Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Automatic protocol reverse engineering for industrial control systems with dynamic taint analysis[*#]

Rongkuan MA[1], Hao ZHENG[2], Jingyi WANG[2], Mufeng WANG[2], Qiang WEI[‡1], Qingxian WANG[1]

[1]*State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China*

[2]*Zhejiang University NGICS Platform, Hangzhou 310000, China*

E-mail: rongkuan233@gmail.com; zjuzhenghao@gmail.com; wangjyee@gmail.com; csewmf@zju.edu.cn; weiqiang66@126.com; wangqingxian2015@163.com

Received Dec. 19, 2020; Revision accepted Apr. 15, 2021; Crosschecked Jan. 13, 2022

Abstract: Proprietary (or semi-proprietary) protocols are widely adopted in industrial control systems (ICSs). Inferring protocol format by reverse engineering is important for many network security applications, e.g., program tests and intrusion detection. Conventional protocol reverse engineering methods have been proposed which are considered time-consuming, tedious, and error-prone. Recently, automatical protocol reverse engineering methods have been proposed which are, however, neither effective in handling binary-based ICS protocols based on network traffic analysis nor accurate in extracting protocol fields from protocol implementations. In this paper, we present a framework called the industrial control system protocol reverse engineering framework (ICSPRF) that aims to extract ICS protocol fields with high accuracy. ICSPRF is based on the key insight that an individual field in a message is typically handled in the same execution context, e.g., basic block (BBL) group. As a result, by monitoring program execution, we can collect the tainted data information processed in every BBL group in the execution trace and cluster it to derive the protocol format. We evaluate our approach with six open-source ICS protocol implementations. The results show that ICSPRF can identify individual protocol fields with high accuracy (on average a 94.3% match ratio). ICSPRF also has a low coarse-grained and overly fine-grained match ratio. For the same metric, ICSPRF is more accurate than AutoFormat (88.5% for all evaluated protocols and 80.0% for binary-based protocols).

Key words: Industrial control system (ICS); ICS protocol reverse engineering; Dynamic taint analysis; Protocol format

https://doi.org/10.1631/FITEE.2000709 **CLC number:** TP309.2

1 Introduction

Industrial control system (ICS) protocols are widely adopted in industrial automation systems, such as the supervisory control and data acquisition (SCADA) system for power grids or water treatment and the distributed control system (DCS) for power generation and chemical industry. Their formats are usually undocumented in practice, which hinders advancement in ICS cybersecurity research, such as intrusion detection, investigation forensics, and fuzz testing. Reverse engineering of protocol format is great helpful to speed up relevant research in other areas (Fang et al., 2020; Fioraldi et al., 2020; Yang et al., 2020). For example, in Peach[*] (Luo et al., 2020), the reorganization of the ICS

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 61833015)

# Electronic supplementary materials: The online version of this article (https://doi.org/10.1631/FITEE.2000709) contains supplementary materials, which are available to authorized users

ORCID: Rongkuan MA, https://orcid.org/0000-0002-4791-6847; Qiang WEI, https://orcid.org/0000-0002-7207-6691 protocol function code improves the efficiency of protocol fuzzing significantly. However, ICS protocol reverse engineering has long been considered time-consuming and error-prone, and requires substantial efforts (Denton et al., 2017; Senthivel et al., 2017).

Previous works on reverse engineering ICS protocols focus mainly on network-based methodologies that extract protocol format from network traffics using algorithms like differential analysis and clustering algorithms (Choi et al., 2016; Chang et al., 2017; Ji et al., 2017). These methodologies have the following limitations:

1. Their accuracy depends heavily on data collection, which is difficult to achieve in practice without tremendous manual effort.

2. The information extracted from traffic is not accurate due to encoding or encryption (e.g., they even fail to recognize a unicode-encoded byte in a message as a protocol field or multiple individual protocol fields).

3. Their analysis methodologies need to assume the types and features of message fields for classification, which reduces their generality.

Other works based on program analysis perform well on text-based protocols (e.g., HTTP) (Caballero et al., 2007; Lin et al., 2008). Among these, Polyglot uses heuristics to obtain message semantics by monitoring protocol application execution traces. However, its heuristic methodologies are also non-trivial when applied to ICS protocols. For example, in ICS protocols, there are usually no delimiters or keywords in a text-based protocol message. A framework has been proposed in AutoFormat to reverse engineer protocol format based on monitoring the call stack, without a wealth of information about the protocol format. However, for analyzing a binary-based protocol program, it is often too coarse-grained and not accurate enough.

In this paper, we propose an automatic protocol reverse engineering framework designed for ICS protocols while adopting the power of program analysis. Our key insight is that an individual field in a message is typically handled in an individual basic block (BBL) group, based on which we implement an automatic ICS protocol reverse engineering framework (ICSPRF) that is based on dynamic taint analysis. We implement a dynamic taint analysis engine from scratch, which is suitable and scalable for binary-based protocol analysis. Compared with previous works, ICSPRF is automated and does not need any prior knowledge of the message fields. We evaluate ICSPRF on six open-source implementations of four popular ICS protocols (i.e., Modbus, IEC104, DNP3, and s7comm). Compared to Wireshark (SharkFest, 2020), one of the most popular protocol analysis tools, ICSPRF can reverse the ICS protocol format more effectively; i.e., it can identify every single protocol field with an accuracy of 94.3% on average.

2 Preliminaries

In this section, we first introduce the protocol terminology used in this work and then demonstrate our problem scope and the background.

2.1 Terminology

A protocol message is composed of a sequence of field chunks, where a field chunk is the smallest contiguous sequence of byte data with some meaning. For example, Fig. 1 shows a read register query message of the Modbus protocol, which includes several field chunks such as length including two bytes in positions 4 and 5, function code including a byte in position 7, a start address including bytes in positions 8 and 9, and register quality including bytes in positions 10 and 11.

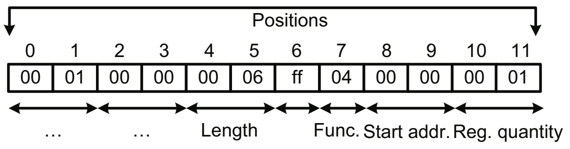


Fig. 1 An example of the Modbus protocol read register query message
When the program processes the input data, the destination memory is tainted with the original offset position

2.2 Problem scope

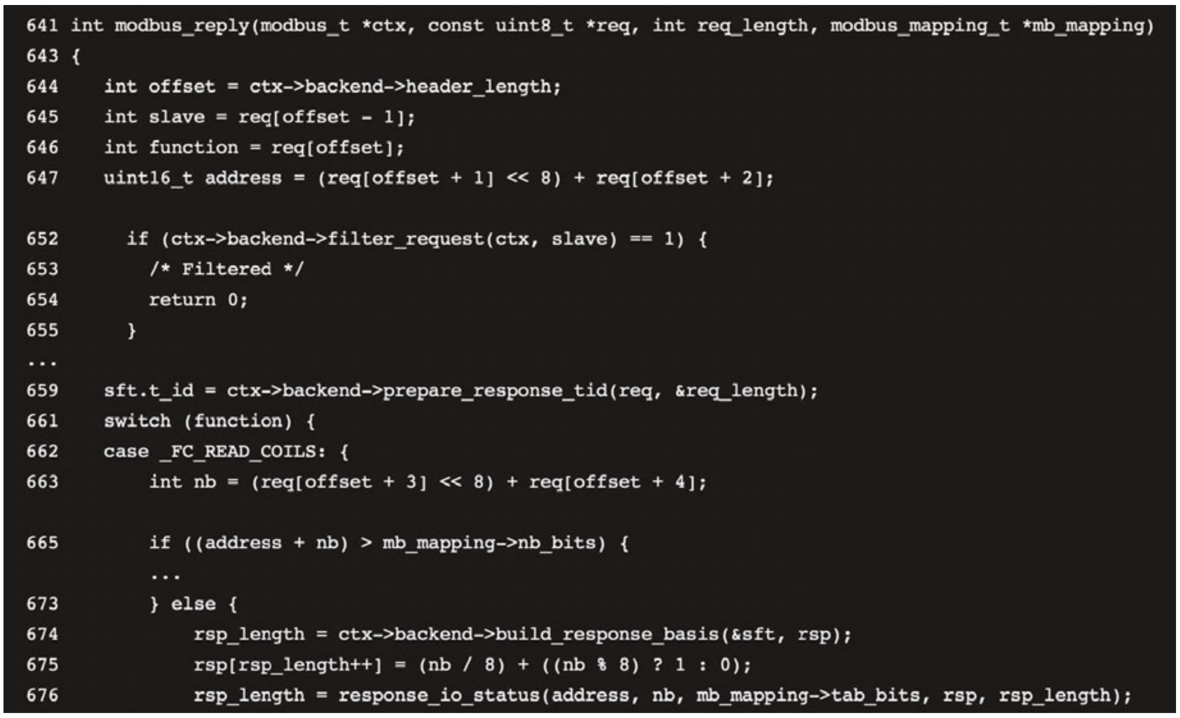
Protocol reverse engineering techniques are classified mainly into two categories: (1) network inference, which means that protocol reverse engineering is based on the analysis of traffic (i.e., exchanged messages) between two components; (2) application inference, which means that protocol reverse engineering is based on the analysis of the program itself. Our work belongs to the class of application inference.

In protocol reverse engineering, there exist two main challenging tasks: (1) The first task focuses on each individual protocol message and aims to identify the boundary of every single protocol field as well as the entire structure built on the fields. (2) The second task aims to identify the type attributes of protocol fields and then to understand protocol message semantics. The first task lays the foundation for protocol reverse engineering, whose accuracy and completeness significantly affect the following tasks. In addition, ICS protocols are designed to work in a time-sensitive network by exchanging messages between slave devices and master devices, where a single protocol message usually represents an individual function. We thus focus on the first task and leave the second one as the future work. Therefore, our question is that, given a single message received by a binary program, how can we effectively identify the boundary of every single protocol field completely.

3 System design

We provide our methodology based on the intuition that an individual field in a received message is handled by an individual BBL group in a binary application, where a BBL group is defined as the continuous basic block before a subroutine is called in a function's execution trace. For example, Fig. 2 shows a code snippet from a real-world Modbus protocol library, i.e., libmodbus (Stephane, 2020). Note that we provide the C source code to ease explanation, and in practical analysis, we target binary programs. In this snippet, the slave, function, and address variables representing corresponding fields are processed first in the modbus_reply function (lines 645, 646, 647, respectively). In line 652, the slave is processed in the filter_request function and in line 659, the Modbus transaction id (i.e., sft.t_id) is extracted by the prepare_response_tid function. Thereafter, the function code field is used again (lines 661–676) to decide which branch will be executed in the program. Additionally, we can know that the address (line 647) and nb (line 663) are composed of two sequential bytes, which means that each corresponding protocol field includes two bytes.

The goal of ICSPRF is to reverse engineer such an ICS protocol format automatically. By monitoring the execution trace of the application, we can record the mappings that include the processed protocol fields and the corresponding BBL groups, and then infer the borders of each field chunk in the message automatically.


```
641 int modbus_reply(modbus_t *ctx, const uint8_t *req, int req_length, modbus_mapping_t *mb_mapping)
643 {
644    int offset = ctx->backend->header_length;
645    int slave = req[offset - 1];
646    int function = req[offset];
647    uint16_t address = (req[offset + 1] << 8) + req[offset + 2];

652      if (ctx->backend->filter_request(ctx, slave) == 1) {
653        /* Filtered */
654        return 0;
655      }
...
659    sft.t_id = ctx->backend->prepare_response_tid(req, &req_length);
661    switch (function) {
662    case _FC_READ_COILS: {
663        int nb = (req[offset + 3] << 8) + req[offset + 4];

665        if ((address + nb) > mb_mapping->nb_bits) {
           ...
673        } else {
674            rsp_length = ctx->backend->build_response_basis(&sft, rsp);
675            rsp[rsp_length++] = (nb / 8) + ((nb % 8) ? 1 : 0);
676            rsp_length = response_io_status(address, nb, mb_mapping->tab_bits, rsp, rsp_length);
```
Fig. 2 Code snippet in modbus.c of libmodbus

Specifically, ICSPRF is interested in how and what field-specific trace information can be recorded and analyzed to extract the protocol format. Fig. 3 shows an architectural overview of ICSPRF, which has two main components: a program taint analysis engine and an offline log analyzer. Given a binary application to be analyzed, ICSPRF works as follows:

1. On receiving a protocol message, it first taints the data by instrumenting system calls that read data from a socket.

2. Once the bytes of a message are tainted, the propagation engine keeps track of their propagation at the byte level. Meanwhile, the taint analyzer logs each tainted byte and its position offset in the entire received message, the semantic-sensitive opcode and operands of relevant executed instructions, and the call stack of the execution trace.

3. With the collected information, the offline protocol log analyzer is invoked to identify and extract protocol fields in the message.

3.1 Dynamic taint analysis

There are two stages during the dynamic taint analysis of a program: taint initialization and taint propagation. Algorithm 1 shows our dynamic taint analysis framework. At the taint initialization stage, we intercept the system calls (e.g., recv and read) that read data from a remote socket by routine-grained instrumentation, taint the received data, and record every byte with its position offset in the entire message.

The taint propagation stage is focused on the application-level program images that process the received message. We track the taint propagation following our taint propagation policies (as shown in Tables 1 and 2) by instrumenting the chosen instructions (e.g., mov and lea) and functions (e.g., memcpy). Specifically, for a data movement instruction or function as shown in Table 1, we check whether the source operand is tainted. If yes, we taint the destination operand with the source operand's attributes, i.e., the tainted bytes in the source operand and the position offsets of the bytes in the original message. The operands can be a register or a memory location. If the source operand is not tainted, we unmark the destination operand. In Table 1, $T(X)$ means obtaining the tainted attributes of X. For arithmetic/logic instructions (e.g., add, and, or) as shown in Table 2, we merge or change their attributes according to the meanings of the instruction (e.g., for the add operation, the result is the union of the operands if they are both marked).

3.2 Logs

ICSPRF logs mainly two classes of execution context information: the call stack and the semantic-sensitive instructions (e.g., lea, mov, and cmp) that process tainted data. To acquire the call stack

Algorithm 1 Dynamic taint analysis framework

```
1: addr, len ← instrumenting taint introduction function;
2: if len is not empty then
3:     Taintinit(addr, len);
4: end if
5: if Img ∈ selected images then
6:     left_op, write_op ← instrumenting instructions in Img;
7: else
8:     left_op, write_op ← instrumenting data movement function called in Img;
9: end if
10: if isTainted(left_op)||isTainted(write_op) then
11:     Run taint, unmark, or merge according to our taint propagation policy;
12: end if
```

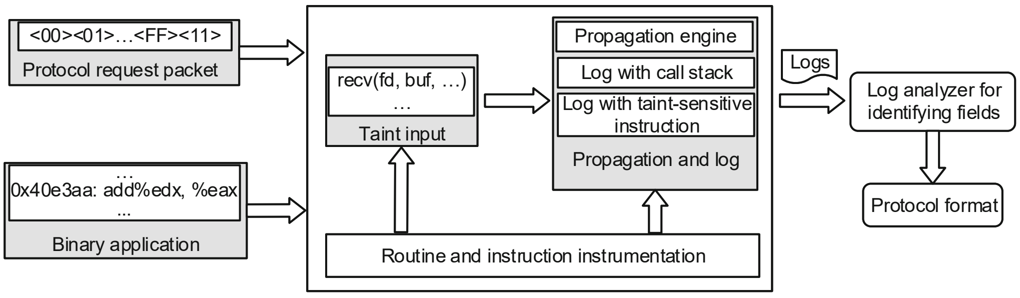


Fig. 3 ICSPRF overview

Table 1 Policy I: taint propagation policy of data movement operations

Operation type	Policy	Example(s)
reg_l ← reg_r	T(reg_w) = T(reg_r)	mov, esi, edi
reg_l ← mem_r	T(reg_w) = T(mem_r)	mov eax, dword ptr [rbp]
reg_l ← imm_r	T(reg_w) = false	mov eax, 0x1
mem_l ← reg_r	T(mem_w) = T(reg_r)	push rbp
mem_l ← mem_r	T(mem_w) = T(mem_r)	push qword ptr [rbp-0x4]
mem_l ← imm_r	T(mem_w) = false	mov qword ptr [rsp], 0x0
dst ← src	T(dst) = T(src)	memcpy(dst, src, size), memmove(dst, src, size)

Table 2 Policy II: taint propagation policy of arithmetic and logic operation instructions

Instruction	Examples	Policy
xor	xor edi, edi	Untaint(edi)
sub	sub esi, esi	Untaint(esi)
and	and eax, 0x0	Untaint(eax)
shl/shr	shl reg, 0x8	ShiftT(reg)
or/add	add eax, edx	MergeT(eax, edx)
and	and eax, 0xff	PartialSaveT(eax)

information, we need to traverse the stack frames by routine instrumentation. Specifically, in the entry of a function, we can obtain the return address inside the function frame and then further derive the call function from the return address. We also instrument a function at its exit to record the end of its execution. The vectors ⟨function flag, thread ID, flag, function_name, (start_addr, end_addr, return_addr)⟩ and ⟨function flag, thread ID, flag, function_name⟩ represent a function entry log and a function exit log, respectively. It is necessary to record both the entry and exit of a function to obtain a complete call chain. At the offline analysis stage, we scan the pairs of function entry/exit logs to reconstruct the call stack.

By instruction instrumentation, we can easily record the instruction context when it processes tainted bytes. We save this information in the vector ⟨instruction flag, address, disassembly, thread ID, offsets, memory_addr, values⟩, in which offsets mean the position offsets of the processed tainted bytes in the received message and values mean the values of the operands.

3.3 Border identification

At the stage of offline log analysis, we aim to identify the borders of the field chunks in a protocol message. First, we provide an algorithm to construct a hierarchical tree that demonstrates the call relationship and sequential relationship among all BBL groups that process tainted bytes. From this tree, we can infer individual fields based on several principles.

In this subsection, we provide the algorithm as shown in Algorithm 2. The log input works as the input of the algorithm. We give a detailed example of the log file in the supplementary materials (Fig. S1). In Algorithm 2, if a record item is a function-type item, it represents entering or exiting a subroutine (line 5). The current data node includes all tainted bytes processed by the previous BBL group. Thus, we add the data node to the current function node (lines 6–9). If a record item means an entry of a function (line 11), we add this new function node to the current one and set this new node as the current node. If this record means an exit of a function (line 16), we set the parent node of the current node as the current one. If a record item is an instruction-type item, we add it into a data node (lines 20–22). When this algorithm is finished, a hierarchical tree is generated.

In this tree, the nodes in the same parent node represent the sequentially executed BBL groups, in which the leaf nodes include the position offsets of the processed bytes, and its edges mean the hierarchical call of subroutines. We use the following principles of judgment to split the field chunks from the tree:

1. If a single byte or multiple discontiguous bytes are processed in an individual BBL group, they are considered as individual field chunks.

2. If multiple contiguous bytes are processed together in an individual BBL group, they are considered as a field chunk.

We give an example in Fig. 4. As we can see, the bytes at offsets 0, 1, and 6 of the received message are not processed in any BBL group, the byte at offset 7 is processed in a single BBL group, and the bytes at offsets [2, 3], [4, 5], [8, 9], and [10, 11] are processed together in individual BBL groups separately. From this tree, we can conclude that the field chunks in the received message can be split as [0, 1] [2, 3] [4, 5] [6] [7] [8, 9] [10, 11], in which the numbers represent the position offsets of the received message and the numbers in each pair of brackets represent that the bytes in the corresponding offsets are identified as those in a single protocol field chunk.

Algorithm 2 Hierarchical tree generation

```
Require: the log items logs.
Ensure: a hierarchical tree.
 1: root ← FunctionNode();
 2: node ← root;
 3: data ← DataNode();
 4: for each log ∈ logs do
 5:     if log.type == FUNCTION then
 6:         if data is not empty then
 7:             Add data → node;
 8:             data ← DataNode();
 9:         end if
10:         name ← log.name;
11:         if log.keyword == enter then
12:             newnode ← FunctionNode(name);
13:             Add newnode → node;
14:             node ← newnode;
15:         end if
16:         if log.keyword == exit then
17:             node ← node.parent;
18:         end if
19:     end if
20:     if log.type == INSTRUCTION then
21:         Add log.offsets → data;
22:     end if
23: end for
24: return root;
```

4 Evaluation

We set our experiment involving 12 protocol messages from six different open-source applications. Table 3 shows the list of the protocols. Specifically, we choose two third-party libraries, i.e., freemodbus (Cwalter-at, 2020) and libmodbus of the Modbus protocol, two applications (Airpig2011, 2020; MZ Automation GmbH, 2020) of the IEC104 protocol, one application (Green Energy Corporation, 2020) of the DNP3 protocol, which are implemented according to standard specifications, and a third-party implementation of the Siemens-owned s7comm protocol (Nardella, 2020). The third-party libraries are obtained by compiling the source code with a default configuration on a 64-bit Ubuntu16.04 operating system platform.

Table 3 Six open-source ICS protocol implementations for evaluation of ICSPRF

Protocol	Application	Size	DTA time	Offline time
Modbus	libmodbus-3.0.0	92.5×10^3	700 ms	≤3 s
	freemodbus	49.6×10^3	600 ms	≤3 s
IEC104	IEC104	43.0×10^3	600 ms	≤3 s
	lib60870-2.20	333.9×10^3	2400 ms	≤3 s
DNP3	gec-dnp3	27.7×10^6	14.5 s	≤3 s
s7comm	snap7-full-1.2.1	332.3×10^3	2400 ms	≤3 s

DTA: dynamic taint analysis. The unit of size is byte

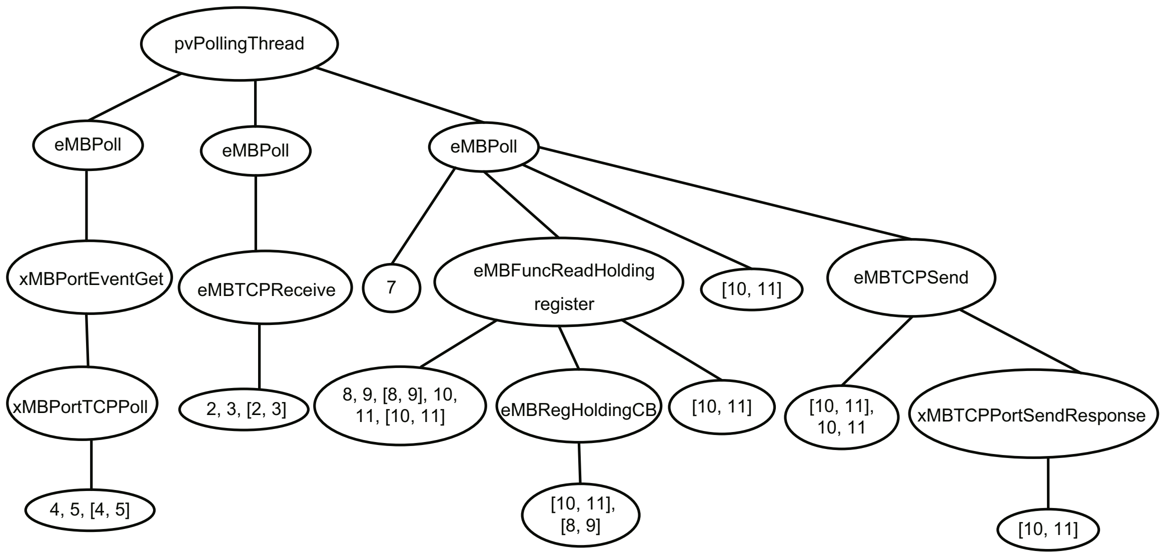


Fig. 4 An example of the hierarchical tree generated from the analysis of freemodbus

Overall, ICFPRF has good performance for all evaluated applications as shown in Table 3. At the taint analysis stage, the dynamic taint analysis (DTA) time cost increases with the increase in the application size, but it is acceptable (at most tens of seconds). Its efficiency benefits from the lightweight instrumentation of our approach, and we leave further analysis for the offline analysis stage. The offline log analysis stage is also efficient and relatively stable. The offline time cost is not more than 3 s for all the evaluated applications.

To evaluate the effectiveness of ICSPRF in terms of field chunk identification, we compare our results with those of the latest version of a popular network protocol analyzer, Wireshark. We use SF to represent the identified single field chunks and |SF| to represent their quantity. R_e represents the match ratio, which is obtained by calculating the number of Wireshark-identified fields and the number of fields automatically identified by other methods. $\overline{R}_\mathrm{e}$ represents the average R_e of all the evaluated messages. If a single field chunk has been split into multiple chunks by ICSPRF, we regard each of the chunks as an overly fine-grained field. We count the number of overly fine-grained fields as $|\mathrm{SF_o}|$. Note that the quantity of a single field chunk in |SF| always counts as 1, no matter how many overly fine-grained chunks are split from a single field chunk. Similarly, if multiple field chunks identified by Wireshark are combined into a single chunk by ICSPRF, we regard this chunk as a coarse-grained field. The quantity of a coarse-grained field is also counted as 1 in |SF|. We count the number of coarse-grained fields as $|\mathrm{SF_c}|$.

Table 4 shows the detailed results. For all the evaluated projects, the match ratio R_e is no less than 82% and we obtain $\overline{R}_\mathrm{e} = 94.3\%$ with small $|\mathrm{SF_o}|$ and $|\mathrm{SF_c}|$. In contrast, for the same metric, AutoFormat achieves a lower accuracy than ICSPRF (88.5% for all their evaluated protocols and 80.0% for binary-based protocols). For certain examples, e.g., freemodbus, IEC104, IEC60870, and DNP3, we even obtain R_e=100% with zero $|\mathrm{SF_o}|$ and $|\mathrm{SF_c}|$. In the following, we demonstrate the detailed results of each protocol implementation.

1. Modbus

In this experiment, we choose two different implementations of the Modbus protocol (i.e., libmodbus and freemodbus) and evaluate ICSPRF by sending three different request messages: read registers, multiple write and read registers, and multiple write coils. Detailed results are shown in Table 5. ICFREF fails only to split the transaction id and protocol id fields on libmodbus. The reason is that these two fields are ignored and not handled in the libmodbus implementation program. Interestingly, ICSREF splits the data fields into more fine-grained fields in both libmodbus and freemodbus implementation programs; in contrast, Wireshark combines them into a single chunk, as shown in Fig. 5. By a closer investigation of the Modbus protocol, the mixed combined four bytes in Wireshark represent different write values of two individual registers. We

Table 4 Identified field chunk comparison between Wireshark and ICSPRF

Project	Message type	Length	Wireshark	ICSPRF					AutoFormat	
			Size	|SF|	R_e	$\|\mathrm{SF_o}\|$	$\|\mathrm{SF_c}\|$	$\overline{R}_\mathrm{e}$	$\overline{R}_\mathrm{e}(A)$	$\overline{R}_\mathrm{e}(B)$
	Read registers	12	7	6/7	86%	0	1			
libmodbus	Write and read registers	21	11	10/11	91%	**0**	1			
	Write coils	14	7	6/7	86%	**0**	1			
	Read registers	12	7	7/7	100%	0	0			
freemodbus	Write and read registers	21	11	11/11	100%	**0**	0			
	Write coils	14	7	7/7	100%	**0**	0	94.3%	88.5%	80.0%
IEC104	U-format	6	4	4/4	100%	0	0			
IEC60870	U-format	6	4	4/4	100%	0	0			
DNP3	Read class	18	11	11/11	100%	0	0			
	Read SZL	33	22	18/22	82%	0	3			
Snap7	List blocks	31	22	20/22	91%	0	1			
	Write variables	36	26	25/26	96%	1	1			

$\overline{R}_\mathrm{e}(A)$ and $\overline{R}_\mathrm{e}(B)$ of AutoFormat represent the match ratios for all the evaluated protocols and for the binary-based protocols, respectively. The units of length and size are both byte consider ICSPRF to be more accurate than Wireshark in this part. Therefore, we count $|SF_o|$ of libmodbus and freemodbus as 0 (the bold ones in Table 4), although ICSPRF splits the data field into two single fields compared with the identification result by Wireshark (marked as overly fine-grained in Table 5).

2. s7comm

We choose an open-source implementation of the Siemens-owned s7comm protocol, which is more complex than other open ICS protocols and includes more than 20 fields in its request message. In our experiment, as shown in Table 6, ICSPRF does not identify the first two bytes in the message because they are not handled by the program. ICSREF also incorrectly splits the address field into two fields, because the compilation optimization misleads ICSPRF. Specifically, in the source code, the snap7 program processes the address field in the following code snippet: Start=SwapDWord(*PAdd & 0xFFFFFF00); the pointer *PAdd points to the address field of the protocol message. The C code sentence means that the start address comes from three bytes of *PAdd (i.e., offsets 27–29 in the message). However, in the binary application, the source code is compiled into the following two assemble codes: mov edi, [r14+8]; dil, 0. ICSPRF can only identify that an integer (i.e., offsets 27–30) and a single byte (i.e., offset 30) are processed separately. Thus, ICSPRF considers offset[30] as an individual field chunk.

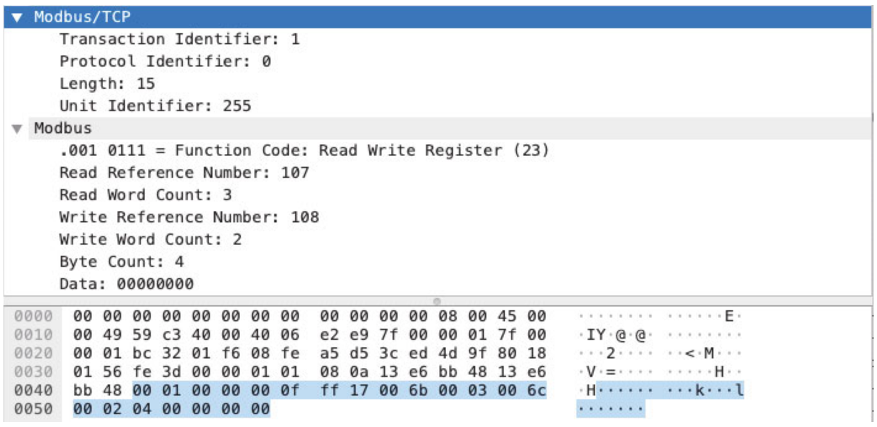
5 Related works

In this section, we introduce closely related works that aim at protocol reverse engineering and

Table 5 Field chunk identification comparison of libmodbus and freemodbus in the write and read register request messages between Wireshark and ICSPRF

Field name	Wireshark	ICSPRF for libmodbus		ICSPRF for freemodbus	
	Size	Size	Match	Size	Match
Transaction id	2	4	Coarse-grained	2	✓
Protocol id	2		Coarse-grained	2	✓
Length	2	2	✓	2	✓
Unit id	1	1	✓	1	✓
Function code	1	1	✓	1	✓
Read number	2	2	✓	2	✓
Read count	2	2	✓	2	✓
Write number	2	2	✓	2	✓
Write count	2	2	✓	2	✓
Byte count	1	1	✓	1	✓
Data	4	2	Overly fine-grained	2	Overly fine-grained
		2	Overly fine-grained	2	Overly fine-grained

The units of length and size are both byte



```
▼ Modbus/TCP
      Transaction Identifier: 1
      Protocol Identifier: 0
      Length: 15
      Unit Identifier: 255
▼ Modbus
      .001 0111 = Function Code: Read Write Register (23)
      Read Reference Number: 107
      Read Word Count: 3
      Write Reference Number: 108
      Write Word Count: 2
      Byte Count: 4
      Data: 00000000

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 49 59 c3 40 00 40 06  e2 e9 7f 00 00 01 7f 00   ·IY·@·@· ········
0020  00 01 bc 32 01 f6 08 fe  a5 d5 3c ed 4d 9f 80 18   ···2···· ··<·M···
0030  01 56 fe 3d 00 00 01 01  08 0a 13 e6 bb 48 13 e6   ·V·=···· ·····H··
0040  bb 48 00 01 00 00 00 0f  ff 17 00 6b 00 03 00 6c   ·H······ ···k···l
0050  00 02 04 00 00 00 00                               ·······
```

Fig. 5 Wireshark for Modbus

Table 6 Field chunk identification comparison of snap7 in the write variable request message between Wireshark and ICSPRF

Field name	Wireshark	ICSPRF	
	Size	Size	Match
Version	1	2	Coarse-grained
Reserved	1		Coarse-grained
TPKT length	2	2	✓
COTP length	1	1	✓
PDU type	1	1	✓
Destination	1	1	✓
Protocol ID	1	1	✓
Job	1	1	✓
Reserved	2	2	✓
PDU reference	2	2	✓
Parameter length	2	2	✓
Data length	2	2	✓
Function code	1	1	✓
Count	1	1	✓
Variable specification	1	1	✓
Address length	1	1	✓
Syntax ID	1	1	✓
Transport size	1	1	✓
Length	2	2	✓
DB number	2	2	✓
Area	1	1	✓
Address	3	2	Overly fine-grained
		1	Overly fine-grained
Return code	1	1	✓
Transport size	1	1	✓
Length	2	2	✓
Data	1	1	✓

The unit of size is byte

compare them with ICSPRF.

In the literature (Denton et al., 2017; Senthivel et al., 2017), researchers achieved successful ICS protocol reverse engineering that relies on manual effort, which is slow and costly. Our work provides new automatic techniques that can be used to reduce the cost and time associated with these projects.

From a perspective of automatic protocol reverse engineering, network inference and application inference are two main approaches used in previous works. Among them, PI (Beddoe, 2012), Discoverer (Cui et al., 2007), and Netzob (Bossert et al., 2014) are based on network inference, using sequence alignment algorithms and clustering algorithms and aiming at extracting protocol format from collected network traffics. Their correctness and completeness rely on manually collected network traffics, which may significantly limit the accuracy of the extracted protocol formats. Moreover, their effectiveness on binary-based protocol reverse engineering has not been evaluated.

Polyglot (Caballero et al., 2007) and AutoFormat (Lin et al., 2008) are based on application inference and most closely relate to our work, sharing the key insight that the way in which a protocol is implemented to recognize and handle protocol messages provides a wealth of information about the protocol format. ICSPRF differs from them in its way of identifying field chunks in the received message. Using the fine-grained execution context (i.e., BBL groups), ICSPRF collects and analyzes run-time execution context information to infer the borders, achieving better accuracy and completeness in extracting protocol format.

6 Conclusions

We have presented ICSPRF, a framework for automatic protocol reverse engineering. ICSPRF is based on the insight that a protocol program has chunk-feature behaviors in its execution contexts. By instrumenting and monitoring the program execution, we can obtain the taint propagation process among BBL groups of the execution trace and use it to infer protocol fields. We have implemented a prototype of ICSPRF and evaluated it with a variety of protocol messages from six real-world protocol implementations. Our experimental results showed that ICSPRF achieved higher accuracy (on average, a 94.3% match ratio for the binary-based ICS protocol) in protocol field identification; in comparison, AutoFormat achieved an 88.5% match ratio for all evaluated protocols and only 80.0% for binary-based protocols.

Contributors

Rongkuan MA designed the research. Rongkuan MA and Hao ZHENG designed the supporting algorithms and implemented the software. Rongkuan MA drafted the paper. Jingyi WANG and Mufeng WANG helped organize the paper. Qiang WEI and Qingxian WANG led the research planning and execution. Rongkuan MA, Qiang WEI, and Qingxian WANG revised and finalized the paper.

Compliance with ethics guidelines

Rongkuan MA, Hao ZHENG, Jingyi WANG, Mufeng WANG, Qiang WEI, and Qingxian WANG declare that they have no conflict of interest.

References

Airpig2011, 2020. IEC104. https://github.com/airpig2011/IEC104 [Accessed on Nov. 20, 2020].

Beddoe MA, 2012. Network Protocol Analysis Using Bioinformatics Algorithms. https://raw.githubusercontent.com/wiki/unmarshal/protocol-informatics/pi.pdf

Bossert G, Guihéry F, Hiet G, 2014. Towards automated protocol reverse engineering using semantic information. 9th ACM Symp on Information, Computer and Communications Security, p.51-62. https://doi.org/10.1145/2590296.2590346

Caballero J, Yin H, Liang ZK, et al., 2007. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. Proc 14th ACM Conf on Computer and Communications Security, p.317-329. https://doi.org/10.1145/1315245.1315286

Chang Y, Choi S, Yun JH, et al., 2017. One step more: automatic ICS protocol field analysis. Int Conf on Critical Information Infrastructures Security, p.241-252. https://doi.org/10.1007/978-3-319-99843-5_22

Choi K, Son Y, Noh J, et al., 2016. Dissecting customized protocols: automatic analysis for customized protocols based on IEEE 802.15.4. Proc 9th ACM Conf on Security & Privacy in Wireless and Mobile Networks, p.183-193. https://doi.org/10.1145/2939918.2939921

Cui WD, Kannan J, Wang HJ, 2007. Discoverer: automatic protocol reverse engineering from network traces. Proc 16th USENIX Security Symp, p.199-212.

Cwalter-at, 2020. Freemodbus. https://github.com/cwalter-at/freemodbus [Accessed on Nov. 20, 2020].

Denton G, Karpisek F, Breitinger F, et al., 2017. Leveraging the SRTP protocol for over-the-network memory acquisition of a GE Fanuc Series 90-30. *Dig Invest*, 22:S26-S38. https://doi.org/10.1016/j.diin.2017.06.005

Fang CR, Qi YF, Cheng P, et al., 2020. Optimal periodic watermarking schedule for replay attack detection in cyber–physical systems. *Automatica*, 112:108698. https://doi.org/10.1016/j.automatica.2019.108698

Fioraldi A, D'Elia DC, Coppa E, 2020. WEIZZ: atomatic grey-box fuzzing for structured binary formats. Proc 29th ACM SIGSOFT Int Symp on Software Testing and Analysis, p.1-13. https://doi.org/10.1145/3395363.3397372

Green Energy Corporation, 2020. gec-dnp3. https://github.com/gec/dnp3 [Accessed on Nov. 20, 2020].

Ji R, Wang J, Tang CJ, et al., 2017. Automatic reverse engineering of private flight control protocols of UAVs. *Secur Commun Netw*, 2017:1308045. https://doi.org/10.1155/2017/1308045

Lin ZQ, Jiang XX, Xu DY, et al., 2008. Automatic protocol format reverse engineering through context-aware monitored execution. Proc 15th Symp on Network and Distributed System Security, p.29-43.

Luo ZX, Zuo FL, Shen YH, et al., 2020. ICS protocol fuzzing: coverage guided packet crack and generation. 57th ACM/IEEE Design Automation Conf, p.1-6. https://doi.org/10.1109/DAC18072.2020.9218603

MZ Automation GmbH, 2020. lib60870. https://github.com/mz-automation/lib60870 [Accessed on Nov. 20, 2020].

Nardella D, 2020. Snap7. https://sourceforge.net/projects/snap7/files/1.2.1/ [Accessed on Nov. 20, 2020].

Senthivel S, Ahmed I, Roussev V, 2017. SCADA network forensics of the PCCC protocol. *Dig Invest*, 22:S57-S65. https://doi.org/10.1016/j.diin.2017.06.012

SharkFest, 2020. Wireshark. https://www.wireshark.org/ [Accessed on Nov. 20, 2020].

Stephane, 2020. libmodbus. https://github.com/stephane/libmodbus [Accessed on Nov. 20, 2020].

Yang ZY, He L, Cheng P, et al., 2020. PLC-sleuth: detecting and localizing PLC intrusions using control invariants. 23rd Int Symp on Research in Attacks, Intrusions and Defenses, p.333-348.

List of electronic supplementary materials

Fig. S1 A log example recorded by ICSPRF when monitoring the execution of processing a request by freemodbus