IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 14, 2020, accepted January 29, 2020, date of publication February 4, 2020, date of current version February 12, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2971615

Analyzing Security Protocol Web Implementations Based on Model Extraction With Applied PI Calculus

XUDONG HE[1], QIN LIU[1], SHUANG CHEN[1], CHIN-TSER HUANG[2], DEJUN WANG[1], AND BO MENG[1]

[1]School of Computer Science, South-Central University for Nationalities, Wuhan 430074, China
[2]Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208, USA

Corresponding author: Bo Meng (mengscuec@gmail.com)

This work was supported in part by the Fundamental Research Funds for the Central Universities, South-Central University for Nationalities, under Grant CZT20013 and Grant QSZ17007, and in part by the Natural Science Foundation of Hubei Province under Grant 2018ADC150.

ABSTRACT Analyzing security protocol web implementations is a crucial part of web security. Based on the model extraction technology, this paper first defines SubJavaScript and SubPython languages, and then establishes mapping models from SubPython and SubJavaScript to Applied PI Calculus respectively, after that, develops the semi-automatic model extraction tools SubPython2PV and SubJavaScript2PV to analyze the four widely used security protocol web implementations. The experiment shows that the four typical security protocol web implications have confidentiality, but lack of authentication.

INDEX TERMS Security protocol implementations, model extraction, SubJavaScript, SubPython, formal method, ProVerif.

I. INTRODUCTION

In recent years, Python and Javascript are widely used in the security protocol Python web implementations [1], [2]. Therefore, it is significant to analyze security protocol Python web implementations to protect web security. The primary methods for analyzing the Security of Security Protocol Implementations (SSPI) are program verification methods [3]–[7] and model extraction methods [8]–[12]. The program verification methods mainly focus on logic proof and type-based methods. However, most of these methods not only overlook the verification correctness of the analysis process but also rely on adding a large number of comments and assertions in the Security Protocol Implementations (SPI). Goubault-Larrecq and Parrennes [5] and Jürjens [3] first proposed SSPI analysis methods for SPI written by C and SPI written by Java, respectively. Backes *et al.* [13] first performed the automated security analysis of the JavaScript.

Based on the symbolic model, Chaki and Datta [8] and Dupressoir *et al.* [14] analyzed the authentication and confidentiality of SPI written by Bengtson *et al.* [4], Bhargavan and Gordon [15], Backes [16], Swamy *et al.* [17]–[19] analyzed the authentication and confidentiality based on the F* type checker.

In general, the model extraction method first extracts the security protocol abstract specification from the SPI and then analyzes extracted security protocol abstract specification using formal methods. This method is effective and suitable for analyzing SPI because SPI is essentially a piece of code. Bhargavan *et al.* [20]–[22] extracted the abstract model of the SPI written by F* and analyzed its security. Mihhail *et al.* [9] and Aizatulin *et al.* [23] extracted the abstract model of the SPI written by C and evaluated the confidentiality using ProVerif [25], [26] and CryptoVerif [27] respectively. O'shea [28] and Li *et al.* [29] extracted the abstract model of the SPI written by Java and analyzed its security.

However, to our best knowledge, there is no related work in the literature on analyzing SPI whose Security Protocol Server Implementations (SPSI) written by Python and Security Protocol Client Implementations (SPCI) written by JavaScript. On one hand, Python is an interpretive language and it can't be encrypted as executive code. On the other hand, the function of JavaScript is transmitted from server

The associate editor coordinating the review of this manuscript and approving it for publication was Remigiusz Wisniewski.

to client via browser. Hence SPSI and SPCI are easy to be comprehended and analyzed. The model extraction method is suitable for analyzing the security of SPSI and SPCI. Therefore, in this paper, we use the model extraction technology to extract the Security Protocol Model Represented by Applied PI Calculus (SPMRAPC) from SPSI written by Python and SPCI written by JavaScript and then verifies its security through ProVerif. The main contributions of our works are described below:

(1) We analyze the core statements of the SPSI written by Python and SPCI written by JavaScript, and then define SubPython – a subset of the Python, and SubJavaScript – a subset of the JavaScript, and Backus-Naur Form of the SubPython (BNF[SubPython]) and BNF[SubJavaScript] respectively.
(2) We establish the Mapping Model from SubPython to Applied PI Calculus (MMSP2APC) and the Mapping Model from SubJavaScript to Applied PI Calculus (MMSJS2APC), which includes statements mapping and type mapping.
(3) We develop the semi-automatic model extraction tools SubPython2PV and SubJavaScript2PV based on MMSP2APC and MMSJS2APC respectively.
(4) We apply SubPython2PV, SubJavaScript2PV and ProVerif to analyze 51 Talk [30], mall management system [31], Yintou Securities [32] and Miku Platform [33]. The experimental results show that the four web projects have confidentiality, but lack of authentication.

The rest of the paper is organized as described below. Section II discusses the related works of SSPI Analysis. Section III introduces the Applied PI Calculus and BNF. Section IV defines the SubPython and presents the BNF of SubPython and establishes the MMSP2APC. Section V defines the SubJavaScript and presents the BNF of SubJavaScript and establishes the MMSJS2APC. Section VI develops the semi-automatic model extraction tools SubPython2PV and SubJavaScript2PV. Section VII applies SubPython2PV, SubJavaScript2PV, and ProVerif to analyze the confidentiality and authentication of the four web projects and compares it with the first automated analysis method for JavaScript. Section VIII compares our semi-automatic method with Backes' s method [13]. Section IX presents the conclusion and future works.

II. RELATED WORKS

Formal analysis methods for SSPI are mainly classified into two categories: model extraction methods [8]–[12] and program verification methods [3]–[7].

Using model extraction methods, people mainly study the implementations security such as F* [15], [20], C [9], JAVA [34], [35], Swift [36], and ProScript [37]. Bhargavan *et al.* [20] proposed a model extraction tool called fs2pv which extracts the SPI written by F*. They use fs2pv to analyze the Windows CardSpace protocol and TLS protocol and use ProVerif to verify its security. O'Shea [28] proposed the Elygah system which transforms the SPI written by Java into the Lysa Calculus process and obtains the formal model of the SPI and analyzes its authentication. However, this paper does not prove the correctness of the extraction methods. Aizatulin *et al.* [9] proposed an automatic solution to the SPI written by C. In the beginning, this solution obtains symbolic descriptions for the network messages. Then, it applies algebraic rewriting to obtain a process of Calculus description and applies ProVerif to prove security properties. Bai *et al.* [38] proposed AUTHSCAN, which extracts protocol specifications from the Web identity authentication system based on network traffics and JavaScript execution traces and generates the TML intermediate. TML captures the details of protocol implementations and translates it into a formal specification. Then, it converts the TML into Applied PI Calculus and verifies the SSPI by the verification tool. Bhargavan *et al.* [39] proposed the DJS2PV tool which converts encryption data and encoding library and defense library to the Applied PI Calculus and then uses ProVerif to analyze the security of the encryption protocol. Bhargavan *et al.* [24] also proposed methodologies for developing symbolic and computational models of TLS 1.3 and automatically analyzing its protocol core code by extracting the ProVerif model from its typed JavaScript code.

Based on the program verification, Goubault-Larrecq and Parrennes [5] built a module that links the SPI written by C with the abstract Dolev-Yao model using the program verification method. It produces the semantic Horn clause by the csur_cc compiler, and the H1 solver takes the Horn clause as input to verify the security. Bengtson *et al.* [4] proposed a Typechecker for verifying security properties of the source code based on the Z3 solver. Typechecker checks the authentication properties of cryptographic protocols by type-checking their source code.

Based on graph slicing, Backes *et al.* [13] first performed an automated security analysis of the real JavaScript implementation of the Helios voting client. First, the code transformation method refactors the implementations to deal with problems caused by the non-standard libraries by converting non-standard libraries into newer standards libraries for the browser to provide equivalent functions for most of jQuery's APIs. Then, The WALA tools create an intermediate representation that is converted into system dependency graphs. Finally, graph slicing is used to find all nodes in the graphs. Information flows analysis focuses on paths between high levels and low levels in the graph in order to reduce the large flows to a handful of potentially harmful flows.

III. APPLIED PI CALCULUS AND TYPICAL FUNCTIONS AND EQUATION THEORY

In this section, first, we introduce the Applied PI Calculus which is used to model security protocols. Second, we present the typical functions and equation theory.

A. APPLIED PI CALCULUS

Formal language Applied PI Calculus is an extended language of PI Calculus and is mainly used for the formal analysis of security protocols. It can easily convert a security protocol into the input of the Proverif [26]. Applied PI Calculus is composed of "Process", "Extended Process", "Query Statement", "Conditional Expression", and "Declaration" and models the communication processes and encrypted operation in the security protocol. The data types of Applied PI Calculus include the bitstring, bool, type bool, true or false. and the user-defined data type. In general, Plaintext, Ciphertext, Channel, Key, Signinput and Signoutput are the typical user-defined data types.

P, Q, R ::=	Process
0	Empty Process
P | Q	Concurrent Process
!P	Replication process
vn.P	Restricted name
if M = N then P else Q	Condition
u(x).P	Message input
$\bar{\text{u}}$(N).P	Message output

FIGURE 1. **The process in the Applied PI Calculus.**

Fig. 1 shows the Process in Applied PI Calculus. Each of P, Q, and R is a process. The empty process "0" does not execute any operation. Concurrent Processes "P|Q" executes "P" and "Q" at the same time. Replication process "!P" executes multiple processes "P" concurrently. The restricted name "vn.P" first generates a new name "n" and then executes the operation of the process "P". Condition process "if M = N then P else Q" identifies whether the condition "M = N" is true or not. If the condition is true, executes process "P", otherwise executes process "Q". Message input process "u(x).P" receives messages from channel "u" and use channel "u" to present the message, and then executes process "P". The messages output process $\bar{\text{u}}$(N).P outputs the message "N" from channel "u" and executes process "P".

Compared to the Process in the Applied PI Calculus, the extended process in the Applied PI Calculus has an active substitution. Fig. 2 shows the extended process in the Applied PI Calculus. It uses variables to represent multiple types of data, such as channels and keys. It uses functions to represent encryption operations, signature operations, and decryption operations. It also uses "free" to represent the channel of data transmission and set the private channel using the keyword "private".

A,B,C::=	Extended process
P	Ordinary process
A | B	Parallel composite
vn.A	Limited name
vx.A	Restrict Variable
{ M/x }	Active replacement

FIGURE 2. **The extended process in the Applied PI Calculus.**

```
process::=
    0
    | <ident> | (<process>)
    | <process> | <process>
    | !<process>
    | if <fact> then <process> [else <process>]
    | in(<term> , <term>) [;<process>]
    | out(<term>,<term>) [;<process>]
    | let <pattern> = <term> in <process> [else <process>]
    | event <term> [; <process>]
```

FIGURE 3. **The process in the Applied PI Calculus.**

```
query::=
        ev:seq <ident> [; <query>]
        | evinj: seq <ident> [; <query>]
        | let <ident> = <gterm> [; <query>]
        | <gfact> [; <query>]
        | <realquery> [; <query>]
```

FIGURE 4. **The query statements in the Applied PI Calculus.**

Fig. 3 shows the process in the Applied PI Calculus. The "if-then-else" denotes "if <fact> then <process> [else <process>]", where <fact> is a condition. The function of the "let-in-else" is definition and assignment. The "let-in-else" denotes "let <pattern>=<term> in <process> [else <process>]", where the "pattern" assigns the value to "term".

The "event <term>" defines an event and generally used in authentication proof. It places before and after authentication statements and then applies a query statement to verify the authentication. The query statement "⇒" represents the events on the right of the "⇒" when something happens on the left. For example, query statement "ev: seq <ident$_1$>⇒ ev: seq <ident$_2$>". It queries the events "ev: seq <ident$_2$>" when "ev: seq <ident$_1$>" happens. The content of the "ev: seq <ident>" is the "<term>" in the "event <term>". Fig. 4 shows the query statement in the Applied PI Calculus.

The condition expressions include the simple term "<ident>: seq <term>", equivalent expression "<term> = <term>", inequivalent expression "<term><><term>". Fig. 5 shows the condition expressions in the Applied PI Calculus.

The declaration statements are composed of "free", "fun", and "new". The functions and types are declared using "fun" and "free" statements respectively. "[private]" is an optional item and means the type of the "ident" is not known with the attacker. seq <ident> denotes a sequence of ident. Channels can be defined in a statement "free". "new" defines a public variable. Fig. 6 shows the declaration in the Applied PI Calculus.

```
term::=
        <ident> (seq <term>) | (seq <term>) | <ident>
fact::=
        <ident> : seq <term> | <term> = <term>
      | <term> <><term>
```

FIGURE 5. The condition expressions in the Applied PI Calculus.

```
declaration ::=
        | free | fun | new
free ::= [private] free seq <ident>
        | [private] channel  seq <ident>
fun ::= fun <ident>/n
new::= new  <ident>
```

FIGURE 6. The declaration in the Applied PI Calculus.

B. TYPICAL USER DEFINED FUNCTIONS AND EQUATIONAL THEORY

If we want to put the Applied PI Calculus into practice, the functions and equation theory have to be defined and specified according to the special security protocols. Fig. 7 presents the typical functions and equational theory of security protocols. We model cryptography in a Dolev-Yao model as being perfect. Cryptography is composed of symmetric cipher and asymmetric cipher. The digital signature consists of a signature generation algorithm and a signature verification algorithm. Typical functions include "senc(x, key)", "sdec(y, key)", "pub(r)", "pri(r)", "aenc(x, pub)", "adec(y, pri)", "sign(x, pri)" and "versing(y, pub, x)", where encryption algorithm "senc(x, key)" used in symmetric encryption encrypts plaintext x using secrete key "key", decryption algrithm "sdec(y, key)" used in symmetric encryption decrypts the ciphertext y using secrete key "key", public key generation algrithm "pub(r)" generates the public key with the input random number "r", privatekey generation algrithm "pri(r)" produces the private key with the random number "r", asymmetric encryption algrithm "aenc(x,pub)" in asymmetric encryption encrypts the plaintext "x" with the public key "pub", decryption asymmetric decryption algrithm "adec(y,pri)" in asymmetric encryption decrypts the ciphertext "y" with the private key "pri", digital signature generation algrithm "sign(x,pri)" generates the digital signature for message "x" using the private key "pri", digital signature verification algrithm "versign(y, pub, x)" verifies the digital signature "y" for message "x" with the public key pub. The typical equational theory consists of "sdec(senc(x, key), key) = x", "adec(aenc(x, pub(r)), pri(r)) = x", and "versign(sign(x, pri(r)), pub(r), x) = true".

```
(*Typical functions*)
fun senc(x,key)
fun sdec(y,key)
fun pub(r)
fun pri(r)
fun aenc(x,pub)
fun adec(y,pri)
fun sign(x,pri)
fun versign(y, pub,x)
------------------------------------------
(*Typical equational theory*)
equation sdec(senc(x,key),key) = x
equation adec(aenc(x,pub(r)),pri(r)) = x
equation versign(sign(x,pri(r)),pub(r),x) = true
```

FIGURE 7. The typical functions of an equational theory.

IV. SUBPYTHON AND MMSP2APC

In this section, we first define the SubPython and BNF[SubPython], and then establish a Mapping Model from SubPython to Applied PI Calculus (MMSP2APC), apart from that, present a simple example for using MMSP2APC to translate the SubPython code into Applied PI Calculus. SubPython mainly contains "PassStatement", "DeclarationStatement", "CompondStatement", "ImportStatement", and "Expression". The MMSP2APC is composed of MMSP2APC[statements] (the statements mapping defined in MMSP2APC), and MMSP2APC[types] (the types mapping defined in MMSP2APC).

A. SUBPYTHON AND BNF[SUBPYTHON]

Python is a complicated programming language and widely used to develop security protocol server applications. According to the investigations on lots of SPSI written by Python from the popular open-source website Github, we find only a core part of Python, SubPython, which is used to develop SPSI. SubPython showed in Fig. 8 mainly contains "PassStatement", "DeclarationStatement", "CompoundStatement", "ImportStatement", and "Expression".

```
Statement ::= PassStatement | DeclarationStatement
        | CompoundStatement | ImportStatement | Expression
```

FIGURE 8. The BNF[statement] in the SubPython.

As shown in Fig. 9, the "PassStatement" is defined as a keyword "pass" to process the event when a statement is required syntactically analysis. There is no specific "DeclarationStatement" written by Python to represent variables and constants respectively because the declarations of variables and constants are contained in the assignment expression. Hence the "DeclarationStatement" is defined as "AssignmentExpression".

"CompoundProcess" consists of "IfStatement", "Function-Define", "ClassDefine", and "ReturnStatement". In the "IfStatement", "Expression" is a Boolean

```
PassStatement ::= pass
DeclarationStatement ::= AssignmentExpression
CompoundStatement ::= | IfStatement | FunctionDefine
                      | ClassDefine | ReturnStatement
IfStatement ::= if Expression : suite1  [else : suite2]
FunctionDefine ::=
        def FunctionName  (ArgumentList) : suite
ClassDefine ::= class ident : suite
ReturnStatement ::= return [Ident]
suite ::= stmt_list NEWLINE
stmt_list ::=  Statement (; Statement)* [;]
ident::= <IDENTIFIER_NAME>
```

FIGURE 9. The BNF[Statement] definition in the SubPython.

```
ImportStatement ::=
    import module [as name]( , module [as name] )*
    | from module import *
module ::= (ident .)* Ident
name ::= ident
```

FIGURE 10. The BNF[ImportStatement] definition in the SubPython.

value. When a Boolean value is true, the "IfStatement" executes "suite_1", otherwise, it executes "suite_2", where "suite" is a code block designated a new line and indentation. Function and class are defined in "FunctionDefine" and "ClassDefine" using the keywords "def" and "class" respectively.

From Fig. 10, the "ImportStatement" includes "import" and "from…import". The "module" is a program block. The name is designated as a specific module using keywords "as". "ReturnStatement" returns the output from the function and returns to the main function.

The "Expression" given in Fig. 11 includes "PrimaryExpression", "EqualityExpression", "FunctionCall", and "AssignmentExpression". "PrimaryExpression" includes "StringLiteral", "LongInteger", and "FloatNumber". "EqualityExpression" returns a bool using "comp_operator" which includes "==", and "! = ". "FunctionCall" invokes a function by function name "ident" and its argument list. "AssignmentExpression" defines constants, variables, assignment expressions, and logical expressions. It also assigns a value from a Rtarget which consists of "PrimaryExpression", "EqualityExpression", and "FunctionCall".

B. MMSP2APC

Based on the semantics of SubPython and Applied PI Calculus, the MMSP2APC is established in Fig. 12. The "suit" is converted into "Process". The "Socket Declaration" is translated into the "Channel Declaration". The "Message Sending Method" and "Message Receiving Method" is converted

```
Expression::=
    PrimaryExpression | EqualityExpression
    | FunctionCall | AssignmentExpression
PrimaryExpression ::=
    StringLiteral | Integer | LongInteger | FloatNumber
EqualityExpression ::= target comp_operator target
comp_operator :: == | !=
FunctionCall ::= Call ( ArgumentList )
Call ::=  ident ( . Ident )*
ArgumentList ::= positional_arguments [","  keyword_arguments]
  | keyword_arguments | "*" Expression | "**" Expression
positional_arguments ::= Expression  ("," Expression)*
keyword_arguments ::= keyword_item  ("," keyword_item)*
keyword_item ::= ident "=" expression
AssignmentStatement ::= Ltarget "=" Rtarget
Ltarget ::= PrimaryExpression | FunctionCall
Rtarget ::= PrimaryExpression
        | EqualityExpression | FunctionCall
```

FIGURE 11. The BNF[Expression] definition in the SubPython.

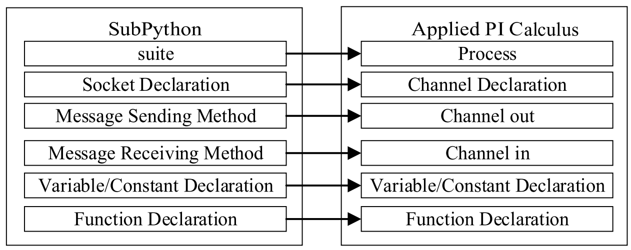

FIGURE 12. MMSP2APC.

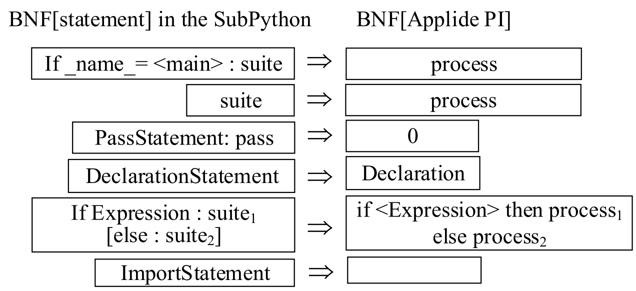

FIGURE 13. The statements mappings in MMSP2APC.

into "Channel out" and "Channel in" respectively. The "Variables or Constant Declaration" and "Function Declaration" are changed into "Variable or Constant Declaration" and "Function Declaration". Specifically, the MMSP2APC is defined by MMSP2APC[statements] and MMSP2APC [types].

1) MMSP2APC[STATEMENTS]

MMSP2APC[statements] shown in Fig. 13. The programmer usually makes an entrance using "if_name_ = main" because

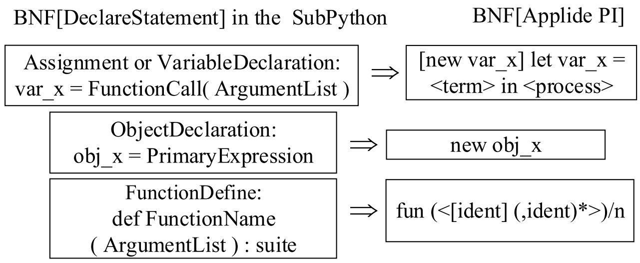

FIGURE 14. The declaration statements in MMSP2APC.

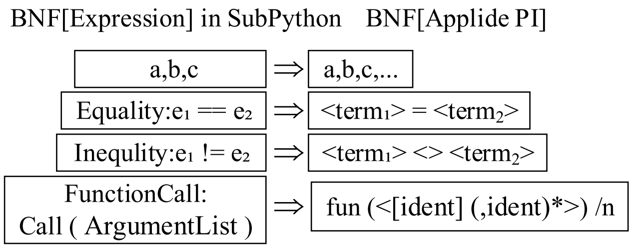

FIGURE 15. The expression statements in MMSP2APC.

Python has no program entrance. "suite" is the code block of Python. Hence "if_name_ = "main": suite" is mapped into the process in Applied PI Calculus. "PassStatement" "pass" maps into the empty process "0". "DeclarationStatement" shown in Fig. 14 includes the mappings of "assignment or VariableDeclaration", "ObjectDeclaration", and "FunctionDefine". Since the "VariableDeclaration" often created in the initialization statement and assignment statement, it maps into "new" and "let…in". The ObjectDeclaration "obj_x = PrimaryExpression" converts into "new obj_x". The "FunctionDefine" statement "def FunctionName "(ArgumentList): suit" converts into "fun (<[ident] (,ident)*>)/n". The expression statements are shown in Fig. 15. The primary expressions variables "a, b, c…" in SubPython maps into the "a, b, c…" in the Applied PI Calculus. The equality expressions is consists of "$e_1 == e_2$" (e, expression) and "$e_1 != e_2$", and it maps into the "<$term_1$> = <$term_2$>" and "<$term_1$> <> <$term_2$>" respectively. The "Call (ArgumentList)" maps into the "fun (<[ident] (,ident)*>)/n". The conditional statement "if Expression: $suite_1$ [else: $suite_2$]" is translated into "if <Expression> then $process_1$ else $process_2$". The import statement imports the necessary content. But there is no import statement in Applied PI Calculus, so the mapping of the import statement is empty.

2) MMSP2APC[TYPES]

The MMSP2APC[types] in TABLE 1 establishes the type mapping from SubPython to Applied PI Calculus in MMSP2APC. The data types of SubPython mainly include "PrimaryExpression" shown in Fig. 11. The datatype of the Applied PI Calculus is the bitstring, bool, true or false, and the user-defined data type. Plaintext, Ciphertext, Channel, Key, Signinput, Signoutput and etc. are the typical user-defined data type. In general, "PrimaryExpression" is converted into plaintext.

TABLE 1. MMSP2APC[Types].

SubPython type	Applied PI Calculus type	Type Explanation
PrimaryExpression Function definition	Plaintext	Plaintext
Function definition	Ciphertext	Ciphertext
Function definition	Channel	Channel
Function definition	Pub/Pri	Pair of asymmetric key
Function definition	Seed	Seed
Function definition	SymKey	Symmetric key
Function definition	Signinput	Input of digital sign
Function definition	Signoutput	Output of digital sign
Function definition	Hashinput	The input of hash function
Function definition	Hashoutput	The output of hash function

The semantic of data depends on the special functions used. Thus, we can only judge the data type from the function's parameter list. For example, host is a list of strings and numbers, "host = ("http://login.51talk.com", 80)". We don't sure the real semantic of host before the function "serverSocket.bind(host)" executes. This function is only used when biding sockets. Thus, the host converts into a channel declaration statement "free <ident>".

Similarly, for user-defined functions, we define the data type by the type of input and output of the function which declared in the standard library. For example, "plaintext = rsa.decrypt(crypto_tra, privkey)". Function "rsa.decrypt" is already defined in the library rsa. Thus, it converts into "let plaintext = adec(crypto_tra, pri(r))", user-defined functions asymmetric encryption presented in Fig.7, "crypto_tra" turn into ciphertext, the output of it translate into plaintext.

For non-standard security-related functions or developer-defined functions, we convert it by manual.

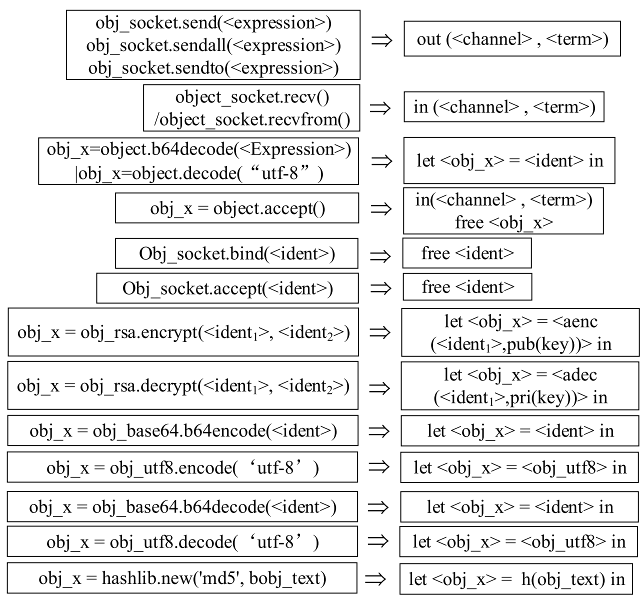

FIGURE 16. Types and statements in MMSP2APC.

Fig.16 shows the types and statements mapping based on function semantic. "obj", i.e. "ident", represents the

Part a: Python code

```
1 If __name__ == "main"
2   password = 123
3   password_Symenc = SymEncrypt(password)
4   serverSocket.send(password_Symenc)
```

⇓

Part b: Applied PI calculus

```
1 free c.
2 fun senc(x,key).
3 fun sdec(x,key).
4 Equation sdec(senc(x,key),key) = x
5 let processPython=
6   new password;
7   let password_Symenc =senc(password,key) in
8   out(c,password_Symenc)
```

FIGURE 17. An example of MMSP2APC translation.

class object. The "obj.send" and "obj.recv()" statements are message sending and receiving methods respectively, so these statements are converted into the "out (<channel>, <term>)" and "in (<channel>, <term>)" respectively. "obj_x = object.accept()" denotes that the server receives the data from the client and returns the link to the obj_x. Thus, it converts into "in(<channel>, <term>)" and "free <obj_x>". The meaning of the rest types and statements mapping as the same as these. Specifically, Encoding functions, such as base64 and utf8, do not change the security of the data. Therefore, we just pass the original value using "let…in".

3) AN EXAMPLE FOR MMSP2APC

A simple example is added to illustrate how to translate the Python code into Applied PI Calculus using MMSP2APC.

Fig. 17 shows that the server sends a password to the client, which needs to be encrypted by symmetric encryption function "SymEncrypt()". First, according to the mapping defined in Fig. 13 lines 1, If __name__ == "main" denotes a new process which should be created as processPython (Fig. 17 part b lines 5). The suite (Fig. 9 lines 8) is a set of many statements. When the compiler executes the suite, it interprets the code block iteratively. Second, "password = 123" is mapped into "new obj_x" (Fig. 14 ObjectDeclaration) which is interpreted as an integer (Fig. 11, PrimaryExpression, Integer) and is translated into a plaintext, the public variable, in Applied PI calculus. Third, "password_Symenc = SymEncrypt(password)" is converted into a "let…in" statement using the rule shown in Fig. 14 line 1. The compiler executes "SymEncrypt(password)" Iteratively using the definition shown in Fig. 11. "FunctionCall". "SymEncrypt(password)" is interpreted as "FunctionCall", where "SymEncrypt" recognized as "$ident_1$" and "password" recognized as "$ident_2$" (Fig.11, keyword_arguments). According to the semantic, the semantic of function

```
Statement::= Block
    | EmptyStatement | DeclarationStatement
    | ControlProcess | Expression
```

FIGURE 18. The BNF[Statement] in the SubJavaScript.

"SymEncrypt()" is set as symmetric encryption. The definitions of symmetric encryption for Applied PI Calculus (Fig. 7) are added. Function "$ident_1(ident_2)$" is converted to "enc($ident_2$,key)", that is to say, the function "SymEncrypt(password)" is converted to "enc(password,key)". Note that "key" in "enc($ident_2$,key)" just denotes a pair of functions, the "senc(x,key)" and "sdec(x,key)". Finally, before translating the "serverSocket.send(password_Symenc)" into "out(c,password_Symenc)", channel c is declared, which model the public channel between the sender and receiver and add "free c" at the top of part b.

V. SUBJAVASCRIPT AND MMSJS2APC

In this section, we first define SubJavaScript and BNF[SubJavaScript], and then establish the Mapping Model from SubJavaScript to Applied PI Calculus (MMSJS2APC). The SubJavaScript Statement consists of the "Block", "EmptyStatement", "DeclarationStatement", "ControlProcess", and "Expression". The MMSJS2APC establishes the MMSJS2APC[statements] (the statements mapping defined in MMSJS2APC), and MMSJS2APC[types] (the types mapping defined in MMSJS2APC).

A. SUBJAVASCRIPT AND BNF[SubJavaScript]

JavaScript is a complex programming language and widely used in security protocol client applications. According to the analysis of a large amount of SPCI written by JavaScript, the open-source website: Github, we find that only the key components of JavaScript, SubJavaScript, are used to develop SPCI. SubJavaScript mainly consists of "Block", "EmptyStatement", "DeclarationStatement", "ControlProcess", and "Expression". The BNF[Statement] of the SubJavaScript is shown in Fig. 18.

The components of BNF[Statement] shown in Fig. 19. "Block" is a code block. It allows you to use multiple statements where JavaScript expects only one statement. "EmptyStatement" is a semicolon indicating that no statement will be executed. "ControlProcess" is the same as SubPython. "DeclarationStatement" includes "Variable-Declaration", "ConstDeclaration", "ObjectDeclaration" and "FunctionDeclaration". In the "VariableDeclaration", Variables are declared using the keywords "Var" and "let". Keyword "Var" declares the triple variable, signal variable, local variable, and global variables. Keyword "let" declares the variables in the block level scopes. "const" declares a const object which value cannot change in a certain range. Object declaration uses the keyword "new" to create an object and then calls the constructor to initialize the object. "FunctionDeclaration" consists of function name and function body.

```
Block::= { (Statement_List)? }
StatementList ::= (Statement)+
EmptyStatement::= ;
DeclarationStatement::=
      | VariableDeclaration | ConstDeclaration
      | ObjectDeclaration | FunctionDeclaration
ControlProcess::= IfStatement | ReturnStatement
VariableDeclaration::=
            ( var | let )<VariableDeclarationList>( ; )
VariableDeclarationList::=
            VariableDeclaration( ","VariableDeclaration)*
VariableDeclaration ::= ident(Initialiser)?
Initialiser ::= "="AssignmentExpression
ConstDeclaration::= Const <ident> = <ident>( ; )?
ObjectDeclaration::= <ident> = new <FunctionCall>
FunctionDeclaration ::= "function" ident
      ( "(" (FormalParameterList)? ")" )FunctionBody
FormalParameterList ::= ident( ","ident)*
FunctionBody ::= "{" (SourceElements)? "}"
SourceElements ::= (SourceElement)+
SourceElement ::=FunctionDeclaration|Statement
IfStatement::= if ( Expression ) Statement ( else Statement )?
ReturnStatement ::= return ( Expression )? ( ; )?
```

FIGURE 19. The components of BNF[Statement] in the SubJavaScript.

"Expression" shown in Fig.20 consists of "PrimaryExpression", "EqualityExpression", "FunctionCall", and "AssignmentExpression". Considering that the logical mathematic part does not exist in Applied PI Calculus, we do not involve this part in SubJavaScript. The "PrimaryExpression" includes "literal", and "ident", and "variable". "Literal" is composed of number, character, and bool. "ident" is the name of variable and function. "Variable" is uncertain values. The program searches and executes a variable. If it does not exist, the program returns "undefined", otherwise it returns the value of the variable. The "EqualityExpression" estimates the relation of the values. In SubJavaScript, "==" and "===" represent the equal relation, while "! = " and "!==" represent the unequal relation. For example, the operator "===" compares the values and the type on both sides. If the values and types on two sides are the same, it returns true. Otherwise, it returns false. "FunctionCall" and "AssignmentExpression" in SubJavaScript are the same as SubPython.

B. MMSJS2APC

Based on the semantics of SubJavaScript and Applied PI Calculus, the MMSJS2APC is constructed in Fig. 21. Javascript is driven by events that call the functions when

```
Expressiont::=
     PrimaryExpression | EqualityExpression
     | FunctionCall | AssignmentExpression
PrimaryExpression ::= this
                    | Literal | ident | Variable
Literal ::= ( <DECIMAL_LITERAL>
        | <HEX_INTEGER_LITERAL>
        | <STRING_LITERAL>
        | <BOOLEAN_LITERAL> )
ident::= <IDENTIFIER_NAME>
EqualityExpression ::=
        Expression  ( Operator Expression )*
Operator :: = (  ==  |  !=  |  === |  !== )
FunctionCall ::= MemberExpression Arguments
                 ( FunctionCallPart )*
MemberExpression ::=
           ( (FunctionExpression|PrimaryExpression)
FunctionExpression ::= "function" (ident)?
        ( "(" (FormalParameterList)? ")" )FunctionBody
FunctionCallPart ::= Arguments
          | ( [ Expression ] ) | ( . ident )*
Arguments ::= ( ( ArgumentList )? )
ArgumentList ::= AssignmentExpression
               ( , AssignmentExpression )*
AssignmentExpression::=
         FunctionCall | MemberExpression Operator
         AssignmentExpression;
```

FIGURE 20. The BNF[Expression] in the SubJavaScript.

events are triggered. We turn "Function" into the "Process". The "Request Create" translates into the "Channel Declaration". The "Message Sending Method" and "Message Receiving Method" are converted into "Channel out" and "Channel in". The "Variable/Constant Declaration" and "Function Declaration" are changed into "Item Creation" and "Function Declaration", respectively. Specifically, the MMSJS2APC is defined by MMSJS2APC [statements] and MMSJS2APC [types].

1) MMSJS2APC[STATEMENTS]

The statements in MMSJS2APC are shown in Fig. 22. The EmptyStatement ";" is converted into the process "0". SubJavaScript is events driven, so we define the events function "function click_name(<Expression>) {(statement)*}" as the program entrance and maps it into "Process" because javascript is usually started by click events. "if <Expression> <Statement$_1$> else <Statement$_2$>" is translated into "if <Expression>

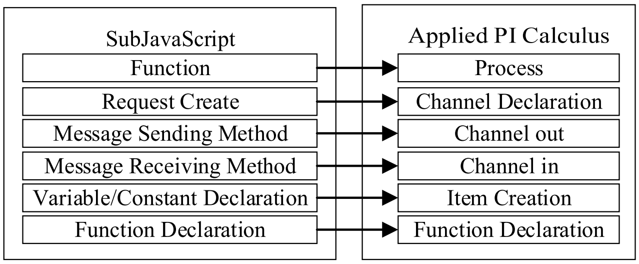

FIGURE 21. The MMSJS2APC.

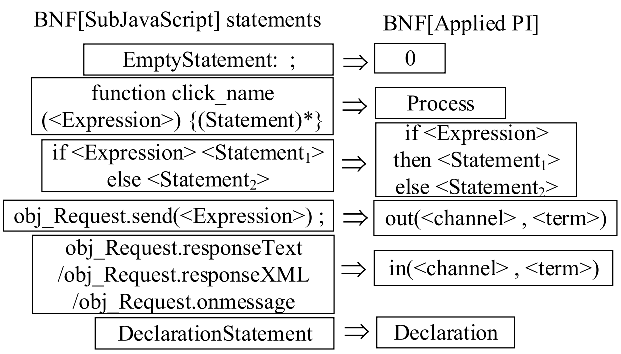

FIGURE 22. The statements in MMSJS2APC.

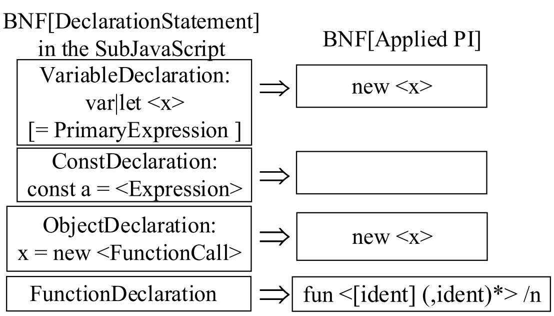

FIGURE 23. The declaration statements in MMSJS2APC.

then Statement$_1$ else Statement$_2$". "obj_Request.send (<Expression>);" is converted into "out(channel, term)". "obj_Request.responseText" and the other functionally similar functions are interpret into "in(<channel>, <term>)".

Declaration statement showed in Fig. 23, "var" and "let" are used to declare the variable. The "VariableDeclaration" statement "var|let <x> [=PrimaryExpression];" is converted into "new <x>" "ConstDeclaration" statement "const x = <Expression>" is mapped into empty because Applied PI Calculus does not support const. The "ObjectDeclaration" statement " x = new <FunctionCall>" is mapped into "new <x>". Function declaration statements map into "fun <[ident] (,ident)* > /n".

From Fig. 24, The primary expressions variables "a,b,c…" are changed into "a,b,c…". The relational expressions "Equality: e$_1$ == e$_2$" (e, expression), "Inequality: e$_1$! = e$_2$", "StrictEquality: e$_1$ === e$_2$", and "StrictInequality: e$_1$! == e$_2$" is converted into "<term$_1$> = <term$_2$>", "<term$_1$> <> <term$_2$>", "<term$_1$> = <term$_2$>", and "<term$_1$> <> <term$_2$>", respectively. The "FunctionCall" is translated into the "fun <[ident] (,ident)* > /n". Assignment statement "assignment: <x>=<Expression>" is turned to "let…in". The object function calls "object.send(<Expression>);".

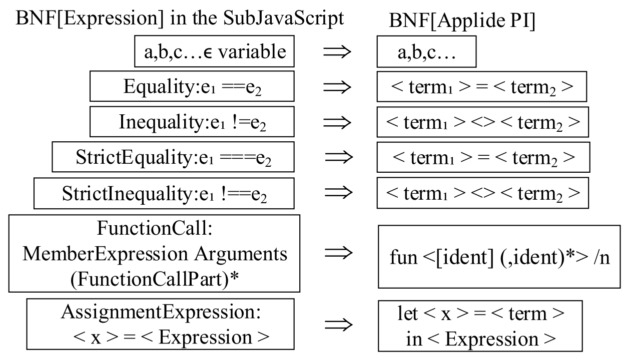

FIGURE 24. The expression statements in MMSJS2APC.

2) MMSJS2APC[TYPES]

MMSJS2APC[types] in table 2 establishes the type mapping from SubJavaScript to Applied PI Calculus in MMSJS2APC. The data types of SubJavaScript mainly includes literals shown in Fig. 20. The semantic of data depends on what functions used. Thus, we can only judge the data type according to the function's parameter list. The type mapping methods of MMSJS2APC is similar to the SubPython. Fig. 25 shows the types and statements mapping based on the function's parameter list semantic.

TABLE 2. MMSJ2APC[Types].

SubJavaScript Type	Applied PI Calculus type	Type Explanation
Literal Function definition	Plaintext	Plaintext
Function definition	Ciphertext	Ciphertext
Function definition	Channel	Channel
Function definition	Pkey	Public key
Function definition	Seed	Seed
Function definition	Skey	Secret key
Function definition	Key	Type of key
Function definition	Signinput	Input digital sign
Function definition	Signoutput	Output digital sign
Function definition	Hashinput	The input of hash function
Function definition	Hashoutput	The output of hash function

VI. SUBPYTHON2PV AND SUBJAVASCRIPT2PV

Based on MMSP2APC and MMSJS2APC, we develop the SubPython2PV tool and the SubJavaScript2PV tool using the JavaCC. JavaCC is an open-source parser generator for Java code developed by the SUN corporation. SubPython2PV accepts SPI in SubPython as input and produces SPSI in Applied PI Calculus. Similarly, SubJavaScript2PV outputs the SPCI written by Applied PI Calculus. Then, we combine the SPSI written by Applied PI Calculus and the SPCI written

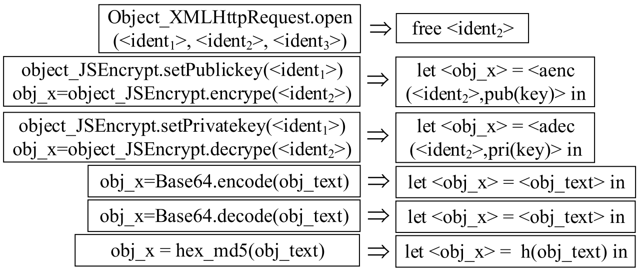

FIGURE 25. Types and statements in MMSJS2APC.

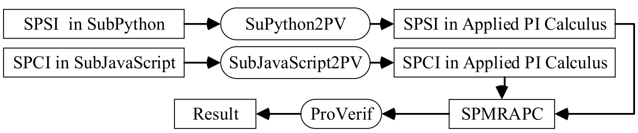

FIGURE 26. The framework of SPWI analysis.

by Applied PI Calculus to construct the Security Protocol Model Represented by Applied PI Calculus (SPMRAPC). After that, SPMRAPC is processed by ProVerif to verify the SSPI. Fig. 26 presents the framework of SPWI analysis.

A. THE DEVELOPMENT OF SUBPYTHON2PV AND SUBJAVASCRIPT2PV

The modules of the SubPython2PV are the same as SubJavaScript2PV, which includes the lexical analysis module, parsing module, translation module, and code generation module. Fig. 27 shows the modules of the SubPython2PV. First, we prepare the SPI and use the lexical analysis module to analyze and verify the correctness of the SPI according to the syntax of SubPython. If verification is successful, the lexical elements, for example, tokens, are generated. Second, the parsing module is used to address tokens and produce an abstract syntax tree, which is used to express the structure of the SPI. Third, the translation module is used to map the abstract syntax tree into an abstract syntax tree. Finally, the code generation module obtains the abstract syntax tree and produces the SPSI written by Applied PI Calculus.

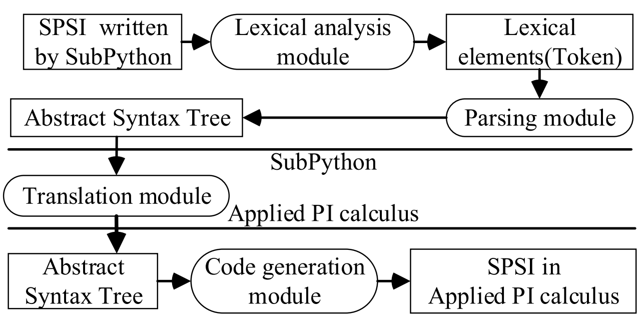

FIGURE 27. The modules of the SubPython2PV.

B. ANALYSIS

SubPython2PV and SubJavaScript2PV are semi-automated tools. The inputs to the SubJavaScript2PV and the SubPython2PV tool is a piece of pre-processed security-related code selected from the SPCI in JavaScript and SPSI in Python by manual. The outputs are SPSI written by Applied PI Calculus and the SPCI written by Applied PI Calculus. Selecting a piece of security-related code of JavaScript from the web client application is a manual process because JavaScript code is embedded in Html combined with CSS. For Python, we just extract the security-related code from server-side code files. Before sending the code to the tools, the mapping model MMSP2APC and MMSJS2APC still need to add some statement mapping or type mapping because we don't sure whether the functions used in the web application are from the standard library. As Fig. 17 shows, the non-standard function "SymEncrypt(x)" marked with symmetric encryption maps into "senc(x,key)", and the type of "password_Symenc" in Applied PI Calculus is assigned by "senc(x,key)"(Ciphertext). Finally, SPSI written by Applied PI Calculus and the SPCI generated by SubPython2PV and SubJavaScript2PV is combined into SPMRAPC.

VII. EVALUATION

This section uses SubJavaScript2PV, SubPython2PV, and ProVerif to evaluate the security of the four wildly used SPWIs which include the 51 Talk user login protocol [30], the data transfer protocol in the Mall Management system [31], the login protocol in the Yingtuo Securities [32], and registration protocol in the Miku Diversified Interfusion Platform [33]. First, SubJavaScript2PV and SubPython2PV extract SPMRAPCs from the four wildly used SPWIs, and then ProVerif will take the SPMRAPCs as input and generates the security analysis results. The experimental results show that these four SPWIs have confidentiality, but lack of authentication.

A. USER LOGIN PROTOCOL IMPLEMENTATIONS SECURITY ANALYSIS IN 51 TALK

The SPCI of the 51 Talk login protocol is presented in Fig. 28. The "userid" and encrypted "password" are sent to the server when the function "onclick" (①) is executed. Code ② shows that "userid" sends to the webserver directly when "sso_switch" is false. "encrypted = encrypt.encrypt (password)"(④) obtains the "public_key" from statement (③) through statement "encrypt.setPublicKey(public_key)" and produces the encrypted password. "xhr.send(password)" (⑤) sends the password to the server.

The SPSI of 51 Talk login protocol presents in Fig. 29. "(pubkey, privkey) = rsa.newkeys(1024)"(①) generates the pubkey and privkey. Code "while" (②) is waiting for the link. When statement ③ receives client socket successfully, code "clientSocket.send(pubkey)" (④) sends the "pubkey" to the client. statement ⑤ receives the data from the client and statement ⑥ decrypt the data using RSA. The structure of 51 Talk login protocol is shown in Fig. 30.

Before passing the original SPCI in JavaScript and SPSI in Python to the SubJavaScript2PV and SubPython2PV, there

```
function getPublicKey(clientid){
  var xhr;
  xhr=new XMLHttpRequest();
  xhr.open(post,ori_login_url,true);
  xhr.send(clientid);
  function onreadystatechange(){
  if(xhr.readyState==4){
   if(xhr.status == 200){
     return xhr.responseText;
     }else{
     console.log(Error);
    }
     return false;
   }
  }
 }

①function onclick(){
  if(sso_switch!=true){
  var xhr,msg;
  xhr=new XMLHttpRequest();
  xhr.open(post,ori_login_url,true);
② xhr.send(userid);
  function onreadystatechange(){
  if(xhr.readyState==4){
   if(xhr.status == 200){
     console.log(xhr.responseText);
     }else{
     console.log(Error);
    }
     return false;
   }
  }
 }

  var public_key,username,password;
③public_key=ssoController.getPublicKey(userid);
   if(user==null){
    console.log(Error);
    if(password==null){
      console.log(Error);
     }
   }
  if(public_key==null){
  public_key=ssoController.getPublicKey(userid);
  }
  var encrypt;
  encrypt=new JSEncrypt();
  encrypt.setPublicKey(public_key);
  var encrypted;
④encrypted=encrypt.encrypt(password);
 password=Base64.encode(encrypted);
  if(encrypt==null){
   password=hex_md5(password);
  }
  var xhr;
  xhr=new XMLHttpRequest();
  xhr.open(post,ori_login_url,true);
⑤xhr.send(password);

 function onreadystatechange(){
  if(xhr.readyState==4){
   if(xhr.status==200){
    console.log(xhr.responseText);
   }else{
    console.log(Error);
   }
  }
 }
 }.
```

FIGURE 28. The SPCI of the 51 Talk login protocol.

```
#/usr/bin/env python
# -*- coding: utf-8 -*-
import rsa
import socket
import base64
import hashlib
import sys
import base64
if __name__ == "__main__":
  serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  host = ("http://login.51talk.com/sso/login/ori_login_url", 80)
  serverSocket.bind(host)
  serverSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
  serverSocket.listen(5)
  print("server running")
① (pubkey, privkey) = rsa.newkeys(1024)
② while True:
    print("getting connection")
  ③clientSocket.addressInfo = serverSocket.accept()
  ④clientSocket.send(pubkey)
    print("get connected")
    while True:
    ⑤receivedData = str(clientSocket.recv(2048))
      entities = receivedData.split("\\r\\n")
      for a in entities:
            a=a.split(" ")[1].split("&")
       if(a[0]== 'userid'.encode('utf-8'))
             print("backen connected...")
       else:
            if(a[1].split("=")[0] == 'password'.encode('utf-8'))
            password = a[1].split("=")[1].strip()
      crypto_tra = base64.b64decode(password.encode('utf-8').decode("utf-8"))
    ⑥plaintext = rsa.decrypt(crypto_tra,privkey)
            message = plaintext.decode('utf-8')
      print(message)
```

FIGURE 29. The SPSI of the 51 Talk login protocol.

are some special statements to deal with. In Fig. 29, "while" statemen monitor the communication to receive the messages from the server, which is not closely related to the security of the 51 Talk login protocol. Apart from that, Applied PI Calculus is a formal modeling language and it is hard to directly support the loops. At the same time, the loop may lead to non-termination of ProVerif. Hence we ignore the condition expressions in a while statement and retain the functions of the while statement by hand. "for" statement is ignored for the same reason. Besides, we add some new mappings manually in the model for the functions whose semantic cannot find in the standard library and which cannot be translated by MMSP2APC and MMSJS2APC. For example, user-defined function "onreadystatechange()" and message formatting function "split()", and so on.

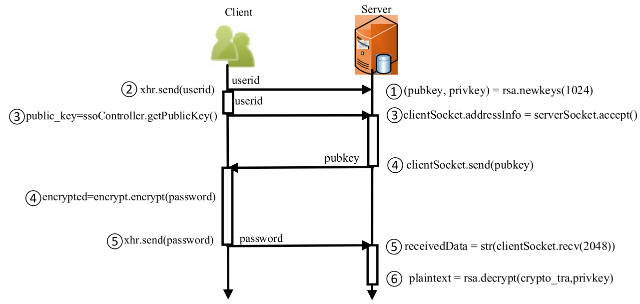

FIGURE 30. The code structure of the 51 Talk login protocol.

SPSI in Applied PI calculus

```
free ori_login_url.
let process_onclick =
if sso_switch<>true then
new xhr;
new msg;
out(ori_login_url,userid);
fun onreadystatementchange().
new public_key;
new username;
new password;
out(ori_login_url,userid);
in(ori_login_url, responseText)
let public_key = responseText in"
if username=0 then
if password=0 then 0 else 0;
if public_key=0 then
  out(ori_login_url,userid);
  in(ori_login_url, responseText)
  let public_key = responseText in";
new encrypt;
new encrypted;
let encrypted  =
        aenc(password, pub(key)) in
let password = encrypted in
if encrypt=0 then
   let password = h(password) in
out(ori_login_url,password);
```

SPCI in Applied PI calculus

```
free host.
free clientSocket.
fun aenc(x.pub).
fun adec(x,pri).
fun pub(key).
fun pri(key).

equation adec(aenc(x,pub(key)),pri(key)) = x.
let process_main =
in(host, msg1)
in(clientSocket, msg2)
out(clientSocket, pub(key)).
in(clientSocket, msg3);
let receivedData = msg3 in
let password = receivedData in
let crypto_tra = password in
let plaintext = rsa_dec (crypto_tra, pri(key)) in
let message = rplaintext  in
```

FIGURE 31. The SPMRAPC of the 51 Talk login protocol.

Fig. 31 is SPCI and SPSI in the Applied PI Calculus generated from the SubJavaScript2PV and the SubPython2PV, which takes SubJavaScript code (Fig. 28) and SubPython code (Fig. 29) as input, respectively.

In Fig. 28, the translation of SubJavaScript2PV begins with "onclick()". Function "onclick()" is translated into a process. "if(sso_switch! = true)" is mapped into "if sso_switch <> true then". "var xhr, msg;" are converted into "new xhr; new msg;". "xhr = new XMLHttpRequest();" turn into "new xhr;". The duplicate declared statement "new xhr;" are combined. "xhr.open(post,ori_login_url,true);" creates a new HTTP request. We turn it into channel "free ori_login_url." by the mapping in Fig. 25. "xhr.send (userid);" is translated into "out(ori_login_url,userid);"

TABLE 3. Analysis results in 51 Talk.

Properties and object	Formal presentation	Result
Confidentiality password	query attacker: password	True
server authenticates client Password	ev: endauthPY_JS(x) ==> ev: beginauthPY_JS(x)	False

The function "onreadystatechange()" does not relate to the security protocol, and it merely declare to "fun onreadystatementchange()." instead of translating its function body. "var public_key, username, password;" are translated into "new public_key; new username; new password;". The translation of "public_key = ssoController.getPublicKey (clientid);" is manual processing because it is not a standard function in the library. "getPublicKey(clientid)" (③) receives clientid and returns the public key. We add new mappings for this statement instead of translating all the function's body. After analyzing the function body, it converts into "free ori_login_url.", "out(ori_login_url, userid);", and in(ori_login_url, responseText), and "let public_key = responseText in". The duplicate "free ori_login_url." are combined and moved on the top of the code."if(user == null) {console.log(Error); if(password == null) {console.log (Error);}}"maps into "if username = 0 then if password = 0 then 0". Since no mapping for the function "log(error)" which does not relate to security, it maps into empty. The rest of the code transformation is not covered in detail.

Combining function definitions and channel declarations before sending Applied PI Calculus SPMRAPC to Priverif is a very easy task for manual work. After that, we use the "query attacker: password" to analyze the confidentiality of user passwords and use "query ev: endauthPY_JS(x) ==> ev: beginauthPY_JS(x)" to analyze the authentication from the server to the client. Table 3 shows that the password in the 51 Talk login protocol is confidential and the 51 Talk login protocol doesn't have authentication from server to client. Since there is no authentication mechanism in the process of password encryption, the attacker can disguise his identity. The server cannot authenticate a certain client. Therefore, the 51 Talk login protocol SPWI does not have a valid authentication mechanism.

After passing the Applied PI Calculus of SPCI and SPSI into the Proverif, there are some parameters and functions to be handled. The channels declared in SPCI and SPSI should be combined into one. The encryption functions used in SPCI and SPSI should be presented into one. Statements "query attacker" and "query ev:" should be added so as to analyze the property of confidentiality and authentication.

B. ANALYSIS ON THE SECURITY OF THE OTHER THREE IMPLEMENTATIONS

Next, we will employ the same procedure discussed above to analyze the other three SPWIs, which are the data transfer protocol in the Mall Management system, the login protocol in the Yingtuo Security, registration protocol in the Miku Diversified Interfusion Platform.

TABLE 4. Analysis results in the Mall Management system.

Properties and object	Formal presentation	Result
Confidentiality PubKey	query attacker: PubKey	True
Confidentiality DESkey	query attacker: DESkey	True
server authenticates client	ev: endauthPY_JS(x) ==> ev: beginauthPY_JS(x)	False

TABLE 5. Analysis results in the Yingtuo Securities.

Properties and object	Formal presentation	Result
Confidentiality pwAndPhone	query attacker: pwAndPhone	True
server authenticates client	ev: endauthPY_JS(x) ==> ev: beginauthPY_JS(x)	False

For the data transfer protocol used in Mall Management system, the security model depiction is described below: the client gets the PublicKey to send by the server, then the client PublicKey is encrypted by the server PublicKey, and send it to the server. After that, the server uses the client PublicKey to encrypt the DESKey, the ciphertext of the DESKey is sent to the client. The experimental result shows in table 4 that the DESKey and the client PublicKey have confidentiality, but the server can't authenticate the client. So, this protocol is equipped with confidentiality. But the server is not sure whether the client public key comes from an intended client. Hence anyone can launch the counterfeit attack.

For the Yingtuo Securities login protocol, the security model is described below: it uses the keywords "pwAndPhone" to store the user's phone number and password. The "pwAndPhone" sent from the client to the server is encrypted by the public key of the server. The experimental results show in table 5, this login protocol is equipped with confidentiality, but not has authentication from server to client. However, the server cannot authenticate the user because everyone could get the public key of the server. Hence there exists a counterfeit attack.

In the Miku Diversified Interfusion Platform registration protocol, the security model depiction is described below: The registration protocol uses "publickey_a" and "publickey_b" generated by the server which is transferred from the server to the client. The client uses the "publickey_a" and "publickey_b" to encrypt the original password and re-entered password respectively after that send it to the server. If the original password and re-entered password are the same, the server returns successful registration for a response. Otherwise, the server asks the client to re-enter the password until the original password and re-entered password are matched. The experimental result shows in table 6 that the password and re_password equipped with confidentiality,

TABLE 6. Analysis results in the Miku Diversified Interfusion Platform.

Properties and object	Formal presentation	Result
Confidentiality password	query attacker: re_password	True
Confidentiality re_password	query attacker: re_password	True
server authenticates client	ev: endauthPY_JS(x) ==> ev: beginauthPY_JS(x)	False

but lack of authentication from the server to the client. The server does not authenticate the user because the server is not sure whether the ciphertexts of the two encrypted passwords come from an intended client. Hence anyone can launch the counterfeit attack.

VIII. DISCUSSION

Here we compare our semi-automatic method to Backes' s method [13], the first automatic security analysis method of JavaScript implementation, from the application field and technology.

The Backes' s method is just suitable for JavaScript implementation. While our method is suitable for JavaScript-Python implementation, JavaScript implementation, and Python implementation.

The Backes' s method used the system dependency graphs to conservatively approximate all possible information flow within a program to detect the security vulnerability by distinguishing between explicit and implicit flows. But our method is different and is a formal method. It applies formal language Applied PI Calculus to formalize the JavaScript implementation, and Python implementation to generate the formal models presented by Applied PI Calculus, after that we use the formal tools to analyze the security properties.

IX. CONCLUSION AND FUTURE WORK

With a large number of SPWI projects developed with Python and JavaScript, it is necessary for its security. However, to our best knowledge, there is no related literature on analyzing SPSI written by Python and SPCI written by JavaScript. Therefore, this paper uses the model extraction technology to extract SPMRAPC from SPSI and SPCI and then verifies its security through ProVerif.

Our contributions in this paper are fourfold. First, we analyze the SPWI written by JavaScript and Python and define the SubJavaScript and SubPython. Second, based on the semantics, we establish MMSJS2APC and MMSJS2APC, respectively, which includes statements mapping and type mapping. Third, we develop semi-automated model extraction tools SubPython2PV and SubJavaScript2PV. Finally, we analyze the confidentiality and authentication of four SPWIs. The experimental results show that these web projects have confidentiality but lack of authentication. Our method has a wide application field.

In the future, we plan to expand SubJavaScript and SubPython to involve more statements and features. Meanwhile, we will continue to analyze more SPWIs by SubJavaScript2PV and SubPython2PV.

REFERENCES

[1] (Aug. 2019). *August Headline: Silly Season in the Programming Language World*. TIOBE. [Online] Available: https://www.tiobe. com/tiobe-index//

[2] (2018). *Python Developers Survey 2018 Results*. JetBrains. [Online]. Available: https://www.jetbrains.com/research/python-developers-survey-2018/

[3] J. Jürjens, "Automated security verification for crypto protocol implementations: Verifying the jessie project," *Electron. Notes Theor. Comput. Sci.*, vol. 250, no. 1, pp. 123–136, Sep. 2009.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, "Refinement types for secure implementations," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 2, pp. 1–45, Jan. 2011.

[5] J. GoubaultLarrecq and F. Parrennes, "Cryptographic protocol analysis on real C code," in *Proc. Int. Workshop Verification, Model Checking, Abstract Interpretation*, Berlin, Germany, Jan. 2005, pp. 363–379.

[6] J. T. Lu, L. L. Yao, X. D. He, and B. Meng, "Improvement of OpenID Connect protocol and its security analysis," *J. Comput. Appl.*, vol. 5, pp. 1347–1352, May 2017.

[7] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proc. ACM Sigplan Sigact Symp. Princ. Program. Lang.*, London, U.K., Jan. 2001, pp. 104–115.

[8] S. Chaki and A. Datta, "ASPIER: An automated framework for verifying security protocol implementations," in *Proc. 22nd IEEE Comput. Secur. Found. Symp.*, New York, NY, USA, Jul. 2009, pp. 172–185.

[9] A. Mihhail, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from C protocol code by symbolic execution," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Chicago, IL, USA, Oct. 201, pp. 331–340.

[10] P. F. Mihancea and M. Minea, "JMODEX: Model extraction for verifying security properties of Web applications," in *Proc. Softw. Maintenance, Reeng. Reverse Eng.*, Antwerp, Belgium, Feb. 2014, pp. 450–453.

[11] M. AlIbrahim and Y. S. AlDeen, "The reality of applying security in Web applications in Academia," *Int. J. Adv. Comput. Sci. Appl.*, vol. 5, no. 10, pp. 7–15, 2014.

[12] M. K. Gupta, M. C. Govil, and G. Singh, "Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in Web applications: A survey," in *Proc. Recent Adv. Innov. Eng.*, Jaipur, India, May 2014, pp. 1–5.

[13] M. Backes, C. Hammer, D. Pfaff, and M. Skoruppa, "Implementation-level analysis of the JavaScript helios voting client," in *Proc. 31st Annu. ACM Symp.*, Pisa, Italy, Apr. 2016, pp. 2071–2078.

[14] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," *J. Comput. Secur.*, vol. 22, no. 5, pp. 823–866, Jul. 2014.

[15] K. Bhargavan and A. D. Gordon, "Modular verification of security protocol code by typing," in *Proc. ACM Sigplan Sigact Symp. Princ. Program. Lang.*, New York, NY, USA, Jan. 2010, pp. 445–456.

[16] M. Backes, M. Maffei, and D. Unruh, "Computationally sound verification of source code," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, Chicago, IL, USA, Oct. 2010, pp. 387–398.

[17] N. Swamy, J. Chen, C. Fourent, P. Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *Proc. 16th ACM Sigplan Int. Conf. Funct. Program.*, Tokyo, Japan, Sep. 2011, vol. 46, no. 9, pp. 266–278.

[18] N. Swamy, C. Hriţcu, C. Keller, P. Y. Strub, A. R. Rastogi, A. Delignat-Lavaud, K. Bhargavan, and C. Fournet, "Semantic purity and effects reunited in F*," *Proc. 20th ACM SIGPLAN Int. Conf. Funct. Program.*, vol. 31, Vancouver, QC, Canada, Aug. 2015, pp. 1–19.

[19] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Y. Strub, and M. Kohlweiss, "Dependent types and multimonadic effects in F*," in *Proc. 43rd Annu. ACM SIGPLANSIGACT Symp Princ. Program. Lang.*, New York, NY, USA, Jan. 2016, Vol. 51, no. 1, pp. 256–270.

[20] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 1, pp. 1–61, Dec. 2008.

[21] K. Bhargavan, R. Corin, and C. Fournet, "Automated computational verification for cryptographic protocol implementations," to be published.

[22] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Cryptographically verified implementations for TLS," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, Virginia, NV, USA, Oct. 2008, pp. 459–468.

[23] M. Aizatulin, A. D. Gordon, and J. Jurjens, "Computational verification of C protocol implementations by symbolic execution," in *Proc. 19th ACM Conf. Comput. Commun. Secur.*, Raleigh, NC, USA, Oct. 2012, pp. 712–723.

[24] K. Bhargavan, K. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *Proc. 38th IEEE Symp. Secur. Privacy*, San Jose, CA, USA, May 2017, pp. 483–502.

[25] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Trans. Dependable Secure Comput.*, vol. 5, no. 4, pp. 193–207, Oct. 2008.

[26] *ProVerif: Cryptographic Protocol Verifier in the Formal Model*. Accessed: Oct. 2016. [Online]. Available: https://prosecco.gforge.inria.fr/personal/bblanche/proverif/

[27] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc.14th IEEE Comput. Secur. Found. Workshop*, Cape Breton, NS, Canada, Jun. 2001, pp. 82–96.

[28] N. Oshea, "Using ELYJAH to analyse Java implementations of cryptographic protocols," in *Proc. Joint Workshop Found. Comput. Secur.*, Pittsburgh, PA, USA, Jun. 2008, pp. 211–223.

[29] Z. Li, B. Meng, D. Wang, and W. Chen, "Mechanized verification of cryptographic security of cryptographic security protocol implementation in JAVA through model extraction in the computational model," *J. Softw. Eng.*, vol. 9, no. 1, pp. 1–32, Jan. 2015.

[30] *The Source Code of 51 Talk User Login Protocol*. Accessed: Jun. 2017. [Online]. Available: https://github.com/zkyeu/wechat

[31] *The Source Code of Data Transfer Protocol in the Mall Management System*. Accessed: Jul. 2018. [Online]. Available: https://github.com/xulayen/pgyer-node-api

[32] *The Source Code of Login Protocol in the Yingtuo Securities*. Accessed: Mar. 2018. [Online]. Available: https://github.com/mysisd/ytrxapp

[33] *The Source Cold of Registration Protocol in the Miku Diversified Interfusion Platform*. Accessed: Apr. 2017. [Online]. Available: https://github.com/katherinewei/miku

[34] R. Sisto, P. Bettassa Copet, M. Avalle, and A. Pironti, "Formally sound implementations of security protocols with JavaSPI," *Formal Aspects Comput.*, vol. 30, no. 2, pp. 279–317, Mar. 2018.

[35] R. Küsters, T. Truderung, and J. Graf, "A framework for the cryptographic verification of Java-like programs," in *Proc. IEEE 25th Comput. Secur. Found. Symp.*, Cambridge, MA, USA, Jun. 2012, pp. 198–212.

[36] B. Meng, X. D. He, J. L. Zhang, L. L. Yao, and J. T. Lu, "Security analysis of security protocol Swift implementations based on computational model," *Chin. J. Commun.*, vol. 39, no. 9, pp. 182–194, Sep. 2018.

[37] E. Abgrall, Y. L. Traon, S. Gombault, and M. Monperrus, "Empirical investigation of the Web browser attack surface under cross-site scripting: An urgent need for systematic security regression testing," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation Workshops*, Cleveland, OH, USA, Apr. 2014, pp. 34–41.

[38] G. D. Bai, J. Lei, G. Z. Meng, S. S. Venkatraman, P. Saiena, J. Sun, Y. Li, and S. Dong, "AUTHSCAN: Automatic extraction of Web authentication protocols from implementations," in *Proc. 20th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2013, p. 20.

[39] K. Bhargavan, A. Delignatlavaud, and S. Maffeis, "Language-based defenses against untrusted Browser origins," in *Proc. 22nd Secur. Symp.*, Washington, DC, USA, Aug. 2013, pp. 653–670.

XUDONG HE was born in 1991. He received the M.S. degree from the School of Computer Science, South-Central University for Nationalities, China. He is currently a Research Assistant with the School of Computer, South-Central University for Nationalities. His research interests include security protocol implementations and reverse engineering.

QIN LIU was born in China, in 1990. She is currently pursuing the degree with the School of Computer, South-Central University for Nationalities, China. Her current research interests include a smart contract on blockchain and the consistency research of legal contracts and smart contract code.

SHUANG CHEN was born in China, in 1994. She received the M.S. degree from the School of Computer Science, South-Central University for Nationalities, China. Her current research interests include automatic generation and verification of security protocol implementations.

CHIN-TSER HUANG received the Ph.D. degree in computer science from the University of Texas at Austin, Austin, TX, USA. He is currently a Professor with the Department of Computer Science and Engineering, University of South Carolina, where he is also the Director of the Secure Protocol Implementation and Development (SPID) Laboratory. His current research interests include network security, network protocol design and verification, secure computing, and distributed systems.

DEJUN WANG was born in 1974. He received the Ph.D. degree in information security from Wuhan University, China. He is currently an Associate Professor with the School of Computer, South-Central University for Nationalities, China. He has authored or coauthored more than 20 articles in international/national journals and conferences. His current research interests include security protocols and formal methods.

BO MENG was born in China, in 1974. He received the M.S. degree in computer science and technology and the Ph.D. degree in traffic information engineering and control from the Wuhan University of Technology, Wuhan, China, in 2000 and 2003, respectively. From 2004 to 2006, he worked with Wuhan University as a Postdoctoral Researcher in information security. From 2014 to 2015, he worked with the University of South Carolina, as a Visiting Scholar. He is currently a Full Professor with the School of Computer Science, South-Central University for Nationalities, China. He has authored or coauthored more than 50 articles in international/national journals and conferences. In addition, he has also published two books *Automatic Generation and Verification of Security Protocol Implementations* and *Secure Remote Voting Protocol* (China, Science Press). His current research interests include blockchain, security protocols, and formal method.