* and *seller_id*, which did not prevent our payment transaction from getting through. In the end, we got a paid version of the app (without Ads) for free. The same attack also succeeded on Chuangyebang. Evidently, not only does BeeCloud fail to verify *notify_id* and *seller_id*, but Clean (the merchant) does not check *seller_id* either, in line with the predictions made by Dilution. These flaws open the door for an attack in which one pays for her own merchant while getting product from a different store [49]. Also note that since the merchant is not given *notify_id*, remaining 15 apps and others using BeeCloud are certain to have the same flaw.

Responsible experiment design. We carefully designed our end-to-end attack in a responsible manner. The entire study was conducted under the guidance of a lawyer at our University. We strictly followed the principles below when attacking the real-world apps and services: (1) we performed no intrusion of either merchant servers and syndicator servers; (2) we ensured that no financial damage caused by returning items, paying the shipping costs, not getting refunded, not using electronic products, paying for items hard to return; (3) we reported all security flaws to affected apps and syndications and did what we could to help them improve their systems. All the flaws discovered have been acknowledged by the syndicators and merchants, who are very grateful for our help.

6 Discussion

Limitations. Our research demonstrates that logic flaws can be predicted from the developer's documentation, even before the system code is inspected. This finding can lead to new techniques that make full use of information available for enhancing software security. Our current design and implementation, however, are still preliminary. We only focus on the payment syndication service with limited targets on missing security checks. More complicated flaws, such as policy enforcement weaknesses (possibly caused by inaccurate guidance), and more complicated service procedures (e.g., refund, bonus, etc.), are all missing from the picture, not to mention documentation-based analysis on other security-critical systems. Further, the NLP techniques underlying Dilution can only deal with well-written documents, not those containing

[5] All the exploit video demos are post online [4].

typos, grammatical errors, ambiguous sentence structures, etc. as observed in real-world developer's guides. Also, our current approach still needs human involvements: particularly, we extracted SRs and FSMs of payment processors manually, which has two reasons. First, the number of payment processors is small. Second, we want the extracted SRs and FSMs to be precise, based on which we built syndicators' FSMs and predicted logic flaws. However, we envision it's possible to automate this process using open-source NER tools. All these issues need to be addressed in the future research.

Future research. Down the road, we expect more explorations on this new direction, toward the end of an intelligent, semantics-based methodology that combine documentation-level and code-level analysis together for more effective flaw discovery. In addition to the direct improvement of our approach mentioned above, we envision that document-based flaw prediction will be applied to secure other syndication services, those for single sign-on services in particular, such as MobSDK [10]. More importantly, we believe that with the help of machine learning and automatic inference, other subtle, semantics-dependent weaknesses only detectable by experienced analysts today will become increasingly manageable by automatic techniques, leading to a significant improvement in software security quality.

7 Related Work

Numerous studies have looked into logic flaw detection in various applications. For example, [49] discovered logic flaws of the payment service. [48, 50] investigated logic flaws on authorization. Traditional logic flaws discovery heavily relies on domain experts [33]. Recent year witnesses the trend of automatic logic flaw detection, mainly based on model checking. The typical approach based on model check first standardizes a logic process, and then detects whether the application violates the predefined logic. For instance, both [41] and [32] automatically extracted a model from a number of correct behavioral patterns. Then, they checked the source code statically, using model checking over symbolic input to identify violated program paths. [34] manually summarized the correct usage of OpenSSL APIs and then statically analyzed whether an application violates the correct usage. Different from previous researches, our approach does not touch program code and automatically utilizes only documentation to predict logic flaws.

The closest to our study are the works of payment logic flaw assessment [49] [51]. [49] is the first work, which relies on human effort, to discover serious payment logic vulnerabilities and reveal their security implications. A set of follow-up studies identified different types of payment logic flaws in various applications. For instance, [51] extended the work of [49] to investigate the logic flaws in mobile payment and detected seven security rule violations to the payment in Android apps. [46] detected violations of the invariant in secure checkout processes, which revealed 11 new logic vulnerabilities in web payment modules. Given these studies, a bunch of policies and guidelines for secure online shopping were investigated [38, 40]. In contrast to previous works, which assessed logic flaws by manual efforts, we report the first work towards automatic payment logic flaw discovery. Also, we investigate a novel payment service, payment syndication, which has never been studied before.

8 Conclusion

In this paper, we report the first step towards automatic documentation-based logic flaw discovery. Our study on the emerging payment syndication services shows that their developer's guide contains abundant information that can be leveraged to predict the presence of logic flaws in their customers' systems. Using a suite of NLP techniques, our approach effectively analyzed over 1.4 MB of technical documentations from real-world syndicators within seconds, and accurately predicted 5 new security-critical flaws in the Chinese merchant systems with millions of users. The research demonstrates that software documentations can be more effectively used to help find the security risks hard to automatically detect today.

9 Acknowledgment

We would like to thank our shepherd Adam Doupé and the anonymous reviewers for their insightful comments. The IU authors are supported in part by NSF-1527141, 1618493, 1838083, 1801432 and 1850725, ARO W911NF-16-1-0127 and Indiana University FRSP-SF. IIE authors are supported in part by NSFC U1836211, U1836209, 61728209, 61602470, 61802394, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, Beijing Natural Science Foundation (No.JQ18011), National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200), National Key Research and Development Program of China (2016QY071405) and the Program of Beijing Municipal Science & Technology Commission (NO. D181100000618004).

References

[1] 66zhifu. https://www.66zhifu.com.

[2] Alipay. https://www.alipay.com.

[3] apktool. https://ibotpeaches.github.io/Apktool/.

[4] Attack demos. https://sites.google.com/view/dilution/home/attack-demos.

[5] Baidu mobile assistant. https://shouji.baidu.com.

[6] Beecloud. https://beecloud.cn.

[7] Burp suite. https://portswigger.net/burp.

[8] Fuqianla. https://fuqianla.net.

[9] iapppay. https://www.iapppay.com.

[10] Mobsdk. http://www.mob.com.

[11] Paymax. https://paymax.cc.

[12] Ping++. https://www.pingxx.com.

[13] Postman. https://www.getpostman.com.

[14] Source code of Dilution:. https://github.com/ccy1991911/Dilution.

[15] Trpay. http://pay.trsoft.xin/front/index.html.

[16] UMF pay. https://xy.umfintech.com.

[17] WeChat pay. https://pay.weixin.qq.com.

[18] Gensim. https://github.com/rare-technologies/gensim, 2018.

[19] LTP. https://github.com/HIT-SCIR/pyltp, 2018.

[20] 66zhifu obfuscation guide. https://www.66zhifu.com/show/help, 2019.

[21] Chuangyebang download link. http://m.cyzone.cn/app/, 2019.

[22] Clean download link. https://shouji.baidu.com/software/25240151.html, 2019.

[23] Fuqianla obfuscation guide. https://fuqianla.net/docs.html?Android_SDK, 2019.

[24] Offphone download link. http://offphone.net, 2019.

[25] Paypal. https://www.paypal.com, 2019.

[26] Trpay obfuscation guide. http://pay.trsoft.xin/front/documentation.html, 2019.

[27] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[28] BeeCloud. Beecloud news. https://beecloud.cn/about/#honor, 2019.

[29] Chandan Kumar Behera and D Lalitha Bhaskari. Different obfuscation techniques for code protection. *Procedia Computer Science*, 70:757–763, 2015.

[30] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2011.

[31] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014.

[32] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[33] OWASP Testing Guide. Testing for business logic. https://www.owasp.org/index.php/Testing_for_business_logic/, 2019.

[34] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 519–534. IEEE, 2015.

[35] Prospective Industry Research Institute. China's syndication payment industry market prospects and investment strategic planning analysis report for 2018-2023. https://bg.qianzhan.com/report/detail/1703301644253052.html, 2018.

[36] iyiou. China syndication payment industry development report in 2018. https://www.iyiou.com/p/88682.html, 2018.12.28.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *neural information processing systems*, pages 3111–3119, 2013.

[38] Chandan Kumar Giri MimansaGantayat. Security issues, challenges and solutions for e-commerce applications over web.

[39] Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada. *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[40] M Niranjanamurthy and DR Dharmendra Chahar. The study of e-commerce security issues and solutions. *International Journal of Advanced Research in Computer and Communication Engineering*, 2(7), 2013.

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

[42] Leonard Richardson. Beautiful soup. https://www.crummy.com/software/BeautifulSoup/, 2019.

[43] David Sacks. System and method for third-party payment processing, February 7 2002. US Patent App. 09/901,962.

[44] Scrapinghub. Scrapy. https://scrapy.org, 2019.

[45] Statista. Number of users of leading mobile payment platforms worldwide as of august 2017. https://www.statista.com/statistics/744944/mobile-payment-platforms-users/, 2017.

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

[47] TechNode. Briefing: Alipay now has over 1 billion users worldwide. https://technode.com/2019/01/10/alipay-1-billion-users/, 2019.

[48] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 365–379. IEEE, 2012.

[49] 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 *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 465–480. IEEE, 2011.

[50] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43. ACM, 2015.

[51] 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 *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2017.

APPENDIX

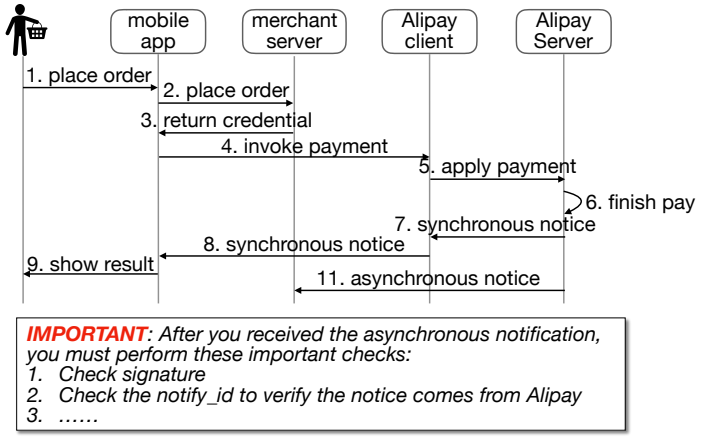


Figure 11: An excerpt of the Alipay diagram and security requirements.

Table 4: Collected apps

App	Package name	Category	Syndication	Downloads
Max+	com.dotamax.app	Game	BeeCloud	1,300,000
Zhihuiwuxi	com.hoge.android.wuxiwireless	Business	BeeCloud	1,160,000
Zhongyizhiku	com.zk120.aportal	Life	BeeCloud	1,100,000
Clean	com.ktls.fileinfo	Tool	BeeCloud	850,000
Yuebanchuxing	com.ynwl.yueban	Social	BeeCloud	770,000
Jiaoshisuishixue	cn.ixunke.suishixue	Education	BeeCloud	660,000
Chuangyebang	com.cyzone.news	Education	BeeCloud	490,000
Yikeweiqi	com.indeed.golinks	Game	BeeCloud	360,000
Hediandian	com.hoge.android.app.hdd	Business	BeeCloud	350,000
Zhihuiyancheng	com.hoge.android.yancwireless	Business	BeeCloud	170,000
Huiyouhui	com.huiuhuisc.zoj	Business	BeeCloud	90,000
Wuxianhuaian	com.hoge.android.huaian	Business	BeeCloud	70,000
Zhongyiguji	com.zk120.ji	Health	BeeCloud	50,000
Zhongyiyian	com.zk120.an	Health	BeeCloud	10,000
Quandashi	com.dream.ipm	Business	BeeCloud	10,000
Shuohua	com.etang.talkart	Art	BeeCloud	10,000
OffPhone	com.alion.silent	Tool	TrPay	830,000

Table 5: Abbreviation summary in alphabetical order.

Abbreviation	Denote
ADV	Adverbial
ATT	Attribute
b,b_i	Buyer state (No.i)
b_1	The initial state
B	Buyer
COO	Coordinate
d	A message transmitted among a buyer, a merchant and a syndicator
D	A set of messages
DoA	Documentation Analyzer
E	A function that drives the transition from one payment state to the next
FSM	Finite state machine
LfP	Logic-flaw Predictor
m, m_i	Merchant state (No.i)
m_f	The final state
M	Merchant
p, p_i	Payment processor state (No.i)
P	Payment processor
POB	Preposition-object
s	A state in an FSM
S	A set of payment states
SBV	Subject-verb
SD	Secure design goal
SI	Secure implementation goal
SR	Security Requirement
SRi	The No.i Security Requirement in Table 2
SR_{obj}	The object to check for an SR
SR_{para}	The parameters for an SR when checking
SR_{state}	An SR's corresponding state
TA	A set of trusted message attributes as collected from related API specifications
TC	A set of collected attributes for preconfigured merchant information
VOB	Verb-object
w, w_i	Syndicator state (No.i)
W	Syndicator