Message Format Extraction of Cryptographic Protocol Based on Dynamic Binary Analysis[1]

Meijian Li, Yongjun Wang, Peidai Xie and Zhen Huang
Computer College, National University of Defense Technology
ChangSha, China
Emails: (Li) 84647697@qq.com; (Wang) wwyyjj1971@126.com; (Xie) 348432959@qq.com;
(Huang) ZhenHuang.NUDT@gmail.com

Shangjie Jin
Military Transportation Institute of the General logistics Department
Tianjin, China
Email: jinshangjie@sohu.com

Shanshan Liu
Military Transportation University, Tianjin, China
Email: 870166743@qq.com

To maintain communications confidentiality, cryptographic protocols are widely used in more and more network applications. Moreover, some malwares even leverage these kinds of protocols to evade inspection by IDS. Observation shows that protocol implementations commonly contain flaws or vulnerabilities. Therefore, research on reverse engineering of cryptographic protocols can play an important role in improving the security of network applications, especially by providing another way to fight against malwares. Nevertheless, previous protocol reverse engineering technologies, which are based on analysis of network traces, encounter great challenges when the network messages transmitted between different protocol principals are encrypted. This paper proposes ProtocolFormat, which aims to infer the message format from dynamic execution of cryptographic protocol applications. The proposed approach is based on the observation that the process of message parsing in cryptographic protocol applications reveals rich information about the hierarchical structures and semantics of their messages. Hence, by observing calls to library function and instruction execution in network programs, the proposed system can reverse derive the formats of each protocol message (even encrypted messages). Experiments show that the message formats output by ProtocolFormat not only accurately identify message fields, but also unveil the structure of the encrypted message fields.

Keywords: *Dynamic Binary Analysis, Protocol Reverse Engineering, Taint Analysis, Protocol Field, Encryption*

ACM Classifications: *C.2.2 (Network Protocols – Protocol Verification), K.4.4 (Electronic Commerce – Security)*

1 Part of this research has been supported by National 863 High Tech Programs under Grants No. 2009AA01Z432 and the National Natural Science Foundation of China under Grants No. 60873215.

Copyright© 2014, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 14 August 2012
Communicating Editor: Seok-Hun Kim

1. Introduction

With network security technologies playing a more and more important role in people's economic activity, cryptographic protocols, as the main techniques that support network security technologies, are increasingly used in many kinds of security applications. On the other hand, to evade inspection by IDS, some malwares leverage cryptographic protocols to hide their communications. However, due to the complexity and difficulty of its design and implementation, the protocol itself or its implementation programs inevitably contain various flaws and vulnerabilities. For example, Shaikh and Bush have applied process algebra CSP (Communicating Sequential Processes) and Schneider's rank function theorem to the Woo-Lam protocol (Woo and Lam, 1992) to expose flaws in its design (Shaikh and Bush, 2005).

For security applications, identifying such flaws is crucial to improving their security; meanwhile, for malwares, discovering the vulnerabilities used in their protocols offers a new way to fight against them. For this reason, a methodological framework, which is based on dynamic binary analysis technology to reverse derive the protocol specification and discover the vulnerabilities of network applications. Nevertheless, it is necessary to solve a crucial problem before achieving this goal: how to derive the encrypted message format from dynamic execution of the cryptographic protocol application.

In this paper, the approach taken is mainly based on the technologies of protocol reverse engineering. Commonly, protocol reverse engineering is a manual, time-consuming, tedious, and error-prone process. To improve this situation, some researchers have recently presented certain kinds of automatic reverse engineering approaches, which can be categorized into two classes: network-based and program-based. For the network-based approach, representative research includes the Protocol Informatics Project (Marshall, 2004) and Discover (Weidong *et al*, 2007). This kind of approach commonly locates field boundaries from a large number of network traces by leveraging the sequence alignment algorithm, which has been used in bioinformatics for pattern discovery. The great advantage of this approach is its simplicity and ease of implementation. However, the disadvantage is also obvious; it inevitably cannot obtain semantic information only from the network traces. Moreover, according to the cryptographic protocol specification, the messages may have been encrypted during their transmission through the network. Therefore, unless a decryption of the encrypted messages is available, it is not possible to use this kind of approach to reverse extract a message format from any cryptographic protocol packet. For the program-based approach, representative studies include *Polyglot* (Caballero *et al*, 2007), *AutoFormat* (Zhiqiang *et al*, 2008), *Tupni* (Cui *et al*, 2008), *Prospex* (Comparetti *et al*, 2009), *ReFormat* (Zhi *et al*, 2009), *Dispather* (Caballero *et al*, 2009: Juan *et al*, 2009), and others. In contrast to the network-based approach, the program-based approach extracts the protocol format based on how a network program parses and processes a protocol message, which reveals plenty of information about the message format. Therefore, the program-based approach can be more accurate and has more obvious advantages than the network-based approach.

In this paper, *ProtocolFormat* is proposed, which is a system that aims to extract accurately the message format for encrypted messages sent by a security application. However, unlike *Dispather*, *ProtocolFormat* simply extracts the message format from received messages, and it is possible to extract the message format of sent messages from the other side of the protocol session. This paper also presents a novel technique to infer the field semantics of encrypted messages received by a security application. The proposed approach is based on the observation that standard library functions, which are used in security applications, reveal rich semantic information about the

message format. On the other hand, *ProtocolFormat* is mainly based on dynamic taint analysis technique (Newsome and Song, 2005), which taints the network input and observes how the tainted data propagate during the execution of a network program. The authors have implemented a prototype of the techniques based on the latest release of *Valgrind* (version 3.6.1) (Nethercote and Seward, 2007; Nethercote, 2004). *ProtocolFormat* is used to analyze the SSL handshake protocol, FTP SSL, and an unknown protocol used by a security communication application. The experimental results are exciting; *ProtocolFormat* is able to extract the hierarchical message format of the encrypted messages with high accuracy.

The rest of this paper is organized as follows. Section 2 defines the assumption and describes the problem scope of this paper. Detailed system design and key techniques for extraction of encrypted message format will be presented in Section 3. Section 4 describes implementation and evaluation results. After discussing related work in Section 5, the limitations of *ProtocolFormat* will be analyzed and possible improvements suggested in Section 6. Finally, Section 7 concludes the paper.

2. Assumption and Problem Scope

2.1 Assumption

In order to achieve security transmission, the implementations of all kinds of cryptographic protocols commonly encrypt their communication. Thus, it is difficult to obtain any semantic information from network flows, which makes it impossible to analyze this type of network application by simply using the network-based approaches. The solution proposed by this paper is to observe how network application parses network messages. Moreover, for the sake of making our solution feasible, we introduce two assumptions as follows:

a) **Standard Library Function.** We assume that cryptographic protocol applications deal with messages adopting standard library function. For instance, network socket library function "*recvfrom*", "*recv*", and others, are called by these applications when receiving messages. At the same time, messages are decrypted by calling standard decryption library functions, such as "*EVP_Decryptlnit_ex*", "*EVP_DecryptUpdate*", "*EVP_DecryptFinal_ex*" from *OpenSSL*;
b) **Message Parsing Process.** Many cryptographic protocol applications parse network messages according to a fixed pattern. Therefore, for simplifying the reverse analysis, we assume that the message parsing process can be divided into four phases: (1) receiving network package, then extracting the message payload from this package and storing it in a memory buffer; (2) extracting fields from the message and parsing them respectively; (3) if the message has encrypted message fields, the network application needs to decrypt them firstly, then parses the decrypted message fields normally; (4) end-of-message parsing.

2.2 Terminology

Generally, a message format is a predetermined or prescribed spatial arrangement of the parts of a message. This paper uses tree structure to express the message format, which reveals the hierarchical and nested structure of a network message. In the following, we give a formal definition of message format.

Message Format (MF). Let mf be a message format, mf can be viewed as a union of message fields set T and relationship set R, mf = (T,R). T contains n (n > 0) elements, $T = \{(f,s,c) | f \in \mathcal{F}\}$. $\mathcal{F}$ is the set of message field types which describe the attribute of the message field. In the current version of our system, $\mathcal{F}$ supports the field types of Length, Text, Binary Data, Key, RSA Encrypt, DES Encrypt

and AES Encrypt. *s* is a nature number that denote the field length, and *c* is the content of this message field. R is the set that describes the relationship between message fields within T, $R = \{(t_i, t_j, r) | t_i, t_j \in T, r \in \Upsilon\}$. Υ is the relationship types set, which includes Father-Son, Son-Father, Older-Younger Brother, Younger-Older brother and Other relationship.

2.3 Problem Scope

The goal of protocol reverse engineering is to extract protocol format, which includes protocol message structure and protocol model that can be represented as a protocol state machine or protocol specification. Reversely extracting protocol model commonly comprises two steps: Firstly, given a set of network packets, this technique extracts the message format for each protocol message; secondly, given a set of message formats, it infers the multiple message types that comprise the protocol, and extracts the state machine that represents the running process of the protocol and the protocol specification that formally describes the protocol.

This paper only deals with the first step of the protocol model extraction. The goal of the proposed approach is to derive the message format of received messages (especially the encrypted message) by observing the dynamic execution of message parsing process in network program. However, there are two main challenges the authors have to face: the first one is to infer message field semantics that will be used in the future work to reverse infer protocol model. In this paper, we focus on typical field semantics, i.e. length, timestamps, encryption, hostnames. The second one is accurately locating the phase transition points during the dynamic execution of network application which is more difficult. According to the assumption, the message parsing process is divided into four phases, however, cryptographic protocol messages commonly consist of several plain-text fields and encrypted fields, and even worse, there may be other encrypted fields in an encrypted field. Take SSL Handshake Protocol (Eric, 2001; David and Bruce, 1996) as example, which is responsible for "secure session establishment" between two network applications, the server and the client interact with each other with several types of messages during the handshake phase, and every message type consists of several (encrypted) message fields. Table 1 shows the message types of SSL handshake protocol and the corresponding number of their message fields.

Message Type	Message Fields	Encrypted Message Field
ClientHello	9	0
ServerHello, Certification, ServerHelloDone	24	0
ClientKeyExchange, ChangeCipherSpec, Finished	16	4
ChangeCiperSpec, Finish	11	4

Table 1: SSL Handshake Protocol

Consequently, a new approach is required to locate the transition points that identify which message parsing phase the cryptographic protocol application is in. The intuition behind our approach is simple and effective: the standard library functions contain enough semantic information to illustrate what the program is doing. According to the assumption, cryptographic protocol application decrypts incoming messages by using standard decryption library, such as DES_Decrypt() and *AES_Decrypt()* in *OpenSSL* cryptography library. These cryptography library functions are so powerful that they are widely used in most security applications, even malwares, such as *Agobot*. In order to support the calling of library functions from external program modules,

library file must export the function symbols and the corresponding entry address in the file. As a result, the transition points can be located by searching the instruction address where the cryptographic protocol application calls these library functions.

3. System Design

3.1 Approach Overview

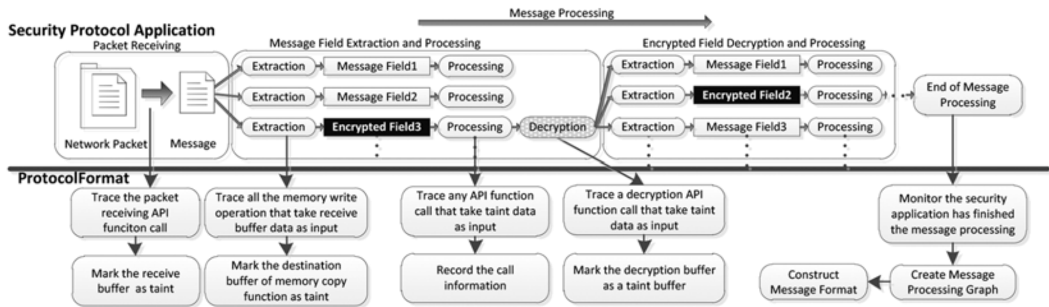


Figure 1: ProtocolFormat Working Process

Depending on the aforementioned approach that locates the transition points between two message parsing processes, the working process overview of the proposed system is shown in Figure 1. The simulative execution of cryptographic protocol application is under *ProtocolFormat*, therefore, the working process of *ProtocolFormat* shown in Figure 1 corresponds to each message parsing process in cryptographic protocol application.

According to the characteristics of standard library functions called in different message parsing phases, the proposed system works as follows:

1. **Network Packets Receiving.** In this phase, cryptographic protocol application reads off message data from network packets. There are a limited number of standard library functions accomplishing the task, such as *read*, *recv*, *recvfrom*, *recvmsg*, which are capable of reading off network flow from a socket. Thus, the call addresses where the application read off network packet from the network is the source point to start the tainted analysis, and *ProtocolFormat* firstly marks the buffer that stores the received message as a tainted buffer after the application completing these function calls, and then tracks and traces the taint propagation during the execution of the application. For the mark operation, we will introduce it in section 3.2.2.
2. **Extraction and Parsing of Message Field.** Generally, a cryptographic protocol message consists of several message fields, such as length field, encrypted field, and others. Therefore, before the cryptographic protocol application further parses message fields, we assume it must firstly extract the corresponding message data from the receive buffer, and this process commonly is accomplished by calling the standard memory operation function, such as *memcpy*. For this reason, *ProtocolFormat* needs to observe all data movement instructions that take the receive buffer bytes as input and the functions that the data movement instructions belong to. Meanwhile, *ProtocolFormat* marks the destination buffer of memory operation as tainted buffer and every byte in the tainted buffer as tainted data. After that, *ProtocolFormat* tracks a small subset of x86 instructions that take the tainted data as source location, and records the relevant information on a tainted tab (introduced in section 3.2.3) that associates with this tainted data.

In addition, these instructions simply move tainted data around, but without modifying it. For some tainted data that there isn't any decryption function call take as input parameter until the cryptographic protocol application completes the message parsing, we mark them as plaintext, and divide the ones that have the similar dependency chain into a plaintext field (dependency chain will be introduced in section 3.2.2). Nevertheless, for the other tainted data that decrypted functions take as input parameter, we mark them as ciphertext, and divide the ones that possess the same dependency chain into an encrypted message field (comparing to plaintext tainted data, the last called function in dependency chain is a decrypt library function call).

3. **Decryption and Processing of Encyrpted Field.** Due to messages sent among different cryptographic protocol applications being encrypted, it is necessary to identify the memory buffers that hold the decrypted data when the decryption operation finishes. In *ProtocolFormat*, we observe whether the call instructions for decryption library functions are executed; if the decryption functions take the tainted buffer as source argument, then the decryption buffer will be marked as a tainted buffer, and the source buffer will be marked as an encrypted message field. In this paper, we focus on these decryption standard library functions that belong to cryptograph library OpenSSL, such as *EVP_DecryptInit_ex()*, *EVP_DecryptUpdate()*, *EVP_DecryptFinal_ex().* After the application has decrypted the encrypted message field, we assume that it will further parse the decrypted message field. Therefore, the following task of *ProtocolFormat* is a recursive process. During each iteration, it marks the source buffer as an encrypted message field and marks the new decryption buffer as a tainted buffer. The iteration continues until the application finishes the message parsing process.
4. **End-of-Message Parsing.** Unfortunately, for lacking of enough knowledge about the protocol message format, it is difficult to identify the point when the cryptographic protocol application is finishing the message parsing. In *ProtocolFormat*, two approaches are used to overcome this difficulty (introduced in section 3.2.3). After the end transition point of the message parsing has been located, *ProtocolFormat* will construct a "message parsing process tree" according to the tainted information that associate with the tainted data. In a way, "message parsing process tree" describes in detail how the cryptographic protocol application process an incoming message, thus it is enough to derive the message format from this tree, and the detail will be introduce in section 3.2.4.

3.2 System Architecture

According to the differentiation of function definition, the proposed system consists of five key modules as shown in Figure 2:

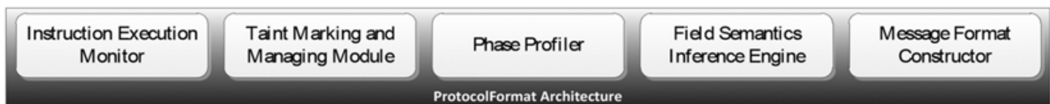


Figure 2: ProtocolFormat System Architecture

Instruction Execution Monitor. By instrumenting the data movement instructions as well as call instructions, this module can observe the dynamic execution of a cryptographic protocol application. Moreover, by observing program execution, the module can intercept network-received function call, memory-related function call and decryption function call, and taint-mark the result buffer.

Taint Marking and Managing Module. This module provides other modules with taint-mark operation. Meanwhile, it also takes the responsibility for managing tainted information associated with the taint data.

Phase Profiler. According to the information that is provided by instruction execution monitor, it could infer which phase the cryptographic protocol application is in, and inform other modules to do the corresponding operation.

Field Semantics Inference Engine. Based on the semantics of function and instruction that operate on tainted data, this module infers the semantics of each byte and field in messages.

Message Format Constructor. When the cryptographic protocol application finishes the processing of an incoming message, this module firstly constructs "message parsing process tree", then creates message format from this tree.

3.2.1 Instruction Execution Monitor

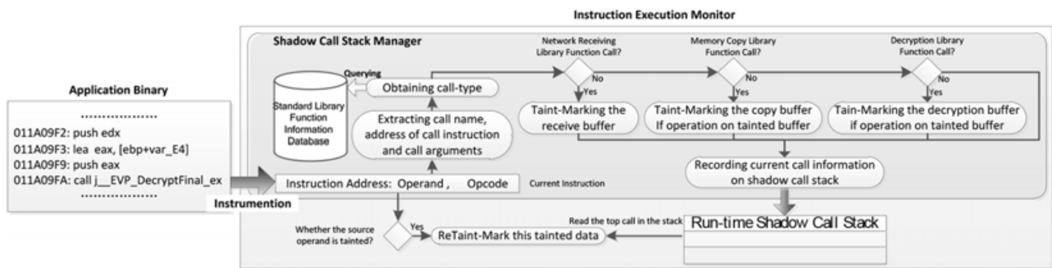


Figure 3: Instruction Execution Monitor Architecture

Instruction	Instruction Description
call	Call Procedure
mov	Move to/from Registers/Memory
movsx	Move with Sign-Extension
movzx	Move with Zero-Extend
push	Push Word or Doubleword Onto the Stack
stos	Store String
pop	Pop a Value from the Stack
xchg	Exchange Register/Memory with Register
bswap	Byte Swap

Table 2: Instructions that ProtocolFormat handles

The architecture of instruction execution monitor as shown in Figure 3. Similar to previous program-based approaches, ProtocolFormat also needs to monitor the execution of a cryptographic protocol application. However, unlike the others, it is just interested in a small subset of x86 data movement instructions. The list of the subset instructions is described in Table 2.

Shadow call stack manager. For the function call instructions, we obtain the run-time shadow call stack by instrumenting this kind of instructions. Although we can traverse the current stack frames to acquire the run-time call stack information, and if the debug information is embedded in the

binary, we can even obtain the relevant function names from the return address, noting that this method works well only when the program or library is built with stack frame pointer support. For overcoming this problem, we design shadow call stack manager to maintain a run-time shadow call stack inside the execution monitor. The shadow call stack contains the call instruction address, function name, call address, and call arguments for all the functions called so far. The working process of the shadow call stack manager can be divided into four phases:

a) When cryptographic protocol application is executing a function call instruction, shadow call stack manager extracts the call address, called function name, instruction address, and arguments of this function call. For obtaining the called function name, if debug info is available, *Valgrind* supports a convenient way to derive the function name from the call address. For obtaining the values of call arguments, firstly, shadow call stack manger query the *Standard Library Function Information Database*, if the called function is a standard library function, we will obtain the definition and semantic information of the parameter from this database, and then acquire the value of each parameter by traversing the stack frame. Otherwise, if the called function is not a standard library function, we will get nothing after querying the database, so just set the call arguments of this function call to NULL.
b) The next step is to obtain call-type of the called function. In the proposed system, we define four kinds of call-type: they are network receiving call "R", memory copy call "M", decryption call "D", and other call "O". In the design of *ProtocolFormat*, *Standard Library Function Information Database* has defined the prototype of each function, thus the call-type of a function call can be derived from the information in the last database query.
c) For different call-types of functions, they are processed in different ways in the shadow call stack manager. Firstly, if the call-type is "R" (network receiving call), the receive buffer will be taint-marked. Secondly, if the call-type is "M" (memory copy call), the destination buffer will be taint-marked. Thirdly, if the call-type is "D" (decryption call), the decryption buffer will be taint-marked.
d) Finally, we will record the corresponding information of the current function call on the run-time shadow call stack. The information contains call address, called function name, address of call instruction, and call arguments. After executing the return instruction, the information of the returned call will be removed from the run-time shadow call stack.

Retaint-mark tainted data. For other types of instructions that *ProtocolFormat* handles, we need to check whether the source operand is a tainted data, and if true, this data will be taint-marked again. More especially, take a data movement instruction as example, if the instruction takes a tainted data as source operand, then a new item that is about the data movement operation will be appended to the dependency chain.

3.2.2 Taint Manager

This module primarily implements two main functions: providing other modules with taint-related operations, and the management and maintenance of the tainted buffer information table (TBIT).

Before further discussion about how to implement the taint-related operations, we firstly introduce the data structure that this module processes and maintains. In the proposed system, there are three data structures relative to tainted data, they are TBIT node data structure, tainted buffer descriptor data structure, and tainted tab data structure. Meanwhile, all these data structures are stored and kept in the tainted buffer information table (TBIT) until *ProtocolFormat* extracts the

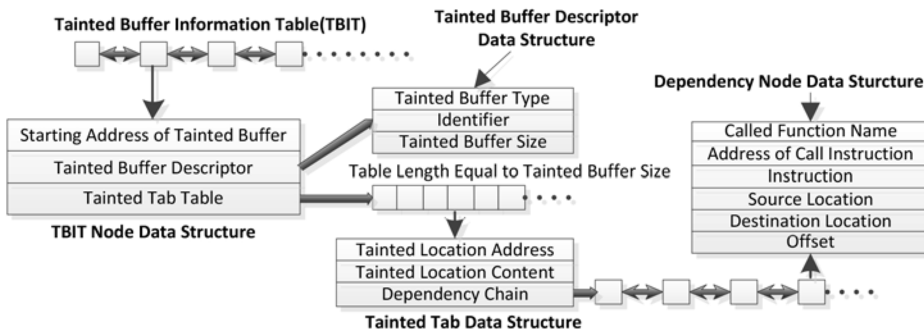


Figure 4: Taint data structure and their relationship

message format. The relationship between the TBIT and the three tainted data structures as shown in Figure 4.

TBIT Node. This structure stores the starting address of tainted buffer, tainted buffer descriptor and tainted tab table. It is primarily used in quick searching for a tainted buffer.

Tainted Buffer Descriptor. This structure stores information about the tainted buffer, which includes the type of this tainted buffer (created by which call-type of library function), the identifier and size of this tainted buffer.

Tainted Tab. There is a one-to-one relationship between the one-byte-long memory unit in cryptographic protocol application and the tainted tab. In a tainted tab, it not only contains the information about the address and the content of this associated memory location, but also stores dependency chain – an important data structure in the proposed system. A dependency chain for a network application is the sequence of reading operations operating on a certain tainted data. A read operation comprises the name of the called function which the read operation instruction belongs to, the address of the function call instruction, the read operation instruction, the source location, the destination location that is written, and the offset of the tainted data with respect to the beginning of the tainted buffer. In our design, the dependency chain of a certain tainted data will grow up after the application executes read operations on it, and *ProtocolFormat* only handles the instructions that simply move the tainted data around but without modifying it. At last, the dependency chain are stopped building when the first read operation for the destination location is: (1) a memory location; (2) an unknown location.

In addition, the taint marking and managing module provides others with three kinds of taint-related operations: the taint-mark buffer operation, the taint-mark data operation and the TBIT querying operation.

Taint-mark buffer operation. When the instruction execution monitor traces that the cryptographic protocol application is executing certain function call instructions related to memory operation, such as *recvfrom()*, *memcpy()*, *EVP_DecryptUpdate_ex()*, it will perform the taint-mark buffer operation on the result buffer of these function calls. The processing of this operation can be divided into three steps: Firstly, we create a TBIT node that associates with the result buffer according to the arguments and the result of the called function. Secondly, we perform the taint-mark data operation on every byte in this taint buffer. Finally, after the TBIT node is created successfully, we will insert it into the TBIT. Moreover, the Tainted Buffer Type in the tainted buffer

descriptor indicates which call-type of library function creates this tainted buffer. In the paper, we use "R" to represent the taint buffer created by network receiving library function, "M" represent that it is created by memory operation library function, and "D" represent that it is created by decryption operation library function.

Taint-mark data operation. Firstly, if the taint data doesn't have an associated tainted tab, the module will create a tainted tab associated with this taint data, and then add the tainted tab to the Tainted Tab Table of TBIT node. Because the size of a tainted buffer is fixed, the Tainted Tab Table in the TBIT node can be constructed by an array of tainted tab data structure. After that, this module will record the run-time execution context on the tainted tab at that moment, which includes the location address, the content and dependency chain of the tainted data. Secondly, if a tainted tab associated with the taint data has already existed, a new dependency chain node will be created, and then be appended to the dependency chain. In *ProtocolFormat*, a dependency chain node includes the information of called function name, address of call instruction, instruction, source operand, destination operand and the offset of this tainted data with respect to the beginning of the tainted buffer.

TBIT querying operation. Given the address of a tainted data, this operation provides other modules with the ability to obtain the tainted data information in TBIT. The querying steps can be summarised as follows: firstly, it is needed to traverse through the linked list node of TBIT to search for which buffer's address range the tainted data belongs to; if the right TBIT node is identified, then the tainted tab table can be queried by computing the offset to the starting address of this buffer. Finally, after reading the tainted tab data structure, other modules can acquire the taint information that associate with this tainted data.

3.2.3 Phase Profiler

According to the assumptions, the process of message parsing in the cryptographic protocol application can be divided into "*packet receiving*" phase, "*message field extraction and processing*" phase, "*encrypted field decryption and processing*" phase, and "*end-of-message parsing*" phase. The reverse analysis system can derive accurate message format only when we know which phase the message parsing phase is in, therefore, it is very important whether the phase profiler can accurately locate the transition point between different message parsing phases.

The proposed approach is based on the observation that network application calls the corresponding library function in different message parsing phase, and the specific process is as follows:

a) When the execution of a network receiving library function call instruction (such as *recefrom*()) is observed, then the message parsing phase will be marked as "*packet receiving*", and the address of the function call instruction will be marked as the transition point to the "packet receiving" phase. Meanwhile, we can use the same approach to mark the "*message field extraction and processing*" phase and the "*encrypted field decryption and process*" phase;

b) As previously mentioned, it is not easy to locate the "*end of message parsing*" phase transition point, but we have overcome the problem by adopting the following methods: Firstly, for single thread application, this goal can be accomplished just by tracing whether another call instruction to the network receive library function is executed. Secondly, for multithread network application, the receiving of network packets and the processing of message are usually implemented in different threads. For this reason, the call of the next network receiving library function doesn't mean the application has finished the message parsing. Therefore, the

first method does not work in this case. Our solution is to observe whether the dependency chain in all tainted tabs have been stopped building, and we assume that the application has finished the message parsing if they are done. As mentioned before, the condition that *ProtocolFormat* stop building the dependency chain is whether the destination location in read operation is a memory location or an unknown location. Usually, it is enough to judge whether the application has finished the processing of a message, because the application commonly will not perform further operation on tainted data after that point.

Most important of all, it is very critical to accurately locate the transition point to the "*end of message parsing*" phase. Because only when the cryptographic protocol application just finishes the message parsing, the tainted information recorded on TBIT is sufficient to present the whole message parsing process; and just at this moment, we can use them to construct the "*message parsing process tree*".

3.2.4 Message Format Constructor

When *ProtocoFormat* identifies that the cryptographic protocol application has finished message parsing process, then the message format constructor will be launched to analyze and process the tainted information that stores in TBIT. In our design, the extraction of message format is divided into three steps: firstly, we need to create a "*message parsing process tree*"; after that, the tree will be traversed to identify message fields in every tainted buffer; Finally, message format constructor infers the semantics of each field, and finishes the message format extraction.

3.2.4.1 Creating Message Parsing Process Tree

Message parsing process tree accurately reveals how a cryptographic protocol application parses an incoming message. The input data used for creating the message parsing process tree derives from the TBIT. The details of construction of the message parsing process tree are described in Algorithm 1.

Building message parsing process tree and using it to store the identified fields are the main tasks of *ProtocolFormat*, each node of this tree represents a field in the message. Essentially, Algorithm 1 needs to iterate through the TBIT and searches for a TBIT node whose "Taint Buffer Type" is "R" (line 5, this tainted buffer is created by a network receiving library function such as *recefrom*()). Once found, the ROOT node of the "*message parsing process tree*" will be created and initiated according to the information that stores in the TBIT node. As shown in Figure 5, the data structure of tree node is similar to the tainted tab table in TBIT node, the number of items in a tree node equals the size of the relevant tainted buffer.

After creating the ROOT node, we will read the tainted tab of each item in it (lines 11-12). If the call-type of the last function call in the dependency chain of this taint tab is "M" (created by memory operation library function), then a new plaintext message field tree node will be created according to the starting address and the size of the destination buffer created by this last memory operation (lines 13-14). This information can be obtained by the call arguments stored in the dependency chain. Similarly, if the call-type of the last function call in the dependency chain is "D" (decryption library function call), a decryption field tree node will be created and inserted as a child node to the ROOT node. However, if the call-type of the last function call is "O" (other function call), we do nothing but set the child node of this tree node as NULL (lines 20-23, assign "Linked Tree Node" of this item to NULL).

```
1: Input: the tainted buffer information table - TBIT
2: Result: ProcessTree – the message-processing tree
3: Message_ProcessTree_Creation ( TBIT ) {
4:        ProcessTree ← ROOT;                                /* Create the ROOT node */
5:        TBITNode ← Search_RecvBuf ( TBIT );               /* Search for a TBIT node of which the "Taint Buffer Type" is "R" */
6:        Initialization ( ROOT, TBITNode );                 /* Initialize the ROOT node with the information stored in TBITNode */
7:        Recursion_Process ( ROOT );
8:        Return ProcessTree;
9: }
10: Recursion_Process ( TreeNode ){
11:        for each I ∈ TreeNode.Item {                       /* Process each item in this tree node */
12:               CallType ← Read_Last_CallType ( I );        /* Read the call-type of the last function call in item */
13:               if ( CallType == "M" or   CallType == "D" ){
/* If CallType is "M", NewTreeNode is a memory operation result tree node; and if "D", NewTreeNode is a decryption result tree node */
14:                      Create a new message field tree node NewTreeNode;
                         /* Search for a TBIT node of which the start address is NewTreeNode.ID */
15:                      TBITNode ← SearchBufByAddr ( NewTreeNode.ID );
16:                      Initialization ( NewTreeNode, TBITNode );
17:                      I.LinkField ← NewTreeNode.ID;          /* Insert NewTreeNode as the child of TreeNode */
                         /* Process items in NewTreeNode recursively, until the child node of all tree nodes are NULL */
18:                      Recursion_Process ( NewTreeNode );
19:               }
20:               if ( CallType == "O" ){
21:                         I.LinkField ← NULL;                /* Insert NULL as the child of TreeNode */
22:                      continue; }
23:        }
24:        return;
25: }
```

Algorithm 1: Message Processing Tree Creation

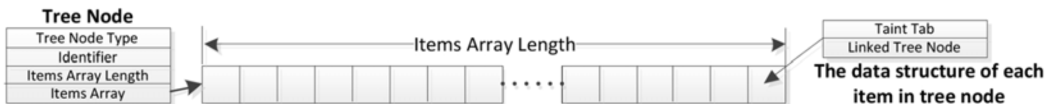


Figure 5: Message Field Tree Node Structure

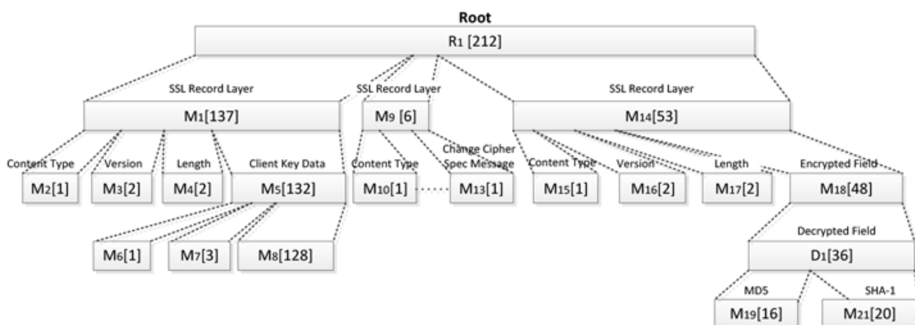


Figure 6: Message Parsing Process Tree for a SSL Handshake Message

The procedure that starts from line 10 to line 25 will be executed recursively until the child nodes of all new created tree nodes are NULL. At that moment, we stop building the message parsing process tree. For illustrating the parsing process more specifically, as shown in Figure 6, we present a message parsing process tree derived from a SSL handshake protocol message, which propagates from client browser to https server during the fourth negotiation phrase.

3.2.4.2 Identifying Message Fields

After construction of message parsing process tree, we need to identify the message fields that exist in every message tree node (in fact, it represents a tainted buffer), the detailed process is described in Algorithm 2.

```
1: Input: ProcessTree - the message-processing tree
2: Result: Identifying the message fields in all message tree node
3: Message_Field_identification ( ProcessTree ) {
4:        TreeNode ← Get_RootNode ( ProcessTree );           /* Get the root tree node of the message processing tree */
5:        Depth-First_Traverse_ProcessTree ( TreeNode ) {
6:               For each Item in TreeNode {
7:                      Compare the dependency chain of this item with other's in the same TreeNode;
8:                      Mark those items with the same dependency chain as a message field;
9:                      CallType ← Read_Last_CallType ( Item );     /* Read the call-type of the last function call in item */
10:                     if ( CallType == "M" ) Mark this message field as a plaint-txt field;
11:                     if ( CallType == "D" ) Mark this message field as an encrypted field;
12:              }
13:              For each TreeNode's child Child[i] {
14:                     Depth-First_Traverse_ProcessTree ( Child[i] ); }
15:       }
16: }
```

Algorithm 2: Message Field Identification

Figure 7: The Dividing Result of "R1" After First Step

Specifically, Algorithm 2 adopts the depth-first search strategy to traverse each node of the message-processing tree. The first step is to obtain the root tree node "R_1". After that, we compare the dependency chain of each item in the root node, and divide ones that have the similar dependency chain into a message field. Take an example as shown in Figure 7; the first step (lines 3-8) will divide the root node "R_1" into two message fields "$M1_1$"and "$M1_2$".

The next step is to infer the field attributes of each message field starting from the lowest address of this tree node (lines 9-11) and iterate these steps until all message fields have been identified (lines 13-14). Currently, our prototype system just handles two types of field attributes: plaintext field and encrypted field. After the algorithm terminates, we derive a hierarchical message format of this example, as shown in Figure 8. On the other hand, the reverse result can also be presented as:

$$\mathbf{R_1: \quad M1_1, \{M2, M3, M4\}K}$$

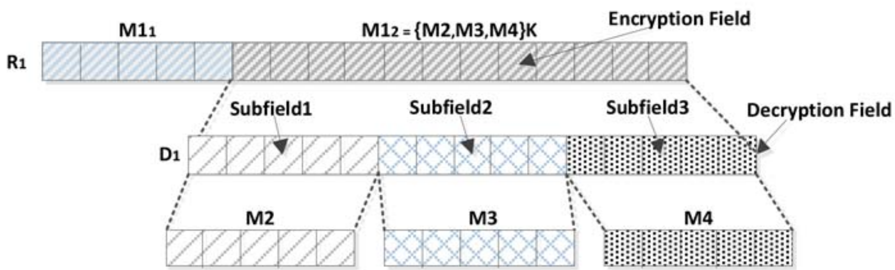


Figure 8: The Message Format of Example Message

3.2.4.3 Message Fields Semantics Inference

After identifying message fields for the message parsing process tree, we can use it to represent the hierarchical structure of the message; however, it does not contain semantic information for each message field, such as whether it represents the length of another message field, whether it is a RSA encrypted field or a hash field. However, in our future work, whether we can successfully reverse extract the formal protocol specification from a security application depends on the semantics of every message field. For this reason, in this section we firstly introduce the approach to identify the field semantics in messages.

Function Name	Field Semantics	Input Buffer	Output Buffer	Call-type
EVP_DigestUpdate	Message Digest Input Field	Second Parameter	NULL	D
EVP_DigestFinal_ex	Message Digest Field	NULL	Second Parameter	D
EVP_DecryptUpdate	Input Buffer is Message Encryption Field Output Buffer is Message Decryption Field	Fourth Parameter	Second Parameter	D
EVP_DecryptFinal_ex	Message Decryption Field	NULL	Second Parameter	D
memcpy	Output Buffer is Memory Write Field	Second Parameter	First Parameter	M
recvfrom	Network Received Field	NULL	Second Parameter	R

Table 3: API Function Prototype Handled by ProtocolFormat

Similar to previous work (Caballero *et al*, 2007; Gilbert *et al*, 2008; Juan *et al*, 2009), our approach is based on the insight that many library functions and instructions used in applications contain rich semantic information. As a result, the semantics of message fields can be inferred by observing whether the tainted data is taken as input parameters of the library function calls and the instructions whose semantics are known. Then the tainted data will be associated with the semantics of the corresponding parameters, which have been defined in the function prototypes.

As mentioned in instruction execution monitor, the call-type of function can be obtained by querying the standard library function information database. In fact, the semantic information of

well-known standard library functions is stored in the *Standard Library Function Information Database*, therefore, message field semantics can be derived by querying this database. Although the documents of these functions or instructions commonly are public, such as the *Microsoft Developer Network* (*MSDN*) or the standard C library, which describe the semantics of the parameter and the goal of these library functions, we need to manually convert them into the formal function prototype that *PotocolFormat* can handle. Table 3 shows parts of the library function prototypes stored in the *Standard Library Function Information Database*, the semantics of their input buffer and output buffer have been defined in the prototype.

4. Implementation and Evaluation

4.1 Implementation

Based on the latest *Valgrind* (version 3.6.1), we have implemented a prototype system of *Protocol-Format* as a tool of the *Valgrind* platform. For the instruction execution monitor in the proposed system, we use *Callgrind* that is a profiling tool under the *Valgrind* platform as the main framework of this module. Meanwhile, a great many of binary analysis techniques supported by *Valgrind* platform can be utilized during the implementation of this module, these techniques include *instruction translation*, *memory marking*, *taint propagation*, and others. Because of limited time, we do not adopt mature database techniques, such as *MYSQL* or *Oracle*, to build the *Standard Library Function Information Database* in the execution monitor, but merely store the standard library function prototype in a text file. Consequently, once *ProtocolFormat* starts it will read this text file first, and then store the information in an object based on array structure.

4.2 Evaluation

In order to evaluate the actual effect of *ProtocolFormat* researched in this paper, we will perform two sets of experiments on *ProtocolFormat*. The first set of experiments involves seven protocol messages from four known cryptographic protocol applications. The second set of experiments involves one protocol message in an unknown cryptographic protocol used by a video communication application.

The experiment is as the following steps: Firstly, a cryptographic protocol application that can parse a cryptographic protocol message is launched in *ProtocolFormat* simulation environment. Then, after the application finishes parsing an incoming message, our system will output a result table about the format of this message. Therefore, we can evaluate *ProtocolFormat* by comparing the result with the protocol specification adopted by this application.

4.2.1 Evaluation on Known Protocols

In this section, the proposed system is evaluated on two well-known cryptographic protocols: SSL Handshake Protocol and FTP SSL. After *ProtocolFormat* outputs all the message formats, they will be compared with the output results of *Wireshark* 1.6.0. For each cryptographic protocol, we will choose a typical application that implements this protocol.

For conveniently comparing with the *Wireshark*, we define two terminologies that relate to the evaluation indicators of the experiments: *leaf field* and *hierarchical field*. A *leaf field* means that this message field cannot be further divided into any subfields. *Hierarchical field* indicates that this message field consists of several subfields. We respectively represent the sets of *leaf fields* and *hierarchical fields* as L and H, and count $|L_W|$, $|H_W|$ and $|L_P|$, $|H_P|$ in the results of *Wireshark* and *ProtocolFormat*. Then, after comparing with standard protocol specification, we use $|E(L_P)|$ and

$|E(H_P)|$ respectively represent the number of errors on the *leaf fields* and the *hierarchical fields* of message format result produced by *ProtocolFormat*. In addition, since *ProtocolFormat* may overly divide a correct field into several sub-fields, we count the total number of these overly divided fields as $|O_P|$.

In this experiment, for each protocol we select a typical application that adopts the protocol as part of their communication. For SSL Handshake protocol, we evaluate *ProtocolFormat* with a HTTPS server and a client browser, and compare the reverse derived message format with *wireshark*. More specifically, we chose *Apache* as the HTTPS sever, run it under the environment of *ProtocolFormat*, observe the execution of it, and then run *Firefox*, a browser client, from another physical machine to establish a SSL connection. The reverse analysis continues until the SSL handshake phrase ends.

		Wireshark		**ProtocolFormat**				
Protocol	**Message Type**	$\|L_W\|$	$\|H_W\|$	$\|L_P\|$	$\|H_P\|$	$\|E(L_P)\|$	$\|E(H_p)\|$	$\|O_p\|$
SSL Handshake Protocol	Client Hello	31	7	41	9	1	0	0
	Server Hello	98	76	29	9	0	0	0
	Client Key Exchange	13	4	17	6	0	0	0
	Change Cipher Spec	8	2	11	5	1	0	1
FTP SSL	Welcome	2	1	3	1	1	0	0
	Auth SSL Request	2	1	2	0	0	1	0
	Server Response	2	1	2	0	0	1	0

Table 4: Message Format Comparison between Wireshark and ProtocolFormat

For FTP SSL, we select *vsftpd* 2.3.4 for FTP SSL server and *Filezilla* 3.5 for FTP client. Similar to the experiment of SSL handshake protocol, FTP SSL server and FTP client respectively run at different physical machines under *ProtocolFormat*. We firstly control the FTP client to send a AUTH SSL request command message to the FTP SSL server, then after the SSL connection is established, we download a file from the sever and close the session. The results are shown in **Table 4**.

Result analysis. The results of SSL handshake protocol and FTP SSL show that *ProtocolFormat* can reversely extract message format satisfactorily, and can even outperform *Wireshark* when security applications use lots of library function to operate on received data, such as identifying *leaf fields* in messages of "*Client Hello*", "*Client Key Exchange*" and "*Change Cipher Spec*". However, the results also show that when network applications do not use standard library functions to parse messages, *ProtocolFormat* will not output accurate message format results. In such case, *Wireshark* will perform better than *ProtocolFormat*, as shown in the result of "*Server Hello*" in **Table 4**.

4.2.2 Evaluation on Unknown Cryptographic Protocol

In this experiment, we evaluate the proposed system on an application that adopts an unknown cryptographic protocol to hide its communication. We choose a video communication application as the experimental application that needs to authenticate each communication entity. The minimum system is composed of a server and two user side applications. We only run a client side program of this communication application in *ProtocolFormat* environment, and separately run another client side and server program on two different virtual machines.

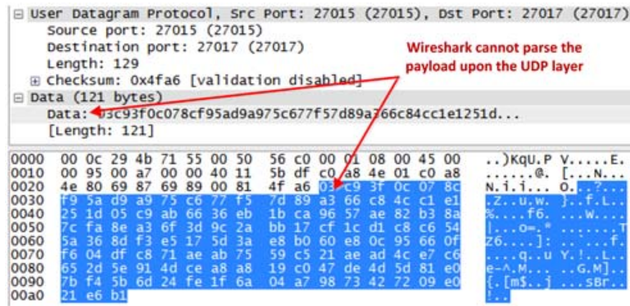


Figure 9: Wireshark Resolving Result

During the experiment, we firstly capture a packet from the server to the user, and parse it by *Wireshark*, the packet parsing results are shown in Figure 9. According to Figure 9, the *Wireshark* cannot decode the application data upon transport layer. We think there are mainly two reasons for this: on the one hand, maybe the authentication module of this cryptographic protocol application program is developed based on a cryptographic protocol specification, however, cryptographic protocol specification commonly does not strictly fix encryption algorithm, length of each message field, and others. As a result, different implementations of a same cryptographic protocol usually have different message formats. Consequently, for the protocol analysis techniques based on network flow, such as *OmniPeek* and *Wireshark*, they cannot parse this kind of encrypted messages for lack of knowledge about how to decrypt it.

After the user side program accomplishes a message parsing process, *ProtocolFormat* outputs the message format result as shown in Figure 10.

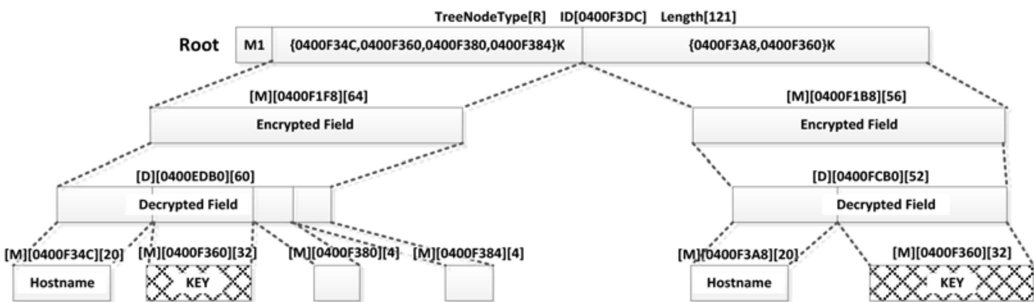


Figure 10: The Message-processing Tree for an Unknown Protocol

In Figure 10, root node shows that this message (application layer data in network packet) consists of three message fields, they are length field "M1", encrypted field "0400F1F8" and encrypted field "0400F1B8". The decrypted field of the encrypted field "0400F1F8" is "0400EDB0", and "0400EDB0" can be further divided into hostname field "0400F34C", key field "0400F360", and two general fields "0400F380", "0400F384". Similarly, the decrypted field of the encrypted field "0400F1B8" is "0400FCB0", and "0400FCB0" can be further divided into hostname field "0400F3A8" and key field "0400F360".

According to the hierarchical structure of the message format, we can further describe the message format as:

M1, {0400F34C,0400F360,0400F380,0400F384}K, {0400F3A8,0400F360}K

Moreover, we find there is an intersecting tree node "0400F360", of which the background colour is black in Figure 10, between the two decrypted fields. Usually, it means that the two decrypted fields have a common field.

5. Related Work

In this section, we present the related work and compare it with *ProtocolFormat*. Note that the instruction execution monitor relies heavily on the dynamic taint analysis technique. Moreover, since this technique has been widely investigated in recent years, there exist abundant related works for taint analysis (Chow *et al*, 2004; Crandall *et al*, 2006; Manuel *et al*, 2007; Newsome and Song, 2005; Suh *et al*, 2004; Philipp *et al*, 2007; Heng *et al*, 2007). Therefore, the detailed description is omitted in this section.

As mentioned before, protocol reverse engineering has recently received significant attention for its importance to analyzing network security applications. The Protocol Informatics (PI) project and *Discoverer* are network-based reverse approaches; therefore, they extract the message format from collected network traces. For just only requiring collection of network flow, these kinds of approaches have advantages when a protocol parsing application is not available, However, they become less effective in face of the encrypted network flow.

Unlike the *PI* and *Discoverer* projects, several reverse systems such as *Polyglot* (Caballero *et al*, 2007), *AutoFormat* (Zhiqiang *et al*, 2008), *Tupni* (Cui *et al*, 2008), *Prospex* (Comparetti *et al*, 2009), *Dispather* (Caballero *et al*, 2009; Juan *et al*, 2009), and *ReFormat* (Zhi *et al*, 2009) which from host perspective, share the key insight that how a protocol application recognizes and parses protocol messages provides valuable information about message format. Based on the insight, *Autoformat* collects and analyzes run-time execution context information to infer message format by recognizing and leveraging field specific execution context. In addition, *Prospex* makes further progress to uncover protocol specification. However, all these system are mainly designed to work with plain-text input message except *ReFormat*. Therefore, they all become ineffective when analyzing encrypted protocol messages.

In the current existing protocol reverse engineering approaches, *ReFormat* can extract the message format when the message is encrypted. However, there are also many limitations, for example, when the processing of a message involves significant arithmetic and bitwise operations, *Reformat* may not work properly. Furthermore, *ReFormat* assumes an application first decrypts an encrypted message and then processes the decrypted message, thus, if an application decrypts part of the message and processes it before decrypting and processing the rest, *ReFormat* will stop working because it can't identify the whole decrypted message correctly. That may be particularly troublesome because a cryptographic protocol message commonly contains several encrypted message fields and plain-text fields, so *ReFormat* cannot extract message format from this kind of message.

Comparing to *ReFormat*, *ProtocolFormat* relies on another technique to locate the decrypted memory buffers: identification of calls to standard decrypted function. Similar to other program-based system, *ProtocolFormat* relies on dynamic taint analysis technique, this technique has been proposed in Newsome and Song (2005).

6. Limitations and Future Work

In this section, we discuss about the limitations of *ProtocolFormat* and suggest possible improvements for future work.

ProtocolFormat relies heavily on the knowledge of relevant standard library function. For example, according to the semantics of the network receive library function *recvfrom()*, the memory copy library function *memcpy()*, and the decryption library function *EVP_DecryptUpdate()*, the message parsing process can be divided into a packet receiving phase, a message field extraction and parsing phase, a message field decryption and parsing phase, and an end-of-message parsing phase. Accurate identification of these message-processing phases strongly determines whether *ProtocolFormat* can reverse derive the message format correctly. To take another example, *ProtocolFormat* relies strongly upon the semantics of function arguments because *ProtocolFormat* can create the tainted buffer or the "message processing tree" correctly only when it obtains the accurate semantics and values of the function arguments. Consequently, for network applications that implement decryption or memory duplication using their own binary program, *ProtocolFormat* may not work properly. One possible way to solve these problems is to discover other characteristics used for identifying different message parsing phases.

ProtocolFormat is mainly used for analysis of benign applications. Therefore, in the proposed design, how to reversely analyze programs that adopt code obfuscation techniques is not considered. In other words, if the network application intentionally introduces abundant redundant instructions, e.g., by embedding unnecessary memory-related library function call instructions that take the receive buffer as a source argument, then the analysis of *ProtocolFormat* can be potentially evaded. Making *ProtocolFormat* applicable to reverse analyze obfuscated applications is still a technical challenge now.

ProtocolFormat identifies whether a network application is extracting a message field mainly by observing calls to memory-related library functions. However, if the application uses "zero-copy technique", it will not call the memory copy library function to extract message fields. Therefore, even though the application has extracted a message field, the dependency chain of these tainted data may be unchanged. In this case, *ProtocolFormat* cannot divide the message field according to the calling sequence of each tainted data.

As for the granularity of message reverse analysis, the current version of *ProtocolFormat* can analyze the message format only at byte granularity; how to analyze the protocol field at the bit level is being considered. Subsequent proposed work includes re-implementing tainted marking and propagation at the bit level so that future versions of *ProtocolFormat* can analyze protocol fields of less than one byte.

ProtocolFormat can only reverse analyze each message independently and cannot correlate multiple messages in the same protocol session. However, the final goal of this research is to reverse reconstruct the cryptographic protocol specification from the implementation of the cryptographic protocol. Therefore, improving the proposed approach to reverse analyze the sequential logic from multiple messages in a single protocol session and to reconstruct the entire protocol state machine is part of the authors' proposed future work.

7. Conclusion

In this paper, *ProtocolFormat* is proposed, which is a system used for reverse extracting the message format from a cryptographic protocol applications program. The design of *ProtocolFormat* is

mainly based on the assumption that the parsing of incoming messages in cryptographic protocol applications is entirely based on calls to standard library functions. This paper first introduces the entire system workflow, the critical data structures, and the structure, principle, and functions of each module in detail, and then conducts an evaluation of *ProtocolFormat* by performing two sets of experimental tests. The experimental results show that *ProtocolFormat* can accurately analyze messages from cryptographic protocol applications which are developed using standard library functions. Of course, the proposed system also contains many defects and problems, and therefore many improvements will subsequently be required. These are: (1) finding more characteristics that can be used to identify various message parsing phases; (2) further refining the granularity of reverse analysis so that future versions of *ProtocolFormat* can analyze a bit-level protocol field, thereby achieving the purpose of improved analytical accuracy; and (3) improving the proposed approach to analyze and correlate multiple messages in a single protocol session, making it possible to reconstruct a cryptographic protocol specification from a cryptographic protocol application.

Acknowledgements

The authors are grateful to the reviewers for their insightful comments that helped improve the presentation of this paper. Part of this work was supported by the National 863 High Tech Programs No. 2011AA01A103, and the National Natural Science Foundation of China under Grants No. 60873215 and No. 61003303.

References

CABALLERO, J., HENG, Y. and ZHENKAI, L. (2007): Polyglot: Automatic Extraction of Protocol Format Using Dynamic Binary Analysis, *Proc. the 14th ACM Conference on Computer and Communications Security*, Alexandria, USA.

CABALLERO, J., POOSANKAM, P. and KREIBICH, C. (2009): Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering, *Proc. CCS'09*, Chicago, Illinois, USA, 621–634, ACM Press.

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K. and ROSENBLUM, M. (2004): Understanding Data Lifetime via Whole System Simulation, *Proc. the 13th USENIX Security Symposium*, San Diego, CA, 22–38, USENIX Association Press.

COMPARETTI, P.M., WONDRACEK, G., KRUEGEL, C. and KIRDA, E. (2009): Prospex: Protocol Specification Extraction, *Proc. 30th IEEE Symposium on Security and Privacy*, Oakland, CA, 110–125, Ieee Press.

CRANDALL, J.R., WU, S.F. and CHONG, F.T. (2006): Minos: Architectural support for protecting control data, *Transactions on Architecture and Code Optimization*, 3(4): 359–389.

CUI, W., PEINADO, M. and CHEN, K. (2008): Tupni: Automatic Reverse Engineering of Input Formats, *Proc. ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 391–402, ACM Press.

DAVID, W. and BRUCE, S. (1996): Analysis of the SSL 3.0 protocol, *Proc. the Second USENIX Workshop on Electronic Commerce*, Oakland, California, 29–40, USENIX Association Press.

ERIC, R. (2001): SSL and TLS: Designing and Building Secure Systems, Addison-Wesley Professional.

GILBERT, W., PAOLO MILANI, C. and CHRISTOPHER, K. (2008): Automatic Network Protocol Analysis, *Proc. the 15th Annual Network and Distributed System Security Symposium*, San Diego, California, USA, 1–18, Citeseer Press.

HENG, Y., DAWN, S., EGELE, M., KRUEGEL, C. and KIRDA, E. (2007): Panorama: Capturing system-wide information flow for malware detection and analysis, *Proc. the 14th ACM Conferences on Computer and Communication Security (CCS' 07)*, October.

ISO/IEC 9899:1999 (1999): C programming language standard (online), available: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

JUAN, C., PONGSIN, P., CHRISTIAN, K. and DAWN, S. (2009): Bidirectional Protocol Reverse Engineering: Message Format Extraction and Field Semantics Inference, UC-Berkeley: EECS.

MANUEL, E., CHRISTOPHER, K. and ENGIN, K. (2007): Dynamic spyware analysis, *Proc. the 2007 USENIX Annual Technical Conference (Usenix' 07)*, June.

MARSHALL, B. (2004): 'The Protocol Informatics', (online), available: http://www.4tphi.net/~awalters/PI/PI.html.

MICROSOFT DEVELOPER NETWORK (2011): (online), available: http://msdn.microsoft.com.

NETHERCOTE, N. (2004): Dynamic Binary Analysis and Instrumentation, University of Cambridge.

NETHERCOTE, N. and SEWARD, J. (2007): Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, USA, 42: 89–100, ACM Press.

NEWSOME, J. and SONG, D. (2005): Dynamic Taint Analysis for Automatic Detection, Analysis,and Signature Generation of Exploits on Commodity Software, *Proc. the 14th Annual Network and Distributed System Security Symposium (NDSS 2005)*, San Diego, CA, Citeseer Press.

PHILIPP, V., FLORIAN, N., NENAD, J., ENGIN, K., CHRISTOPHER, K. and GIOVANNI, V. (2007): Cross-site scripting prevention with dynamic data tainting and static analysis, *Proc. the 14th Annual Network and Distributed System Security Symposium (NDSS' 07)*, San Diego, CA, Press, February.

SHAIKH, S. and BUSH, V. (2005): Analysing the Woo-Lam Protocol Using CSP and Rank Functions, *Journal of Research and Practice in Information Technology*, 38(2): 19–29.

SUH, G.E., LEE, J. and DEVADAS, S. (2004): Secure program execution via dynamic information flow tracking, *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS' 04)*, Boston, Massachusetts.

VALGRIND (2011): valgrind-3.7.0 [online], available: http://valgrind.org/.

WEIDONG, C., KANNAN, J. and WANG, H.J. (2007): Discoverer: Automatic Protocol Reverse Engineering from Network Traces, *Proc. the 16th USENIX Security Symposium*, Bostonm, MA, 199-212, USENIX Association Press.

WIRESHARK, (2011): [online], available: http://www.wireshark.org.

WOO, T.Y.C. and LAM, S.S. (1992): Authentication for distributed systems, *Computer*, 25(1): 39–52.

ZHI, W., XUXIAN, J., WEIDONG, C., XINYUAN, W. and GRACE, M. (2009): ReFormat: Automatic Reverse Engineering of Encrypted Messages, *Proc. the 14th European conference on Research in Computer Security (ESORICS'09)*, 200–215, Springer-Verlag Press.

ZHIQIANG, L., XUXIAN, J., DONGYAN, X. and XIANGYU, Z. (2008): Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution, *Proc. the 15th Annual Network and Distributed System Security Symposium (NDSS 2008)*, San Diego, California, USA, Press.

Biographical Notes

Meijian Li

***Meijian Li** received his MSc in traffic information and control engineering from the Military Transportation University, China in 2009. Then he received his PhD in network and information security from the National University of Defense Technology, China, in 2014. Since then, he has worked with the Chinese Air Force as an engineer. His research interests include network securty, network protocol reverse engineering and web development.*

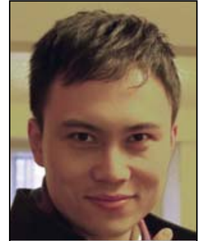
Yongjun Wang

***Professor Yongjun Wang** received his MSc and PhD in computer science from the National University of Defense Technology, China, in 1995 and 1998 respectively. Since then, he has worked at the School of Computers, National University of Defense Technology as a lecturer, associate professor and full professor in 1998, 2000 and 2006 respectively. His current research interests are network security, system security and high performance network.*

Shangjie Jin *received his BSc in vehicle engineering, and his MSc in power machinery and engineering from the Military Transportation University, China, in 2005 and 2008 respectively. He is currently an engineer at this same university. His research interests include network protocol reverse engineering and security protocol formal analysis.*

Shangjie Jin

Shanshan Liu *received his BSc from the PLA Information Engineering University, China, in 2005. Since then, he has gained a masters degree and graduated from the Military Transportation University, China, where he is currently an assistant professor. His research interests are information security and automatization.*



Shanshan Liu

Peidai Xie *is a PhD student at the School of Computer, National University of Defense Technology, China. He is interested in all aspects of computer network and system, including malware, binary analysis, network protocol reverse engineering, etc.*

Pedai Xie

Zhen Huang *received his bachelor degree in computer science and technology from the National University of Defense Technology, China in 2006. From September 2006 to March 2008, he pursued the masters degree in computer science and technology at the same university. He was recommended to pursue a PhD degree in advance since March 2008, and received his PhD in 2012. His research interests are distributed storage systems including peer-to-peer backup systems and cloud storage systems.*

Zhen Huang