Icefex: Protocol Format Extraction from IL-based Concolic Execution

Fan Pan[1], Li-fa Wu[1], Zheng Hong[1], Hua-bo Li[1], Hai-Guang Lai[1] and Chen-hui Zheng[1]
[1]Institute of Command Automation, PLA University of Science and Technology
Nanjing, Jiangsu - China
[e-mail: dynamozhao@163.com, wulifa@vip.163.com, hongzhengjs@139.com, huabolee@163.com, laihaiguang@gmail.com, chh.zheng@163.com]
*Corresponding author: Zheng Hong

Received November 22, 2012; revised January 27, 2013; revised March 3, 2013; accepted March 9, 2013; published March 29, 2013

Abstract

Protocol reverse engineering is useful for many security applications, including intelligent fuzzing, intrusion detection and fingerprint generation. Since manual reverse engineering is a time-consuming and tedious process, a number of automatic techniques have been proposed. However, the accuracy of these techniques is limited due to the complexity of binary instructions, and the derived formats have missed constraints that are critical for security applications. In this paper, we propose a new approach for protocol format extraction. Our approach reasons about only the evaluation behavior of a program on the input message from concolic execution, and enables field identification and constraint inference with high accuracy. Moreover, it performs binary analysis with low complexity by reducing modern instruction sets to BIL, a small, well-specified and architecture-independent language. We have implemented our approach into a system called *Icefex* and evaluated it over real-world implementations of DNS, eDonkey, FTP, HTTP and McAfee ePO protocols. Experimental results show that our approach is more accurate and effective at extracting protocol formats than other approaches.

Keywords: protocol reverse engineering, protocol format extraction, semantic inference, concolic execution, intermediate language

This work was supported by the Nature Science Foundation of Jiangsu China under Grant No.BK2011115; The opening Foundation of Laboratory of Military Network Technology. The authors would like to thank the anonymous reivewers for their valuable comments which improved the quality of this paper.

http://dx.doi.org/10.3837/tiis.2013.03.010

1. Introduction

Knowledge of application-level protocol format is useful for many network security applications, including intelligent fuzzing [1][2], intrusion detection [3][4] and fingerprint generation [5]. However, many protocols in use are closed protocols without publicly available specification. Even for open protocols, certain implementations may not exactly follow the specification. Thus protocol reverse engineering, the process of extracting the application-level protocol used by an implementation, is valuable for the above network security applications.

Currently, protocol reverse engineering is mostly a manual, tedious and time-consuming task. For example, the MSN messager protocol has been persistently reverse engineered, since the open source clients [6] regularly require patching to support proprietary changes in the protocol. To reverse engineer a protocol in a timely manner and keep up the effort through time, a number of automatic solutions to protocol reverse engineering have been proposed. These solutions can be classified into *network-trace based techniques* [7][8][9][10] and *execution-trace based techniques* [2][11][12][13][14][15]. Network-trace based techniques analyze network traffic by recording the communicaiton between a client and a server, and then extract protocol format through message clustering. Although useful in practice, their accuracy is often limited by the diversity of messages in the trace. In contrast, execution-trace based techniques operate with higher accuracy by observing the execution of the application while processing input messages. Furthermore, they provide insight into field semantics that are not available to network-trace based techniques.

The effectiveness of execution-trace based techniques have been proven [11]. However, these techniques still have two major limitations: 1) Common application behaviors, such as bulk accesses and optimized string processing, give rise to redundancies and inconsistencies in the results of field identificaiton. 2) They could not generate constraints on the values of fields, since all of them rely on dynamic taint analysis. Such constraints that a message must satisfy to be valid are critical for security applications.

Our work was motivated by these limitations. In this paper, we propose a new approach for protocol format extraction. Given the program binary and input message, our approach captures the program execution traces, and then reason about the behavior of a program on the input message from IL-based concolic execution. Based on the *concolic execution state* that maps variables and memory addresses to both concrete values and symbolic expressions, we extract protocol format according to the semantics of executed instructions. To realize our approach, We have designed and implemented a system called *Icefex*. We use five real-world protocols to compare the message format automatically extracted by Icefex with the one extracted by AutoFormat and Tupni, two of state-of-the-art approaches for format extraction. The results demonstrate the proposed approach is superior to previous techniques.

In summary, the contributions of our paper are the following:

- We propose a new approach for protocol format execution that perform analysis with low complexity by lifting assembly instructions to BIL [16], a small and well-specified language. This is in contrast to all the current approaches that process hundreds of assembly instructions with intricate and non-intuitive semantics directly.
- We adopt concolic execution, a more insightful approach for data-flow analysis, to track the behavior of a program on the input message. Previous work for extracting

protocol format relies on dynamic taint analysis and reports only the information about which byte of the input data is used. By comparison, our approach performs concolic execution to offer additional information about how the input data is used, which is essential for constraint inference.

- We present new techniques for identifying field boundaries. Protocol implementations usually process input messages in stages, such as lexing, parsing and evaluation. Exsiting techniques identify the consecutive bytes that used in the operand of assembly instruction as a field. Since the instructions in lexing and parsing stages read the input bytes without concern for field boundaries, there are many redundancies and inconsistencies in the results of field identification. In contrast, Our apparoach focuses on the instructions in evaluation stage and is generally more accurate.
- We propose what we believe are the first techniques to infer field constraints. Current approaches have described how to identify field semantics, but none of them have discussed how to infer field constraints, which are critical for message replaying. Our approach enable constraint inference by extracting symbolic predicates from evaluation points of concolic execution, since constraints always exist on the fields with static semantics.

The paper is organized as follows. Section 2 defines the goals of our approach and articulates the main challenges. In Section 3, we describe the approach and system architecture. Then we introduce the details of concolic execution in Section 4 and present the policies for protocol format extraction in Section 5. We evaluate our system in Section 6, Section 7 draws conclusion and summarizes future work.

2. Goals and Challenges

This paper aims to automatically extract protocol format by analyzing the binary execution traces of protocol implementations. The protocol format we seek to extract includes field boundaries, field semantics and constraints on the values of fields. To achieve the goal, we face three major chanllenges:

1) Not all instructions read the message bytes in accordance with field boundaries. From a linguistic perspective, the message processing of protocol implementation can be divided into three stages: lexing, parsing and evaluation [1]. The binary instructions in lexing and parsing stages, such as bulk accesses and optimized string processing, may read the input bytes without concern for field boundaries. However, current techniques[11][12][13][14][15] monitor all the operations done by a program using dynamic taint analysis, and identify the consecutive input bytes that used in the operand of each binary instruction as a field. It has been verified that redundacies and inconsistencies are inevitable in the field identification results of these techniques. For example, **Table 1** shows the field identification at each step of the execution for a fragment of assembly instructions.

Table 1. Field identification at each step of the execution for example program

Line	Instruction	Taint Record	Identification
1	`mov` esi, [esp+019Ch]	esi $\rightarrow \{0,1,2,3\}$	
2	`and` esi, 0FFh	esi $\rightarrow \{0,1,2,3\}$	$\langle 0,4 \rangle$
3	`lea` eax, [esi-1]	esi, eax $\rightarrow \{0,1,2,3\}$	$\langle 0,4 \rangle$
4	`cmp` eax, 0E2h	esi, eax $\rightarrow \{0,1,2,3\}$	$\langle 0,4 \rangle$
5	`jz` loc_527169	esi, eax $\rightarrow \{0,1,2,3\}$	

The code fragment in **Table 1** is acquired from Emule 0.48a, a real-world program for P2P communication.The beginning of this fragment loads the first four bytes of the input message into register `esi`. For all input bytes are marked as tainted, `esi` is marked by a set of symbol tags. On line 3, `esi` (tainted) is subtracted to `eax` (untainted). Since all the operations on line 2, 3 and 4 use the consecutive bytes {0,1,2,3} in the operand, current techniques will identify the 2-tuple $\langle 0,4 \rangle$ as a field repeatedly, where the first element denotes field offset and the second element denotes field length. Even worse, the real effect of the instruction fragment is to make a conditional jump depending on the first input byte, and $\langle 0,4 \rangle$ will conflict with the field $\langle 1,4 \rangle$ which is identified in the subsequent execution.

In order to remove the redundancy in the results of field identification, AutoFormat [12] builds a protocol field tree to store the identified fields and merges the internal nodes which have only one child. Tupni [14] uses greedy algorithm to find a consistent subset F of identified fields such that all fields in F are disjoint and the combined access weight is maximized. However, both of them only provide partial solution and do not guaratee the accuracy of field identification.

2) More detailed information is needed to infer field constraints. Many protocol format extraction approaches have described how to identify field semantics, but none of them have discussed how to infer the constraints on the values of fields. In addition to field boundaries and semantics, we also aim to infer field constraints. Note that current techniques only perform dynamic taint analysis to monitor the program as it processes an input, and provide poor information about the conditions on the values of fields. For example, we can identify $\langle 0,4 \rangle$ as a field with `keyword` semantics according to line 4 in **Table 1**, but we are unaware that the value of the field must be 0E3h to proceed the following execution.

It is natural to employ symbolic execution to reason about constraints on input bytes. However, symbolic execution of large programs is bound to be imprecise, since program instructions such as pointer manipulations and arithmetic operations are complex and calls to operating-system and library functions are impossible to reason about symbolically. Concolic execution [17][18] proceeds with a simplified, partial symbolic execution by using concrete values to simplify symbolic memory reference and path selection. Unfortunately, concolic execution techniques only examine one execution path at a time, where symbolic variables reflect only direct data dependencies. Since constraints on the values of multiple fields are usually checked in the exit condition of a loop, these techniques will miss the loop dependencies that are useful for constraint inference.

3) It is difficult to perform accurate and faithful analysis on binary code. The ubiquity of binary code means any security techniques that only require access to the program binary are widely applicable. To the best of our knowledge, all the approaches for protocol format extraction disassemble binary code into a sequence of assembly instructions and perform program analysis directly over the assembly instructions. However, an assembly-specific approach is unattractive because performing analysis on modern complex architectures tends to be onerous and tedious. The complexity of mordern architectures mainly involves two aspects. First, there are hundreds of instructions in these architectures. For example, x86 consists of over 300 instructions. Second, the instructions often have intricate and non-intuitive semantics.

3. Approach and System Overview

In this section we present our approach for protocol format extraction and introduce the system architecture of Icefex. Details of our approach are discussed in later sections.

Our approach performs IL-based concolic execution on BIL [16], a small set of well-specified and architecture-independent instructions, to reason about how an implementation of a protocol processes the received messages. Based on the concolic execution state, our approach extract protocol according to the semantics of each BIL instruction. Firstly, we identify fields based on the observation that evaluation behavior is the most prominent evidence of field boundary. Instead of analyzing all the instructions that process the received messages, our approach focuses on the instructions which are used to evaluate message fields and merge symbolic bytes in each argument of these instructions as a field. Moreover, we infer field semantics according to the rich semantic information contained in function calls and the effect of field value on the evaluation of other fields. We further enhance IL-based concolic execution with the loop-extended policy in [20], which broadens the coverage of symbolic results with loops. By relating the number of loop iterations with the attributes of fields, we infer constraints on the value of fields with field semantics together.

To realize our approach, the high-level architecture of Icefex has two phases, as shown in **Fig. 1**. The center box represents the primary contributions of this research.

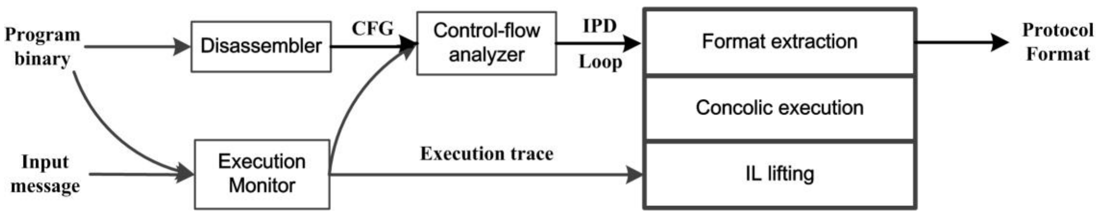


Fig. 1. System architecture of Icefex

In the first phase, we watch over the program execution as it processes a given message and collect necessary control-flow information for format extraction. This phase consists of three modules: *execution monitor*, *disassembler* and *control-flow analyzer modules*.

The execution monitor takes the program binary and the messages as input, and output an execution trace that contains a record of all the instructions performed by the program. Besides the binary code of instructions, the record contains the values of registers and memory accesses in each instruction. To introduce the knowledge of the semantics of platform specific functions (such as memory allocation), the execution monitor hooks these functions and attaches function summary to call instructions in execution trace. The disassembler translates machine language into assembly language and generates CFG for the binary. The control-flow analyzer module takes as input the CFG, as well as the execution trace, and output the *immediate post dominators* (IPD) of branch instructions and loops for constraint inference.

In the second phase, we replay and analyze the recorded execution trace to extract protocol format. This phase consists of three modules: *IL lifting*, *concolic execution* and *format extraction*. For each visited assembly instruction, our analysis consists of three steps. First, the IL lifting module lifts the assembly instruction to BIL instructions. We introduce it in Section 4.1. Next, the concolic execution module gets the operational semantics of BIL instruction and updates the concolic execution state. We present it in Section 4.2. Finally, the format extraction module takes as input the state and the BIL instruction, and outputs protocol formats. We provide the details in Section 5.

4. IL-based Concolic Execution

4.1 Instruction Lifting

As we discussed in Section 2.2, it is onerous and tedious to analyze each binary instruction in execution trace faithful to its semantics. To support analyzing in an easy-to-implement and concise fashion, Binary Analysis Platform (BAP) [16] reduces complex instruction sets to BIL, a small and formally specified intermediate language. However, BIL is designed for both static and dynamic scenarios and has many elements that are unnecessary for our trace-based analysis. For simplicity, we lift assembly instructions to an extended version of BIL, on which all subsequent analysis is performed. **Fig. 2** shows the syntax of the extended BIL.

$$
\begin{array}{lll}
\mathit{program} & ::= & \mathit{instr}\,* \\
\mathit{instr} & ::= & \mathit{var} := \mathit{exp} \,|\, \texttt{goto}\,\mathit{exp} \,|\, \texttt{if}\,\mathit{exp}\ \texttt{then goto}\,\mathit{exp}\ \texttt{else goto}\,\mathit{exp} \\
 & & |\,\texttt{store}(\mathit{exp},\mathit{exp},\tau_{reg}) \,|\, \texttt{label}\ \mathit{label\_kind} \,|\, \texttt{special}\,\text{string} \\
 & & |\,\texttt{call}\,\mathit{exp}\ \texttt{with}\,\mathit{argument}\ \texttt{ret}\,\mathit{var} \,|\, \texttt{halt} \,|\, \texttt{assert} \\
\mathit{exp} & ::= & \texttt{load}(\mathit{exp},\tau_{reg}) \,|\, \mathit{exp}\ \lozenge_b \mathit{exp} \,|\, \lozenge_u \mathit{exp} \,|\, \mathit{var} \,|\, \mathit{integer} \,|\, \texttt{cast}(\mathit{cast\_kind},\tau_{reg},\mathit{exp}) \\
\mathit{label\_kind} & ::= & \mathit{integer} \,|\, \text{string} \\
\mathit{cast\_kind} & ::= & \texttt{high}|\texttt{low}|\texttt{unsigned}|\texttt{signed} \\
\mathit{var} & ::= & (\text{string}, \mathit{id}_v, \tau_{reg}) \\
\lozenge_b & ::= & +,\ -,\ *,\ /,\ /_s, \text{mod}, \text{mod}_s,\ ==,\ !=,\ <,\ \leq,\ <_s,\ \leq_s,\ \&,\ |,\ \oplus \\
\lozenge_u & ::= & -\,(\text{unary minus}),\ \square\ (\text{bit-wise not}) \\
\mathit{argument} & ::= & (\mathit{var})^{+} \\
\tau_{reg} & ::= & \texttt{reg1\_t}|\texttt{reg8\_t}|\texttt{reg16\_t}|\texttt{reg32\_t}|\texttt{reg64\_t}
\end{array}
$$

Fig. 2. The syntax of **extended** BIL

The extension on BIL mainly involves **two** aspects: 1) the arguments indicating the endianness in `store` and `load` instructions are omitted, since currently we focus on x86 platform; 2) `call` instruction is introduced to provide the knowledge of the semantics of platform specific functions. It is obvious that the operational semantics of BIL instructions remain unchanged in spite of the modification and is still suitable for faithful binary program analysis.

Icefex lifts assemply instructions to BIL in a syntax directed manner, following the two steps given by previous work [16]. The first step is to translate assembly instructions into the VEX IL, a RISC-based language designed for the Valgrind dynamic instrumentation tool. In the second step, VEX IL is lifted to BIL by exposing all the implicit side-effects of instructions. Our contribution on instruction lifting is that Icefex also transforms the recorded function summary into `call` instruction, the abstract form of function calls.

4.2 Operational Semantics for Concolic Execution

To reason about the behavior of a program on the input message, Icefex maintains the concolic execution state and updates it according to the operational semantics of each lifted BIL instruction. To describe the operational semantics for concolic execution, we first formally define the concolic execution state.

DEFINITION 1 (**Concolic Execution State**) Concolic Execution State $\mathbb{S}$, mapping variables and memory addresses to both concrete values and symbolic expressions, is defined as a 4-tuple $\langle \mu, \Delta, \mathbb{S}_\mu, \mathbb{S}_\Delta \rangle$, where:

— μ maps a memory address to the current byte-sized value at that address, e.g., $\mu[m]$ denotes the concrete value at address m.

— Δ maps a variable to its value, e.g., $\Delta[var]$ denotes the concrete value of variable var.

— $\mathbb{S}_\mu$ maps a memory address to a symbolic expression, e.g., $\mathbb{S}_\mu[m]$ denotes the symbolic value at address m.

— $\mathbb{S}_\Delta$ maps a variable to a symbolic expression, e.g., $\mathbb{S}_\Delta[var]$ denotes the symbolic value of variable var.

Generally, consecutive memory locations are usually accessed as a whole. For brevity, we define $\mu[(m,w)]$ as the value of w-byte memory chunk starting from address m and $\mathbb{S}_\mu[(m,w)]$ as the corresponding symbolic expression. In DEFINITION 1, a symbolic expression is a function of input tags. Icefex supports seven kinds of symbolic expressions:

1) $c(:w)$ represents a w-byte integer constant.

2) $I_{(o,w)} = [i_o, \ldots, i_{o+w-1}]$ denotes a w-byte symbolic value, which is obtained by grouping byte-sized values represented by input tags $i_o, \ldots, i_{o+w-1}$ together.

3) $e_1 \Diamond_b e_2$ and $\Diamond_u e_1$ represent the symbolic result of binary and unary operations on the values represented by the expressions e_1 and e_2.

4) $\texttt{cast}(cast\_kind, \tau_{reg}, e)$ indexes the value represented by the expression e under different addressing modes.

5) $f(e_1, e_2, \ldots, e_n)$ describes the symbolic result of function call f that takes the expressions $e_1, e_2, \ldots, e_n$ as parameters.

6) $\texttt{table}(e, w)$ represents the symbolic value of w-byte memory chunk starting from the address represented by the expression e.

7) $\texttt{sub}(e, w, n)$ corresponds to the n-th byte in the w-byte value represented by e.

DEFINITION 2 (**Input Dependence**) An expression e depends on the input data $I_{(o,w)}$, denoted by $e \xrightarrow{ID} I_{(o,w)}$, if and only if all the input tags in $I_{(o,w)}$ appears in expression e. If we do not care about the details of input tags, $e \xrightarrow{ID} I_{(o,w)}$ can be abbreviated as $e \xrightarrow{ID} I$.

Based on the above metioned definitions, we redefine the operational semantics of BIL for concolic execution, as shown in **Fig. 3**. Each instruction rule is of the form:

$$\frac{\text{computation}}{< \text{current state} >, instr \mapsto < \text{end state} >} \tag{1}$$

Given an instruction, Icefex pattern-matches the instruction to find the applicable rule, and then performs the computation given in the top of the rule in the current state. If the computation is successful, there will be a transition denoted by the operator $\mapsto$ to update the state. If no rule matches, Icefex will turn next instruction without any operations. Because the next instruction *instr′* never changes, we omit it from the rules for brevity.

The expression rules for computation use a similar notation. We denote evaluating an expression *exp* to both concrete and symbolic values in the current state $\langle \mu, \Delta, \mathrm{S}_\mu, \mathrm{S}_\Delta \rangle$ as $\mu, \Delta, \mathrm{S}_\mu, \mathrm{S}_\Delta \Rightarrow exp \downarrow \langle v, e \rangle$. The expression *exp* is evaluated by matching *exp* to an expression rule and performing the attached computation.

Instructions

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp \downarrow \langle v,e\rangle \quad \Delta' = \Delta[var \leftarrow v] \quad \mathrm{S}'_\Delta = \mathrm{S}_\Delta[var \leftarrow e]}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta, var := exp \mapsto \mu,\Delta',\mathrm{S}_\mu,\mathrm{S}'_\Delta}\ \text{S-ASSIGN}$$

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp_1, exp_2 \downarrow \langle v_1,e_1\rangle,\langle v_2,e_2\rangle \quad w\texttt{=width}(\tau_{reg}) \quad \mu'=\mu[(v_1,w)\leftarrow v_2] \quad \mathrm{S}'_\mu=\mathrm{S}_\mu[(v_1,w)\leftarrow e_2]}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,\texttt{store}(exp_1,exp_2,\tau_{reg}) \mapsto \mu',\Delta,\mathrm{S}'_\mu,\mathrm{S}_\Delta}\ \text{S-STORE}$$

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow (var)^+ \downarrow \langle v_1,e_1\rangle,\langle v_2,e_2\rangle,\cdots \quad \mu' = \mu[var \leftarrow f(v_1,v_2\cdots)] \quad \mathrm{S}'_\mu = \mathrm{S}_\mu[var \leftarrow f(e_1,e_2\cdots)]}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,\texttt{call}\ f\ \texttt{with}\,(var)^+\ \texttt{ret}\,var \mapsto \mu,\Delta',\mathrm{S}_\mu,\mathrm{S}'_\Delta}\ \text{S-CALL}$$

Expressions

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp \downarrow \langle v,e\rangle \quad w\texttt{=width}(\tau_{reg}) \quad (e \xrightarrow{ID} I) = \mathbf{T} \quad P_{tablecheck}(v) = \mathbf{T}}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow \texttt{load}(exp,\tau_{reg}) \downarrow \langle \mu[(v,w)],\texttt{table}(e,w)\rangle}\ \text{S-TABLE}$$

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp \downarrow \langle v,e\rangle \quad w\texttt{=width}(\tau_{reg}) \quad (e \xrightarrow{ID} I) = \mathbf{F} \text{ or } P_{tablecheck}(v) = \mathbf{F}}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow \texttt{load}(exp,\tau_{reg}) \downarrow \langle \mu[(v,w)],\mathrm{S}_\mu[(v,w)]\rangle}\ \text{S-LOAD}$$

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp_1,exp_2 \downarrow \langle v_1,e_1\rangle,\langle v_2,e_2\rangle \quad v = v_1 \Diamond_b v_2}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp_1 \Diamond_b exp_2 \downarrow \langle v, e_1 \Diamond_b e_2\rangle}\ \text{S-BINOP} \qquad \frac{}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow var \downarrow \langle \Delta[var],\mathrm{S}_\Delta[var]\rangle}\ \text{S-VAR}$$

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp \downarrow \langle v,e\rangle \quad v_1 = \Diamond_u v}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow \Diamond_u exp \downarrow \langle v_1, \Diamond_u e_2\rangle}\ \text{S-UNOP} \qquad \frac{v = integer}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow integer \downarrow \langle v,v\rangle}\ \text{S-INT}$$

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp \downarrow \langle v,e\rangle \quad \text{extend (or extract) } v \text{ to } w \text{ bits}}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow \texttt{cast}(cast\_kind,\tau_{reg},exp) \downarrow \langle v,\texttt{cast}(cast\_kind,\tau_{reg},e)\rangle \mathrm{V}}\ \text{S-CAST}$$

Fig. 3. The operational semantics of the simplified BIL for concolic execution

In the computation, we indicate updating a variable or a memory location *x* with value *v* as $x \leftarrow v$, e.g., $\Delta[x \leftarrow 0\text{x3E}]$ denotes setting the value of variable *x* to the hexadecimal value 0x3E. As mentioned, the updates of consecutive memory locations are usually simultaneous when the instruction $\texttt{store}(exp_1, exp_2, \tau_{reg})$ stores the expression exp_2 in a τ_{reg}-type memory chunk starting from the address exp_1.

Since Icefex maintains the memory state (μ and S_μ) with byte granularity, it is essential to split the concrete and symbolic value of exp_2 into byte-sized values before the updates. The update of μ, denoted by $\mu[(m,w) \leftarrow v]$, is straight-forward by setting the byte-sized value at address *m*+*n*-1 to the *n*-th byte of *v* $(0 < n \le w)$. The update of S_μ, denoted by $\mathrm{S}_\mu[(m,w) \leftarrow e]$, is performed according to the form of symbolic expression *e*. The value of $\mathrm{S}_\mu[m+n-1]$ is assigned as follows.

$$\mathrm{S}_{\mu}[m+n-1]=\begin{cases} i_{o+n-1} & \exists I_{(o,w)} s.t.(e=I_{(o,w)}) \\ \texttt{sub}(e,w,n) & e\neq I_{(o,w)}, e\xrightarrow{ID}I \\ \mu[m+n-1] & \text{else} \end{cases} \tag{2}$$

Correspondingly, consecutive memory locations are also accessed simultaneously when the expression $\texttt{load}(exp,\tau_{reg})$ loads a τ_{reg} -type memory chunk starting from the address exp. The access of μ, denoted by $\mu[(m,w)]$, is performed directly by grouping the byte-sized values in the chunk together. The access of S_{μ}, denoted by $\mathrm{S}_{\mu}[(m,w)]$, depends on the form of symbolic values in the chunk $(0<n\leq w)$:

$$\mathrm{S}_{\mu}[(m,w)]=\begin{cases} I_{(o,w)} & \exists o \forall n(\mathrm{S}_{\mu}[m+n-1]=i_{o+n}) \\ e & \exists e \forall n(\mathrm{S}_{\mu}[m+n-1]=sub(e,w,n)) \\ \mu[(m,w)] & \text{else} \end{cases} \tag{3}$$

In addition to tracking direct dependencies, Icefex also pays attention to *table lookup*, a typical form of address dependencies. Table lookups occur when a symbolic expression is used as an index of a table to determine the location from which a value is loaded. In this case, the loaded value also depends on the memory address where this value is taken from. Since table lookups are widely used as switch structures in binary code to parse `keyword` fields in messages, it is important for Icefex to track them as well.

Given the expression $\texttt{load}(exp,\tau_{reg})$, the `S-TABLE` rule evaluates its symbolic value to $\texttt{table}(e,\texttt{width}(\tau_{reg}))$ if *exp* is used as an input-dependent index. In order to distinguish table lookups from normal address dependencies, we use the following predicate:

$$P_{tablecheck}(v)=\begin{cases} \mathbf{T} & \text{the address } v \text{ points to code segment} \\ \mathbf{F} & \text{else} \end{cases} \tag{4}$$

where **T** maps to true and **F** maps to false. If **T** is returned, the `S-TABLE` rule is implemented. Otherwise, the premise for the `S-LOAD` rule is met and only direct dependencies are tracked. By introducing the $P_{tablecheck}(v)$ policy, Icefex focuses on the targeted address dependencies and avoids the explosion caused by full symbolic pointer dereferences [21].

Furthermore, Icefex considers the dependencies between function parameters and the returned output. The `S-CALL` rule is defined for the `CALL` instruction and outputs the returned symbolic value of the form $f(e_1,e_2,\ldots,e_n)$ where $e_1,e_2,\ldots,e_n$ are the symbolic values of parameters. Note that we do not consider the parts of the execution trace inside any of the abstracted functions in the course of concolic execution, since their operational semantics have already been included as a whole.

In **Fig. 3**, we ignore the other six instruction types (`halt`, `assert`, `if`, `goto`, `label` and `special`) for three reasons: 1) Since Icefex performs instruction lifting on dynamic execution trace, the `halt` and `assert` instructions designed for static analysis never exist in the lifted BIL instructions; 2) the operational semantics of the `if`, `goto` and `label` instructions have no influence over the concolic execution state, while they change the control flow of program execution; 3) the `special` instruction, standing for any unhandled instructions during lifting (e.g., floating point instructions), is rarely used in the message processing. Overall, we believe that the concolic execution, performed by using only three instruction rules and seven expression rules, is faithful enough and easy-to-understand.

5. Protocol Format Extraction

In this section we present our techniques for extracting protocol formats in the light of concolic execution state. Icefex extracts protocol format in two phases: *field identification*, *semantics and constraint inference*, which are described in the following sub-sections.

5.1 Field Identification

Current techniques for field identification are based on a unique intuition−the way that an implementation of the protocol accesses the input data reveals a wealth of information about the field boundaries. However, there are also many instructions that read the input bytes without concern for field boundaries, since all variables, such as numbers, pointers and buffers, are processed as fixed-width integers in binary code.

As the basic unit of protocol format, the field is a consecutive sequence of input bytes with some meaning. Due to the semantic atomicity of message fields, all the input bytes that are evaluated simultaneously must be in the same field. In other words, the evaluation behavior is the most prominent evidence of field boundary. Therefore, our approach only focuses on the instructions in evaluation stage to avoid the redundancies and inconsistencies in lexing and parsing stages.

Like the tokens in program language, message fields also have two semantic types: *static semantics* and *dynamic semantics*. The static semantics, such as length, offset and checksums, define the field constraints that are too complex to express in syntactic formalisms. The dynamic semantics, such as port, IP address and timestamps, define how and when the implementation should produce a response behavior. The unique observation we obtain is that the fields with dynamic semantics are usually evaluated by function calls, while the fields with static semantics are evaluated by either branching points (the instructions that have more than one successors in the CFG) or function calls. In BIL, a branching point is a `goto` instruction with input-depended target or an `if` instruction. Based on this observation, Icefex identifies the consecutive bytes that used in the operand of `if`, `goto` and `call` instruction as a field. **Fig. 4** gives the formalized rules of field identification.

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp_1 \downarrow \langle v_1,e_1\rangle \quad e_1 \xrightarrow{ID} I_{(o,m)} \quad \neg\exists I_{(p,n)} s.t.(I_{(o,m)} \subset I_{(p,n)} \wedge e_1 \xrightarrow{ID} I_{(p,n)})}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,F,\texttt{if}\, exp_1 \;\texttt{then goto}\; exp_2 \;\texttt{else goto}\; exp_3 \mapsto \mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,F\cup\{\langle o,m\rangle\}}\ \text{F-IF}$$

$$\frac{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow exp \downarrow \langle v,e\rangle \quad e \xrightarrow{ID} I_{(o,m)} \quad \neg\exists I_{(p,n)} s.t.(I_{(o,m)} \subset I_{(p,n)} \wedge e \xrightarrow{ID} I_{(p,n)})}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,F,\texttt{goto}\, exp \mapsto \mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,F\cup\{\langle o,m\rangle\}}\ \text{F-GOTO}$$

$$\frac{\begin{array}{c}\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta \Rightarrow (var)^+ \downarrow \langle v_1,e_1\rangle,\langle v_2,e_2\rangle,\cdots \quad e_j \xrightarrow{ID} I_{(o_j,m_j)} \\ \neg\exists I_{(p_j,n_j)} s.t.(I_{(o_j,m_j)} \subset I_{(p_j,n_j)} \wedge e_j \xrightarrow{ID} I_{(p_j,n_j)}) \text{ the } j\text{-th param of } func \text{ has protocol-related semantics}\end{array}}{\mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,F,\texttt{call}\; func\; \texttt{with}\,(var)^+\; \texttt{ret}\; var \mapsto \mu,\Delta,\mathrm{S}_\mu,\mathrm{S}_\Delta,F\cup\{\langle o_j,m_j\rangle\}}\ \text{F-CALL}$$

Fig. 4. The formalized rules of field identification

The field identification rules are similar to the rules in **Fig. 3**. Somewhat differently, the former also take F, the set of fields, as a part of current execution state. If the computation of a rule is successful, Icefex will add the identified field to F. Take the `F-IF` rule for instance, all the input bytes in $I_{(o,m)}$ will be taken as a field, if the first parameter's symbolic value of `if` instruction e_1 depends on $I_{(o,m)}$ and there is no $I_{(p,n)}$ that contains $I_{(o,m)}$ and satisfies the condition $e_1 \xrightarrow{ID} I_{(p,n)}$. Note that Icefex applies the `F-CALL` rule to the function calls that have parameters with protocol-related semantics.

The main challenge to apply field identification rules is to find the actual max-sized $I_{(o,m)}$ included in a symbolic expression e. In order to eliminate the impact of register alias on binary analysis, BIL uses `cast` to index registers under different addressing modes. As a result, the identified field will be inconsistent with protocol format, if we take all the input bytes exsited in the symbolic expression as the max-sized $I_{(o,m)}$. Even worse, optimized binary code also uses bitwise logical instructions to index registers under different addressing modes, which further complicates the computation of rules. To achieve higher accuracy of field identification, Icefex revises the input bytes existed as the parameter of `cast` and bitwise logical operators (&, | and $\oplus$) in e, before combines them to the max-sized $I_{(o,m)}$.

To summarize our approach, we now return to the example of **Table 1** and explain how Icefex identifies fields based on concolic execution. **Table 2** shows the BIL instructions lifted from the assembly instructions in **Table 1**, and **Table 3** shows the Field identification at each step of the lifted BIL instructions. For simplicity, **Table 3** only gives the update of S_Δ, since field identification only depends on the symbolic values of variables.

As Line 1~5 of the BIL instructions are assignment instructions, Icefex applies the `S-ASSIGN` rule to update the concolic execution state, as shown in **Table 3**. On Line 6, the `F-IF` rule is applied to find the max-sized $I_{(o,m)}$ that the symbolic value of condition expression $((I_{(0,4)}\&0xFF-0xE3)==0x0)==0x1$ depends on. Although $I_{(0,4)}$ exists in the expression, Icefex identifies $\langle 0,1\rangle$ as a field, according to the logical operator & that cut out $I_{(0,4)}$ to $I_{(0,1)}$.

Table 2. BIL instructions lifted from assembly instructions

Assembly instruction	**BIL instruction**
1. `mov` esi, [esp+019Ch]	**1.** R_ESI := `load`(R_ESP+0x19C, `reg32_t`)
2. `and` esi, 0FFh	**2.** R_ESI := R_ESI & 0xFF
3. `lea` eax, [esi-1]	**3.** R_EAX := R_ESI-1
4. `cmp` eax, 0E2h	**4.** T_32t_1 := R_EAX-0xE2
	5. R_ZF := (T_32t_1 ==0x0)
	…… #instructions that update other flags
5. `jz` loc_527169	**6.** `if` R_ZF == 0x1 `then goto` loc_527169 `else`…

Table 3. Field identification at each step of the lifted BIL instructions

Line	**Rules**	**Update of S_Δ (*var*→*e*)**	***F***
1	`S-ASSIGN`	R_ESI→$I_{(0,4)}$	{}
2	`S-ASSIGN`	R_ESI→$I_{(0,4)}$&0xFF	{}
3	`S-ASSIGN`	R_EAX→$I_{(0,4)}$&0xFF-1	{}
4	`S-ASSIGN`	T_32t_1→$I_{(0,4)}$&0xFF-0xE3	{}
5	`S-ASSIGN`	R_ZF→($I_{(0,4)}$&0xFF-0xE3)==0x0	{}
6	`F-IF`		{ $\langle 0,1\rangle$ }

Compared with the field identification of previous approaches, Icefex avoids the redundancies introduced from parsing instructions on line 2~4, by focusing on the evaluation instruction on Line 5 (Line 6 of the BIL instruction). Moreover, Icefex evaluates the symbolic expression according to the real semantics of the `and` instruction, and derives the real field boundary. It is clear that Icefex identifies fields with higher accuracy than previous approaches.

5.2 Semantic and Constraint Inference

Semantic information indicates the intent of message fields, and constraint information indicates the dependencies across fields that a message must satisfy to be valid. Both of them are critical for understanding or reconstructing messages of unknown protocols. Current approaches have described how to identify field semantics, but none of them have discussed how to infer field constraints. In this section, we firstly present the details of semantic inference in Gofex, and then describe how Gofex infer the constraints accompanied with static semantics.

5.2.1 Semantics Inference

As we mentioned in Section 5.1, fields with dynamic semantics are usually evaluated by function calls, while the fields with static semantics are evaluated by either branching points or function calls. For a field evaluated by function calls, Icefex relies on prior work [15] to infer its semantics by leveraging the rich semantic information contained in the function parameters. For example, given a function call `memcmp`(e_1, e_2, e_3), e_1, e_2, e_3 are the symbolic expressions of parameters. Icefex believes that field $\langle o,w\rangle$ has length semantics if $e_3 \xrightarrow{ID} I_{(o,w)}$.

For a field evaluated by a branching point, the semantics is highly uncertain since binary instructions contain poor semantic information. To our minds, how the field value affects the evaluation of other fields reflects the semantics. Table 4 expresses the correspondence between four field semantics and the corresponding effect of field evaluation.

Table 4. Correspondence between field semantics and the effect of field evaluation

Field semantics	The effect of field evaluation
`keyword`	The field value x should satisfy the symbolic predicate of the form $g(x)=c$ to continue the evaluation of subsequent fields, or the program will pick another path.
`checksum`	The field value x should satisfy the symbolic predicate of the form $g(x)=g(y, z,\ldots)$ to continue the evaluation of subsequent fields, or the program will no longer process the message. y and z are the values of other fields.
`length`	The field value x should satisfy the symbolic predicate of the form $g(x)\{<,>,\neq\}c$ iteratively to access the bytes of a field, or the program will exit a loop.
`count`	The field value x should satisfy the symbolic predicate of the form $g(x)\{<,>,\neq\}c$ iteratively to evaluate a sequence of repeated fields, or the program will exit a loop.

Algorithm 1. Constraint inference Based on IL-based Concolic Execution

Input : BIL instruction $intr_m$, concolic execution state S, the set of identfied fields F current control dependence stack CDS, structure tree ST.
Output: updated F, CDS, ST

```
1. InferConstraint (intr_m, S, F, CDS, ST)
2.     while instr_m=IPD(CDS.top()) do     /*IPDs are obtained in the first phase of Icefex*/
3.         if CDS.top() evaluates f_a then
4.             inferSemanticsAndConstraint(f_a, ST, CDS)
```

```
5.            end if
6.            CDS.pop();
7.        end while
8.        f:=identifyField(S,intr_m);          /*identify field using the rules in Fig. 4 */
9.        if f≠NULL then
10.           F:={f}∪F
11.           if instr_m is a function call then
12.               Infer the semantics and constraint contained in the function
13.           end if
14.           addNode(f, ST);                  /*add a leaf node to the resulting ST.*/
15.           addEdge(CDS.top()→f, ST);        /*add an edge to the resulting ST.*/
16.       end if
17.       if instr_m is a branching point then
18.           addNode(instr_m);                /*add an internal node to the resulting ST.*/
19.           addEdge(CDS.top()→instr_m, ST);  /*add an edge to the resulting ST.*/
20.           CDS.push(instr_m);
21.       end if
22.   return F, CDS, ST
```

In a single execution trace, dynamic control dependence reveals runtime effects of branching points [22]. We reuse the algorithm in [23] which captures dynamic control dependence by a stack called control dependence stack (CDS) and maintains the effects of field identification in a structure tree, as shown in **Algorithm 1**. The underline text describes how we perform semantics and constraint inference.

In **Algorithm 1**, the instruction sequence affected by a branching point $\hat{s}_i$ ends with the immediate post-dominator $IPD(\hat{s}_i)$. Obviously the whole effect of field evaluation is availabe when instruction *IPD*(*CDS.top*()) is given and *CDS.top*() is the evaluation point of a field. At this point (line 3~5), Icefex infers field semantics heuristically based on the correspondence in **Table 4**. The processing steps of semantic inference are illustrated in **Fig. 5**, where $EP(f_a)$ denotes the evaluation point of field f_a and $x_i \xrightarrow{dcd} y_j$ denotes that execution instance x_i dynamically control depends on instance y_j.

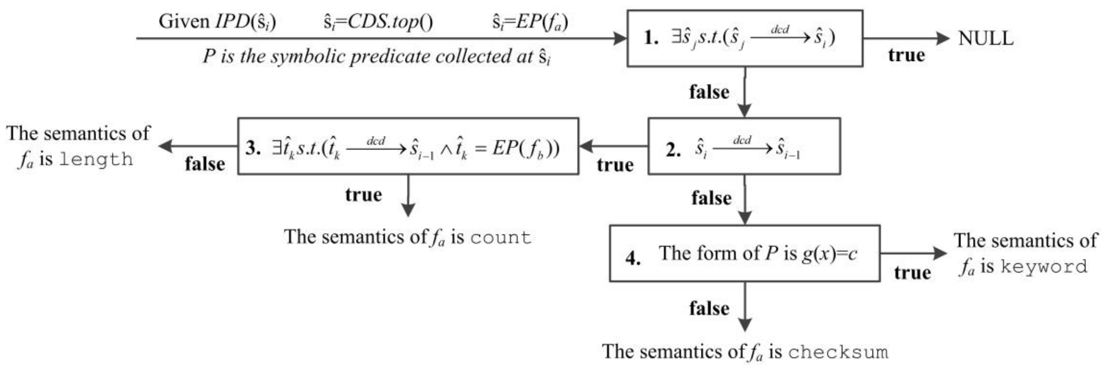

Given $IPD(\hat{s}_i)$ $\hat{s}_i$=$CDS.top()$ $\hat{s}_i$=$EP(f_a)$
P is the symbolic predicate collected at $\hat{s}_i$
1. $\exists \hat{s}_j s.t.(\hat{s}_j \xrightarrow{dcd} \hat{s}_i)$
true
NULL
false
2. $\hat{s}_i \xrightarrow{dcd} \hat{s}_{i-1}$
true
3. $\exists \hat{t}_k s.t.(\hat{t}_k \xrightarrow{dcd} \hat{s}_{i-1} \wedge \hat{t}_k = EP(f_b))$
false
The semantics of f_a is length
true
The semantics of f_a is count
false
4. The form of P is $g(x)$=c
true
The semantics of f_a is keyword
false
The semantics of f_a is checksum

Fig. 5. The processing steps of semantic inference

By the aid of the constructed ST, Icefex derives field semantics as follows:

- **Step 1**: Check whether there is another instance of $\hat{s}$ that dynamically depends on $\hat{s}_i$. If true, $\hat{s}_i$ must be an intermediate instance of the branching point in a loop, and the inference is terminated to avoid pointless repetition. If false, goto Step 2.

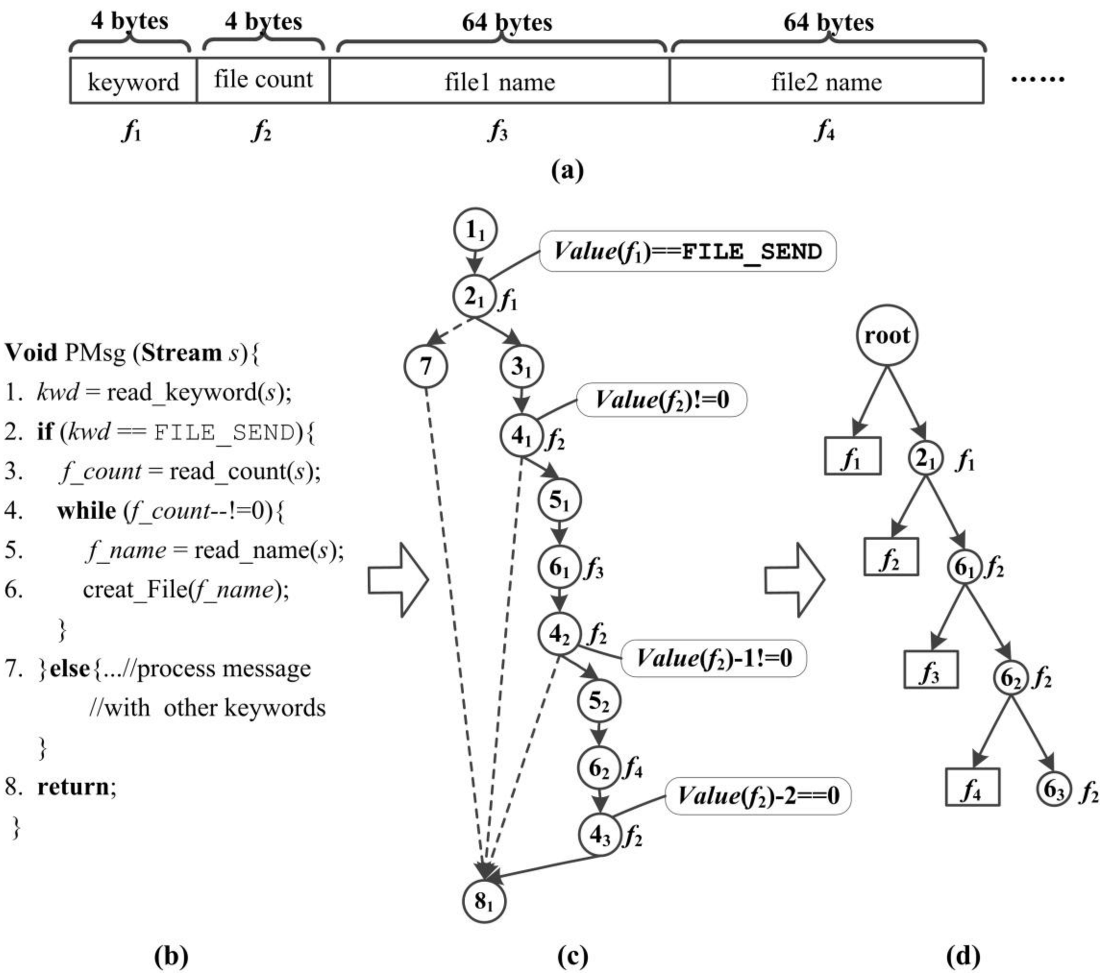
- **Step 2**: The parent node of $\hat{s}_i$ in the ST is checked to determine if $\hat{s}_i$ is the last instance of the branching point in a loop or not. If true, goto step 3. If false, $\hat{s}_i$ must be a normal instance and we should goto step 4.
- **Step 3**: Check if $\hat{s}_{i-1}$ has a child node $\hat{t}_k$ which is labeled by another field fb in the ST. If true, field fa has `count` semantics, or its semantics is `length`.
- **Step 4**: Check if the form of the symbolic predicates derived from $\hat{s}_i$ is g(x)=c. If true, field f_a has `keyword` semantics, or its semantics is `checksum`.

Note that the derived semantics is general and can be refined by using more detailed information. For example, a field with `keyword` semantics will be further refined as a format distinguisher [10] if the field value serves to differentiate the format of the subsequent part of the message. We leave the refinement for future work.

5.2.2 Constraint Inference

As the classification of semantics implies, message format constrains the value of a field, if and only if the field has static semantics. If a field with static semantics is evaluated by function calls, its value is constrained to be consistent with function semantics. Therefore, the constraint can be inferred simutaneously with semantic inference. For example, given a function call `memcmp`(e_1, e_2, e_3), e_1, e_2, e_3 are the symbolic expressions of parameters, Icefex will also obtain the constraint e_3=length(e_1) if the memory chunk referenced by e_1 depends on input bytes.

When a field with static semantics is evaluated by branching points, its value is constrained to execute a path in which all the input bytes of a message are processed correctly. For a `keyword` or `checksum` field, the constraint is a direct numerical relationship between values of fields. While for a `length` or `count` field, the constraint is an indirect relationship between the field value and the length or repetition count of other fields, For example, **Fig. 6(a)** shows a simple field sequence of a message that indicates the names of sended files, and the code in **Fig. 6(b)** processes the message in a top-down manner. Let the value of the field `keyword` be `FILE_SEND` and the value of the field `file count` be 2, concolic execution of execution trace is listed in **Table 5** and the corresponding dynamic control-flow graph is shown in **Fig. 6(c)**. Moreover, The ST constructed by **Algorithm 1** is shown in **Fig. 6(d)**.



```
Void PMsg (Stream s){
1.  kwd = read_keyword(s);
2.  if (kwd == FILE_SEND){
3.      f_count = read_count(s);
4.      while (f_count--!=0){
5.          f_name = read_name(s);
6.          creat_File(f_name);
        }
7.  }else{...//process message
          //with  other keywords
    }
8.  return;
}
```

Fig. 6. The example for semantic and constraint inference

Table 5. Correspondence between field semantics and the effect of field evaluation

Instance	Concolic Execution State	Loop Extended State
1_1	$kwd \to I_{(0,4)}$	∅
2_1	$kwd \to I_{(0,4)}$	∅
3_1	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$	∅
4_1	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$	∅
5_1	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$; $f\_name \to I_{(8,64)}$	∅
6_1	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$-1; $f\_name \to I_{(8,64)}$	$f\_count \to I_{(4,4)}$-TC_1
4_2	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$-1; $f\_name \to I_{(8,64)}$	$f\_count \to I_{(4,4)}$-TC_1
5_2	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$-1; $f\_name \to I_{(72,64)}$	$f\_count \to I_{(4,4)}$-TC_1
6_2	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$-2; $f\_name \to I_{(72,64)}$	$f\_count \to I_{(4,4)}$-TC_1
4_3	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$-2; $f\_name \to I_{(72,64)}$	$f\_count \to I_{(4,4)}$-TC_1
8_1	$kwd{:}\to I_{(0,4)}$; $f\_count \to I_{(4,4)}$-2; $f\_name \to I_{(72,64)}$	$f\_count \to I_{(4,4)}$-TC_1

For a `keyword` or `checksum` field, Icefex also take the symbolic predicate collected at branching point as field constraint at step 4 of semantic inference. In **Fig. 6**, as 2_1 is the only evaluation point of field f_1 and the symbolic predicate collected at 2_1 is

Value(f_1)==`FILE_SEND`, Icefex identifies the semantics of f_1 as `keyword` and take the predicate as constraint.

In contrast, inferring constraint on a `length` or `count` field needs much more effort since it covers a class of paths that include different numbers of loop iterations. In Fig. 6, none of the predicates collected at 4_1, 4_2 and 4_3 are equal to the real constraint. To additionally express how a field value relates to the length or count of other fields, Icefex enhances IL-based concolic execution with a loop-extended policy in [20]. In addition to maintaining the data dependencies of variables on input bytes, Icefex introduce *trip counts*, a new class of symbolic tags for the number of times that each loop executes, to identify loop-dependent variables. More details about the policy such as loop information extraction and the rules for operations on trip counts can be found in [20]. Based on the enhanced concolic execution, Icefex infers the loop-related constraints in three steps:

- **Step 1:** Collect the symbolic predicate at the last instance of the branch point in a loop. Since the symbolic expression of variables also capture certain loop dependent effects, the symbolic predicate expresses the relation between trip count and field values.
- **Step 2:** Match the loop with the fields over which it operate, and link the trip count to the attributes of fields operated in the loop, such as lengths and repetition counts.
- **Step 3:** Combine the symbolic predicate in Step 1 together with the link in Step 2 and output the relations between field values and the attributes of fields as constraints.

Let us revisit the example in **Fig. 6**. Since f_2 is evaluated in the exit condition of a loop, Icefex performs the inference at 4_3, the last instance of the branching point in the loop. In the first step, the predicate collected at 4_3 is $Value(f_2)\text{-}TC_1=0$, where TC_1 is the trip count. In the second step, Icefex find that TC_1 equals to the repetition count of field sequence f_3f_4, and determine the semantics of f_2 as `count`. In the last step, Icefex combines $f_2\text{-}TC_1=0$ with $TC_1=Count(f_3f_4)$ and outputs the resulting constraints as $Value(f_1)= Count(f_3f_4)$.

6. Implementation and Evaluation

We evaluated the effectiveness of Icefex by implementing a prototype system based on the proposed techniques. In this section, we first describe our implementation details and evaluation methodology. After that, we summerize the experimental results.

6.1 Implementation

We have implemented Icefex on Ubuntu 9.04. The implementation details of the six modules in Icefex are the following:

Execution monitor. Icefex utilizes TEMU, the dynamic analysis component in Bitblaze [24], to take a binary execution trace and record function calls. TEMU is built upon a whole-system emulator and supports performing analysis on multi-platforms.

Disassembler. Our infrastructure uses IDA Pro [25], one of the most popular tools for static binary analysis, to disassemble binaries and generate CFGs.

Control-flow analyzer. We implemented it as an IDA plugin to derive IPDs and loops from CFGs, by reusing the standard detection algorithms in [26].This module also takes the execution trace as input to avoid unnecessary analysis on the unexecuted instructions.

IL lifting. Icefex extends the `toil` tool in Binary Analysis Platform (BAP) [16] to lift binary code to the simplified BIL. The extension involves two aspects: 1) transforming the recorded function calls in the trace to `call` instructions; 2) mapping the IPDs and loops in binary code to the lifted BIL.

Concolic execution. We implemented the concolic execution module in Python to reason about the program behavior. It reuses the Python code in the STP solver [27] to perform constant folding and algebraic simplification on the maintained symbolic expression.

Format extraction. This module, the core component of Icefex, is also implemented in Python. It starts to work when the monitored program receives a message, and finishes the analysis after the analysis of all the input bytes.

6.2 Evaluation Methodology

Our experiments evaluated Icefex on 9 representative messages of 5 known protocols (DNS, eDonkey, FTP, HTTP and McAfee ePO), as shown in **Table 6**. The binary size represents the main executable of servers if there are several. For simplicity, all of the servers we analyzed are binaries running on Windows. As BIL is platform-independent, we believe that Icefex can also work well on other platforms.

Table 6. Summary of the five known protocols in the evaluation

Protocol	Server (Target)	Client	Binary Size(KB)
DNS	Deadwood 3.2.02	nslookup	63
eDonkey	eMule 0.48a	eMule 0.48a	5,184
FTP	FileZilla Server 0.9.41	FileZilla 3.5.3	617
HTTP	Apache 2.4.2	Firefox 15.0.1	1,018
McAfee ePO	McAfee ePO 4.5	McAfee Agent 3.6	1,106

We use the above protocols to compare the message format automatically extracted by Icefex with the the real formats from standard or published specifications. For comparison, our experiments also re-implemented and evaluated the approaches in AutoFormat [12] and Tupni [14]. The execution traces with taint information are required by AutoFormat and Tupni, and we obtained them using the tracecap plugin in TEMU. As illustrated in Fig. 7, The procedure of our experiments consists of four steps:

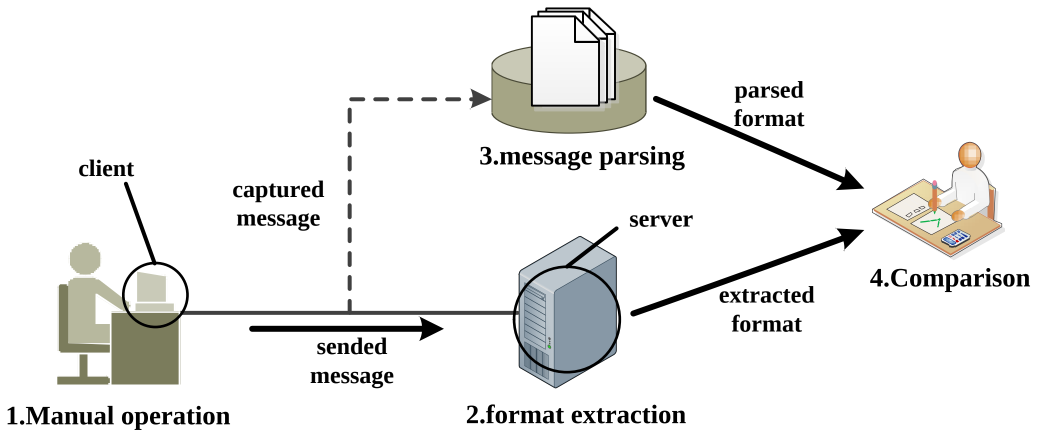


Fig. 7. The procedure of our experiments

- **Step 1**: We operate the client program of protocol implementations manually to sent messages to the server on the guest operating system of TEMU. Both the client and the server are in local network.

- **Step 2**: We use the loaded plugin in TEMU to track how the server processes the messages from client host. Then, the tools for protocol format extraction, are used to analyze the execution trace and output the message format.
- **Step 3**: We also capture the network traffic between the client and server using Wireshark [28], a popular network protocol analyzer. Since the formats extract by Wireshark are not completely error-free [12][15], we further perform deeper field discovery manually according to the protocol specification.
- **Step 4**: We compare the extracted format to the parsed format and measure the accuracy in two aspects: field identification, semantic and constraint inference.

6.3 Experimental results

6.3.1 The accuracy of Field Identification

We count the numbers of *fine-grained* fields that were identified correctly by AutoFormat, Tupni and Icefex. Moreover, we count the number of *over-fine-grained* fields, each of which is only part of a real field. We also count the number of *coarse-grained* fields, which contain input bytes from multiple real fields. The results are reported in **Table 7**.

Table 7. Comparison of the number of fields identified by AutoFormat, Tupni and Icefex

Protocol	**Message Type**	**#Real Fields**	**AutoFormat**			**Tupni**			**Icefex**		
			$\|F\|$	**$\|F_o\|$**	**$\|F_c\|$**	**$\|F\|$**	**$\|F_o\|$**	**$\|F_c\|$**	**$\|F\|$**	**$\|F_o\|$**	**$\|F_c\|$**
DNS	Query	13	9	2	1	9	2	1	10	0	1
eDonkey	AskSharedFiles	3	2	2	0	3	0	0	3	0	0
	ChangeID	6	4	4	0	4	1	1	6	0	0
	Hello	23	21	4	0	21	1	1	21	0	0
FTP	USER Request	4	4	0	0	4	0	0	4	0	0
	RETR Request	4	4	0	0	4	0	0	4	0	0
HTTP	GET Request	39	34	15	0	34	15	0	39	0	0
	POST Request	53	46	21	0	46	21	0	53	0	0
McAfee ePO	IncProps	67	0	966	0	28	854	0	62	0	1
Total		212	124	1014	1	153	894	3	202	0	2

In **Table 7**, we represent the number of fine-grained fields, over-fine-grained fields and coarse-grained fields as $|F|$, $|F_o|$ and $|F_c|$ respectively. We take the totals of $|F|$, $|F_o|$ and $|F_c|$, and obtain the accuracies of field identificaion: 124/212=58.5% for Autoformat, 153/212=72.2% for Tupni and 202/212=95.3% for Icefex. It is obvious that Icefex is more accurate than AutoFormat and Tupni. Particularly, the results of Icefex only contain few coarse-grained fields, while the results of others contain both coarse-grained fields and over-fine-grained fields. In the following, we describe our experiments in greater details.

DNS: In this experiment, the message under study is a DNS Query that requests the IP address of the host *www.google.com*. The detailed fields identified by AutoFormat, Tupni and Icefex are shown in **Fig. 8**. Compared to the real format, all the three results contain a coarse-grained field that consists of three real fields: `ANCount`, `NSCount` and `ARCount`. By analyzing the execution trace, we find out the main reason behind the coarse-grained field is that Deadwood simply ignores these three fields. As the missing bytes are consecutive, they are merged into one field for the integrity of the message. Moreover, **Fig. 8** also shows the existence of over-fine-grained fields. As Deadwood access the field `Flags` at byte-granularity, the two-byte field is considered to be 2 consecutive byte-long fields in both AutoFormat and Tupni results. In contrast, Icefex identified the field `flags` correctly since the field is evaluated on a comparison instruction as a whole.

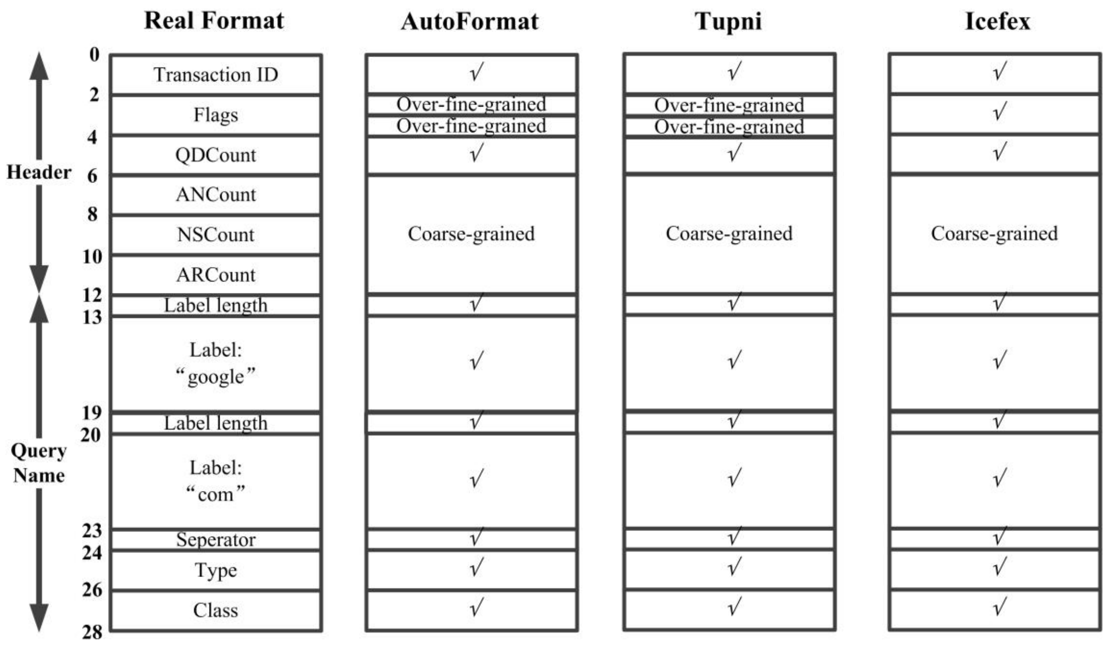


Fig. 8. Detailed comparison on the fields identified for the DNS Query message

eDonkey: Since the messages of eDonkey protocol have various formats, we have used three representative messages that differ significantly in length (AskSharedFiles, ChangeID and Hello messages). The results in **Table 7** show that both AutoFormat and Tupni report over-fine-grained fields as well as a coarse-grained field, while Icefex identifies all the message fields correctly. To find out the root cause of the errors, we perform a detailed comparison between these results and the real format. For example, the fields identified for the ChangeID message are shown in **Fig. 9**.

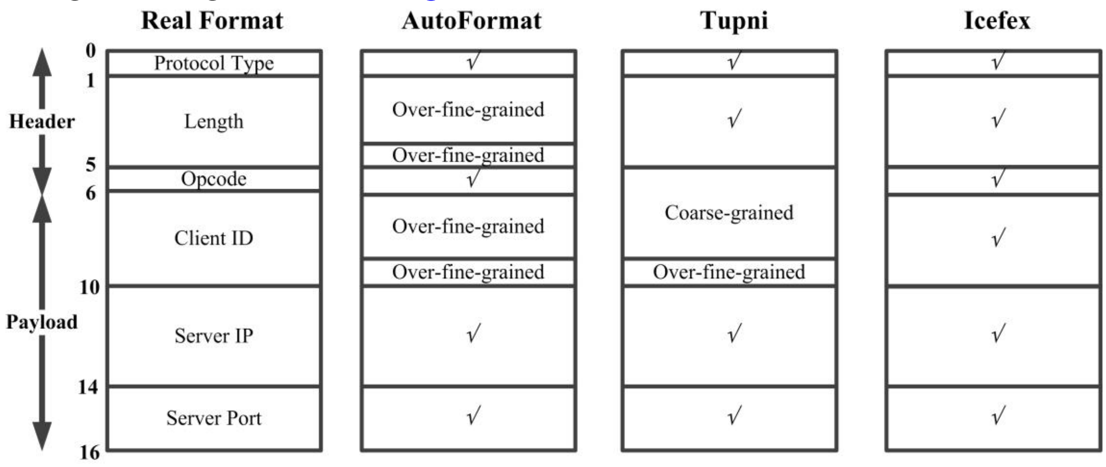


Fig. 9. Detailed comparison on the fields identified for the eDonkey ChangeID message

As illustrated in **Table 7**, the field `Protocol Type` is accessed as a four-byte field $\langle 0,4\rangle$ which overlaps $\langle 1,4\rangle$ (the field `Length`). To remove the inconsistency, AutoFormat aims at the smallest sequence that cannot be further divided into smaller sub-fields and reports $\langle 0,1\rangle$, $\langle 1,3\rangle$ and $\langle 4,1\rangle$ as finest-grained fields. Unlike AutoFormat, Tupni uses a greedy algorithm to find a consistent subset F of identified fields. Since the access weight of field `Length` is larger than that of field `Protocol Type` (the former is accessed at the branching point of a loop more frequently), Tupni reserves the field $\langle 1,4\rangle$ correctly and reports $\langle 0,1\rangle$ as a field of missing bytes. The identification of field `Opcode` is similar. Somewhat differently, Tupni reserves the field $\langle 5,4\rangle$ incorrectly, which leads to a coarse-grained field as well as an over-fine-grained field. As expected, Icefex outperforms AutoFormat and Tupni by directly reporting `Protocol Type` and `Opcode` as byte-long fields at the evaluation points.

FTP: We exprimented with FTP protocol by monitoring the execution trace of the Filezilla server. Because the FTP messages only contain few string fields, Icefex, AutoFormat and Tupni have identified the message fields accurately, as shown in **Table 7**.

HTTP: In this experiment, we perform analysis on GET Request message and POST Request message, both of which send data to the HTTP server as part of request. The results in **Table 7** show that both AutoFormat and Tupni report over-fine-grained fields. By comparing the detailed results with the real format, we found that all the unmatched fields are `Quality Factor` fields which allow the user to indicate the relative degree of preference for the media-range. Generally, a `Quality Factor` field, such as “q=0.500”, expresses the degree as a decimal fraction that scales from 0 to 1. Apache accesses the integer part and the fractional part separately to derive the fraction value, which is the root cause of the over-fine-grained fields in AutoFormat and Tupni results. Instead, Icefex identifies all the `Quality Factor` fields correctly since Apache evaluates each `Quality Factor` field as a whole.

McAfee ePO: In this experimemt, we focus on the IncProps message that allow the client to check its custom properties. Both AutoFormat and Tupni report numerous over-fine-grained fields while Icefex derives almost all the real fields, as shown in **Table 7**. By mapping the message payload to the correponding binary handing code, we found that each message is obfuscated by XORing message data with the static byte 0xAA. Because of this, AutoFormat identifies each byte of the message as a field since it aims at the smallest sequence that cannot be further divided. As regards Tupni, the situation is slightly better. Because the access weight of each length field is much larger than the total weight of included bytes, Tupni reports all the 28 length fields correctly. In comparison, the result of Icefex exactly matches the real format except that 5 fields used for signature are combined to one field. Similar to the experiments on DNS Query message, the main reason is that the ePO server simply ingores these fields.

6.3.2 The accuracy of Semantics and Constraint Inference

In this section, we discuss Icefex’s accuracy on semantics and constraint inference. For each evaluated message, we count the number of fields whose semantics are inferred correctly by Icefex. The results show that Icefex can derive the real field semantics, which is beyond the capabilities of existing approaches. More specifically, **Table 8** presents the number of static semantics, dynamic semantics and constraints inferred by Icefex.

Table 8. The results of semantic and constraint inference performed by Icefex

Protocol	**Message Type**	**Static Semantics**		**Dynamic Semantics**		Constraints	
		#Real	**#Inferred**	**#Real**	**#Inferred**	#Real	#Inferred

DNS	Query	10	7	3	1	10	7
eDonkey	AskSharedFiles	3	3	0	0	3	3
	ChangeID	3	3	3	2	3	3
	Hello	18	18	5	2	18	18
FTP	USER Request	3	3	1	0	3	3
	RETR Request	3	3	1	1	3	3
HTTP	GET Request	34	34	5	1	34	34
	POST Request	48	48	5	1	48	48
McAfee ePO	IncProps	48	45	19	3	48	45

As shown in **Table 8**, Icefex inferred all the fields with static semantics, except the coarse-grained fields that are ignored by the programs. Take the `ANCount`, `NSCount` and `ARCount` fields in **Fig. 8** for example, their values are set to zero as they indicate no resource records in DNS query message. Moreover, unlike current approaches using dynamic taint analysis, Icefex also accurately infers the symbolic constraints involved in static semantics. For instance, the first field in ePO IncProps message, a two-byte magic field with keyword semantics, involves the constraint that $\mathit{Value}(f_{\text{magic}}) \oplus 0\text{xAAAA}=0\text{x4F50}$.

In comparison, many fields with dynamic semantics are not correctly inferred. There are two main reasons behind the limited accuracy of semantic inference: (1) Many fields with dynamic semantics, such as `TransactionID` in DNS message, are evaluated by user-defined functions whose abstracts are unavailable. As a result, Icefex only labels these fields with static semantics according to the internal instructions of user-defined functions. (2) Many fields with dynamic semantics, are not evaluated immediately by the programs. As a representative example, the fields about the client properties in ePO IncProps message are stored in the server's database firstly and will not be evaluated until the check or update event is triggered. Therefore, Icefex has missed the dynamic semantics of these fields.

7. Conclusions and Future Work

We have presented a novel approach for protocol format extraction. Based on IL-based concolic execution, our approach can reason about the evaluation behavior of the programs on input messages and derive the protocol formats. We have implemented our approach into a system called Icefex and evaluated it over several real-world protocols. Experimental results show that Icefex can effectively extract protocol formats with higher accuracy and acceptable time overhead.

Currently Icefex is not capable of extracting formats of encrypted messages. We will resolve this problem by a more sophisticated implementation, which can automatically identify buffers that hold decrypted messages and use these buffers as starting points for protocol format extraction. Another limitation of Icefex is the imperfect dynamic semantics inference. In the future we plan to perform our analysis on the whole execution trace in a session to obtain more information about field semantics.

References

[1] P. Godefroid, A. Kiezun and M.Y. Levin, "Grammar-based whitebox fuzzing," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 206-215, June, 2008. Article (CrossRef Link)

[2] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification

extraction," in *Proc. of 30th IEEE Symposium on Security and Privacy*, pp. 110-125, May 17-20, 2009. Article (CrossRef Link)
[3] H. Dreger, A. Feldmann, M. Mai, V. Paxson and R. Sommer, "Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection," in *Proc. of 15th USENIX Security Symposium*, pp.257-272, July 31-August 1, 2006. Article (CrossRef Link)
[4] V. Paxson, "Bro: A system for detecting network intruders in real time," *Computer Networks*, vol. 31, no. 23, pp. 2435-2463, 1999. Article (CrossRef Link)
[5] J. Caballero, S. Venkataraman, P. Poosankam, M. G. Kang, D. Song, and A. Blum, "FiG: Automatic fingerprint generation," in *Proc. of 14th Annual Network and Distributed System Security Symposium*, February 28-March 2, 2007. Article (CrossRef Link)
[6] About Pidgin, http://www.pidgin.im/about/
[7] Protocol information project, http://www.4tphi.net/~awalters/PI/PI.html
[8] C. Leita, K. Mermoud and M. Dacier, "Scriptgen: an automated script generation tool for honeyd," in *Proc. of 21st Annual Computer Security Applications Conference*, pp. 203-214, December 5-9, 2005. Article (CrossRef Link)
[9] W. Cui, V. Paxson, N. C. Weaver and R. H. Katz, "Protocol-Independent Adaptive Replay of Application Dialog," in *Proc. of 13th Network and Distributed System Security Symposium*, February, 2006. Article (CrossRef Link)
[10] W. Cui, J. Kannan and H. Wang, "Discoverer: automatic protocol reverse engineering from network traces," in *Proc. of 16th USENIX Security Symposium*, pp. 1-14, August 6-10, 2007. Article (CrossRef Link)
[11] J. Caballero, H. Yin, Z. Liang, D. Song, "Polyglot: Automatic Extraction of Protocol Format using Dynamic Binary Analysis," in *Proc. of 14th ACM Conference on Computer and Communications Security*, pp. 317-329, September 29-October 2, 2007. Article (CrossRef Link)
[12] Z. Lin, X. Jiang, D. Xu and X. Zhang , "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution," in *Proc. of 15th Symposium on Network and Distributed System Security*, February 8-11, 2008. Article (CrossRef Link)
[13] G. Wondracek, P. Comparetti, C. Kruegel and E. Kirda, "Automatic network protocol analysis," in *Proc. of 15th Symposium on Network and Distributed System Security*, February 8-11, 2008. Article (CrossRef Link)
[14] W. Cui, M. Peinado, K. Chen, H.J. Wang and L. Irun-Briz, "Tupni: Automatic Reverse Engineering of Input Formats," in *Proc. of 15th ACM Conference on Computer and Communications Security*, pp. 391-402, October 27-31, 2008. Article (CrossRef Link)
[15] J. Caballero, P. Poosankam, C. Kreibich and D. Song, "Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering," in *Proc. of 16th ACM Conference on Computer and Communications Security*, pp. 621-634, November 9-13, 2009. Article (CrossRef Link)
[16] David Brumley and Ivan Jager, *The BAP Handbook*, http://bap.ece.cmu.edu/doc/bap.pdf
[17] P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing," in *Proc. of the 2005 ACM SIGPLAN Conference on Programing Language Design and Implementation*, pp. 213-223, June 12-15, 2005. Article (CrossRef Link)
[18] K. Sen, D. Marinov and G. Agha, "Cute: a concolic unit testing engine for c," in *Proc. of 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 263-272, September 5-9, 2005. Article (CrossRef Link)
[19] Intel IA-32 Architectures Software Developer's Manual. http://download.intel.com/products/processor/manual/253667.pdf
[20] P. Saxena, P. Poosankam, S. McCamant and D. Song, "Loop-extended symbolic execution on binary programs," in *Proc. of 18th International Symposium on Software Testing and Analysis*, pp. 225-236, July 19-23, 2009. Article (CrossRef Link)
[21] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *Proc. of 4th ACM European conference on Computer systems*, pp. 61-74, April 1-3, 2009. Article (CrossRef Link)
[22] B. Xin and X. Zhang, "Efficient online detection of dynamic control dependence," in *Proc. of 16th*

International Symposium on Software Testing and Analysis, pp. 185-195, July 9-12, 2007. Article (CrossRef Link)

[23] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proc. of 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp.83-93, November 9-15, 2008 Article (CrossRef Link)

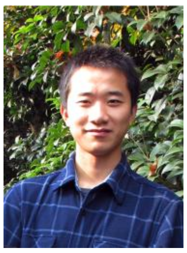
[24] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," ICISS, LNCS 5352, pp. 1-25, 2008 Article (CrossRef Link)

[25] IDA Pro, http://www.hex-rays.com/products/ida/index.shtml

[26] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Second Edition, Addison Wesley, 2006. Article (CrossRef Link)

[27] STP Solver, http://people.csail.mit.edu/vganesh/STP_files/stp.html

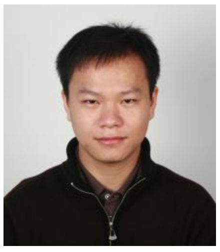
[28] Wireshark. http://www.wireshark.org/



Fan Pan was born in Wuhu, Anhui, China in 1987. He received his B.E. and the M.S degree in network engineering from PLA University of Science and Technology in 2007 and 2009 respectively. Now he is a Ph.D. candidate in the university. His research interests include protocol reverse engineering and software testing.

Li-fa Wu was born in Qichun, Hubei, China in 1968. He received his Ph.D. from Nanjing University in 1998. He is currently a professor in PLA University of Science and Technology. His research fields concern network security, protocol engineering and satellite communication.

Zheng Hong was born in Yingtan, Jiangxi, China in 1979. He received his Ph.D. from PLA University of Science and Technology in 2007. Now he is an associate professor in the university. His research fields concern network security and protocol reverse engineering.



Hua-bo Li was born in Gongan, Hubei, China in 1981. He received his Ph.D. from PLA University of Science and Technology in 2011. He is currently a lecturer in the university. His research fields concern network security and satellite communication.

Huai-guang Lai was born in Pingba, Guizhou, China in 1975. He received his Ph.D. from Nanjing University in 2006. Now he is an associate professor in PLA University of Science and Technology. His research fields concern network management and satellite communication.

Cheng-hui Zheng was born in Guangshan, Henan, China in 1977. He received his M.S degree from PLA University of Science and Technology. Now he is a lecturer in the university. His research fields concern network security and satellite communication.