Automatic Identification of Industrial Control Network Protocol Field Boundary Using Memory Propagation Tree

Chen Kai[1(✉)], Zhang Ning[2], Wang Liming[1], and Xu Zhen[1]

[1] State Key Laboratory of Information Security,
Institute of Information Engineering, Chinese Academy of Sciences,
E-park C1, No. 65 Xingshikou Road, Haidian District, Beijing 100195, China
chenk@iie.ac.cn
[2] State Grid Jibei Electric Company, Hebei, China

Abstract. The knowledge of protocol specification, especially protocol field boundary, is invaluable for addressing many security problems, such as intrusion detection. But many industrial control network (ICN) protocols are closed. Closed protocol reverse engineering has often been a time-consuming, tedious and error-prone process. Some solutions have recently been proposed to allow for automatic protocol reverse engineering. But their prerequisites, e.g. assuming the existence of keywords or delimiters in protocol messages, limit the scope of their efforts to parse ICN protocol messages. In this paper, we present AutoBoundary that aims at automatically identifying field boundaries in an ICN protocol message. By instrumenting and monitoring program execution, AutoBoundary can obtain the execution context information, and build a memory propagation (MP) tree for each message byte. Based on the similarity between MP trees, AutoBoundary can identify protocol field boundaries, automatically. The intuition behind AutoBoundary makes it suitable for ICN protocols, which have the characteristics of no delimiter, no keyword, and no complex hierarchical structure in the message. We have implemented a prototype of AutoBoundary and evaluated it with 62 ICN protocol messages from 4 real-word ICN protocols. Our experimental results show that, for the ICN protocols whose fields are byte-aligned, AutoBoundary can identify field boundaries with high accuracy (100% for Modbus/TCP, 100% for Siemens S7, and 94.7% for ISO 9506).

Keywords: Industrial control network
Protocol field boundary identification · Memory propagation tree

1 Introduction

For industrial control network (ICN), the knowledge of application-level protocol specifications is invaluable for addressing many security problems. Protocol specifications are often required for intrusion detection and firewall systems to

D. Naccache et al. (Eds.): ICICS 2018, LNCS 11149, pp. 551–565, 2018.
https://doi.org/10.1007/978-3-030-01950-1_32

perform deep packet inspection [3,17]. For ICN management, protocol specifications can be used to identify protocols and analyze network traffic. They also allow the automatic generation of protocol fuzzers when performing the black-box testing [16]. Of course, many ICN protocol specifications can be obtained from authority documents directly, such as Modbus. But many ICN protocols are unknown, undocumented or proprietary, such as Siemens S7. For the closed protocol, protocol reverse engineered manually always means time-consuming and error-prone. In this case, the specification could only be specified through automatic protocol reverse engineering.

A key work of protocol reverse engineering is to identify the field boundaries. Some solutions have recently been proposed to allow for automatic protocol reverse engineering, including identifying field boundaries. Most of them are effective and efficient, when the prerequisites are met. The prerequisites include: (1) the existence of keywords or delimiters in protocol messages; (2) utilizing loops and comparison operations to parse protocol messages within the software binary; (3) getting key information ahead of time, e.g., IP address or host name, and so on. On the other hand, the syntactic structures of many ICN protocols have the characteristics of no delimiter, no keyword, and no hierarchical structure, such as Modbus. In this paper, we present AutoBoundary, a new approach that aims at automatically identifying field boundaries in an ICN protocol message. AutoBoundary is based on the key observation that *bytes belonging to the same protocol field of a message have the same propagation traces in the memory, due to they are typically handled together.* The intuition behind AutoBoundary does not depend on delimiter, keyword, or hierarchical structure. So it is more suitable for ICN protocols. By dynamically analyzing program execution, we record the address for a message byte once it propagates from one place to another. At last, all address records of a message byte compose a memory propagation tree. A n-byte message results n memory propagation trees. Through comparing between memory propagation trees, we can decide whether two message bytes belong to the same protocol field or not. Further, based on the similarity between memory propagation trees, we can identify the field boundaries of a protocol message. We have implemented a proof-of-concept prototype and evaluated it with 62 ICN protocol message from 4 ICN protocols.

The contributions of this paper are the following: (1) We present a novel approach to analyze the movement trace of message bytes in the memory. We use memory propagation (MP) tree as storage structure to record all movement traces of a message byte, and describe the detail way how to compare between MP trees. The comparison result embodies the similarity between MP trees. (2) We present AutoBoundary, an MP-tree-based approach to identify the field boundaries in an ICS protocol message. (3) We applied our techniques to a set of real-world applications that implement ICN protocols such as Modbus/TCP, IEC 60870-5-104, ISO 9506, and Siemens S7. Our results show that AutoBoundary can automatically identify field boundaries.

2 Goal and Assumption

There are three essential components in an application-level protocol specification: protocol syntax, protocol FSM and protocol semantics. Inferring protocol syntax, including identifying field boundary, lays the foundation for automatic protocol reverse engineering. In this paper, we focus on identifying the field boundaries. Our goal is to design an algorithm that, given one message of an ICN protocol and an application that can process this message, recovers the boundaries of fields.

As mentioned above, we assume that the application, which can parse the protocol message, could be obtained. Though these appliances that run on very "special" hardware seem to be hardly obtained, it is not always an unsolvable problem. We believe that some cutting-edge technologies (e.g., softPLC, virtualization, and digital twins) would make it possible.

3 System Design

The intuition behind AutoBoundary is simple but effective: "Bytes belonging to the same protocol field of a message have the same propagation traces in the memory, due to they are typically handled together." As such, the propagation traces of each message byte can be compared to uncover field boundaries. AutoBoundary is interested in how memory propagation information can be collected and analyzed to identify field boundaries. It has three processing stages: (1) execution monitor, (2) MP tree generation, and (3) field boundary identification.

3.1 Stage 1: Execution Monitor

In the execution monitor stage, an ICN protocol message is sent to an application that "understands" the protocol that we are interested in, such as a server program implementing a particular ICN protocol. By monitoring application execution, we can intercept the network-related system calls (e.g., sys_socket), and mark the message received as tainted data. Moreover, throughout the message processing life-time, we instrument all instructions that operate on the tainted data to record propagation traces. More specially, for a data movement instruction, we check whether the source operand is tainted. If yes, we will mark the destination operand, which can be a register or a memory location, as tainted data; If no, we will simply unmark the destination operand. At the same time, we record the instruction address and the addresses of both the source operand and destination operand, with the format: "$addr_{ins} : addr_{src} \rightarrow addr_{dst}$". If an instruction has two source operands, we will union of their marks, and record with the format: "$addr_{ins} : addr_{src1} + addr_{src2} \rightarrow addr_{dst}$". Similar to previous systems that use dynamic taint analysis, we establish a relationship between a particular message byte and a location in memory (or a register). We reference interested readers to related literature such as [4,10,18].

As a result of the monitoring process, memory propagation records are produced for each ICN protocol message. They contain all operations that have one tainted operands at least.

3.2 Stage 2: Memory Propagation Tree Generation

To organize memory propagation records efficiently and make it possible for mathematical calculation, we define a new structure and name it as memory propagation tree, which is described as Definition 1. Each byte from an ICN protocol message has one, and only one, corresponding MP tree. In the rest of this section, we discuss the way how to build MP trees from memory propagation records, calculate branch contribution for an MP tree, and compress an MP tree, within stage 2.

Definition 1. *A Memory Propagation (MP) tree is a data structure made up of nodes and edges without having any cycle. An MP tree consists of a root node and potentially many levels of additional nodes that form a hierarchy. The root node represents the initial memory location of a byte from an ICN protocol message. The intermediate and leaf nodes represent the locations where this byte has appeared during its propagation process. An edge means propagating from a location (i.e., parent node) to another (i.e., child node).*

MP Tree Construction. By monitoring the network-related system calls, the buffer that contains the received protocol message is determined. For each byte in the buffer, AutoBoundary creates a blank MP tree, adds a root node into the MP tree, and sets two properties on the root node. One is location property that includes the byte address. The other is growth property that indicates whether a new branch can grow from this node. Then, as shown in Algorithm 1, AutoBoundary repeatedly reads a record from the memory propagation records generated in Sect. 3.1, and tries to insert a new child node and a new edge that is from $record.addr_{src}$ to $record.addr_{dst}$ into the MP tree mpt_i, by using the function *MPtree-InsertEdge()*. If there is an existing node $node_i$, whose location property value is equal with $record.addr_{src}$ and growth property value is "enable", the node and edge can be insert into mpt_i. In this case, AutoBoundary creates a new node $node'$, sets its location property value to $record.addr_{dst}$, and sets its growth property values to "enable". A new edge that is from $node_i$ to $node'$ is inserted into mpt_i. On the other hand, if there is no such node, AutoBoundary checks whether this memory propagation record would have impact on mpt_i. If $record.addr_{dst}$ is the same with any nodes' location property value, AutoBoundary modifies the growth property values of these nodes to "disable". The value "disable" means that a new branch cannot grow from such nodes, namely, they cannot be a parent of a new node. It's important to note that, in the real-world application, after a tainted memory or register is overwritten by other non-tainted data, we should no longer keep tracks of its propagation. The growth property attached to the node helps us to decide whether to keep tracks of the node's propagation in the future.

Algorithm 1. MPtree-Gen

Input. $[buf]$: the buffer contains the protocol message
$[records]$: the memory propagation records generated in Sect. 3.1
Output. $[mpt_1, mpt_2, \cdots, mpt_n]$: MP trees, n means the length of buf

```
for each byte_i in buf do
    create a blank MP tree mpt_i for byte_i
    create root node rootNode_i
    set rootNode_i.locationProperty =getAddress(byte_i)
    set rootNode_i.growthProperty = enable
    insert rootNode_i into mpt_i
    for read a record from records do
        MPtree-InsertEdge(record.addr_src, record.addr_dst, mpt_i)
    end for
end for
Return (mpt_1, mpt_2, ..., mpt_n);
```

Branch Contribution Calculation. The node that has no child is a leaf node. A branch consists of a root node, a leaf node, and all of nodes and edges between them. The branch length is defined in Definition 2.

Definition 2. *The length of a branch is the number of nodes, which belong to the branch. Term "longer" has the same meaning as "with more nodes".*

Definition 3. *Branch contribution is used to quantify weigh values of branches with different length. It embodies the contribution degree of a branch during the process of identifying field boundaries. It is a decimal number between* 0 *and* 1. 0 *means that the branch has no contribution to filed identification procedure. In contrast, being close to* 1 *means that the branch has much impact on filed identification procedure.*

For an MP tree, are the weight values of branches with different length the same? Of course not. The longer a branch is, the wider a byte propagates. A wider propagation always means that the byte has been processed by more instructions, while each instruction can partially reflect characteristics of the byte. Therefore, a longer branch usually provides more help to identify field boundaries. To quantify weight values for branches with different length, we introduce the definition of branch contribution as Definition 3. For the ith branch, its contribution $cont_i$ can be calculated by (1), where n is the total number of branches, and the function $len()$ gets the length of a branch. The contribution degree is attached to every branch as an additional property.

$$cont_i = \frac{len(branch_i)}{\sum_{j=1}^{n} len(branch_j)}, 1 \leqslant i \leqslant n \tag{1}$$

MP Tree Compression. If the length of two branches are equal, and the nodes residing in the same level have the same properties (i.e., location and

growth), we say that these two branches are the same. AutoBoundary does not forbid establishing multiple same branches during the process of MP tree generation. The redundancy branches reduce the efficiency of compare operation in the next stage. To mitigate the problem, AutoBoundary compresses the MP tree by merging the redundancy branches. The compressing approach used by AutoBoundary is simple and easy, but efficient and effective. For the redundancy branches, first rip out all but one, and then add the contribution of discarded branches into the remaining one.

3.3 Stage 3: Field Boundary Identification

Our field boundary identification relies on clustering message bytes. To cluster bytes, we need to invoke both branch comparison and tree similarity calculation. In this section, we first explain these procedures before describing how to divide clusters.

Branch Comparison. To find similar branches across different MP trees, we need a proper approach to compare branches. As Sect. 3.2 describes, each branch consists of many nodes. Therefore, to compare two branches, we align their nodes by using a customized version of sequence alignment algorithm. And the score gotten by aligned nodes represents the result of branch comparison.

We refer to our approach for aligning nodes as *node-based sequence alignment algorithm*. The key observation behind our approach is that, while sequence alignment algorithm [14] cannot be used for comparing branches directly, it can be used to align nodes by leveraging the node's properties (i.e., location and growth property) generated in the MP tree construction phase. In the node-based sequence alignment algorithm, we claim two aligned nodes are matched if they have the same growth property and the distance between their location is smaller than the size of a variable of type char (i.e., 1 byte). For instance, knowing a growth-enabled node N of a branch is placed in a particular location necessitates that its counterpart N' of another branch is also growth-enabled and next to N for these two nodes to be considered a match. We allow gaps in the node-based sequence alignment algorithm. In addition to using gap penalties to control gaps, we introduce extra constraints to make it more suitable for branch comparison. First, a node placed in registers is allowed to align with gaps. This constrain is for handling the case of coalescing multiple registers (i.e., EAX and EDX) to perform one computation. Second, a node placed in memory is allowed to align with gaps, but it must be imposed heavy penalty – the gap penalty is typically double. This constraint is for handling the case that string functions are used to parse fields. For example, when using the function "strncmp" to parse fields, the front part of the field may be handled more times than the rear part, which results in longer branches for the front part.

After aligning nodes, the node-based sequence alignment algorithm outputs a score for a pair of branches. This score quantifies the result of branch comparison. Since AutoBoundary does not focus on the absolute value of the score, it is insensitive to the scoring system (sub-scores for match, mismatch and gap) used by sequence alignment algorithm.

MP Tree Similarity Calculation. Let two target MP tree as mpt_1 and mpt_2. To calculate the similarity between them, we need to find the matched branches and accumulate contribution degrees for each tree, respectively.

Definition 4. *For a given branch from an MP tree, comparing it with all branches from another MP tree (by using branch comparison approach), the corresponding branch is defined as which one outscores others.*

Definition 5. *For a given branch and its corresponding branch, if the score for branch comparison exceeds a certain threshold, we claim they are matched.*

$$\begin{aligned} threshold &= \overline{len} \times fac \times sScore_{ma} + \overline{len} \times (1 - fac) \times sScore_{mi}, \\ \overline{len} &= \frac{len(branch) + len(branch')}{2} \end{aligned} \tag{2}$$

Firstly, we deal with mpt_1. Travel every branch br_i from mpt_1, we search its corresponding branch br_i' in mpt_2. We give the definition of corresponding branch in Definition 4. And then, check whether the branch br_i and its corresponding branch br_i' are matched. The definition of matched branch is given in Definition 5. As what described in it, a threshold is required to decide whether they are matched. This threshold can be obtained by (2), where $\overline{len}$ is the mean value of the branch length, $sScore_{ma}$ and $sScore_{mi}$ are the sub-scores for matched and mismatched nodes respectively, and fac is an adjustment factor whose range is from 0.5 to 1 – the value of fac means the least percentage of matched nodes (for Modbus/TCP, IEC 60870-5-104, ISO 9506 and S7, we suggest setting fac to 0.75). After that, if br_i and br_i' are matched, accumulate the contribution degree for mpt_1. Algorithm 2 describes this process in detail. It traverses all branches in mpt_1, to find the corresponding branch in mpt_2. If the comparison score between branch br_i and its corresponding branch br_i' is larger than the threshold, namely branches are matched, accumulate the contribution degree of br_i into the total contribution degree $matchContribution$. At the end, the accumulated contribution degree of mpt_1 is obtained.

Secondly, dispose mpt_2 with the same procedure as mpt_1, but reverse roles of two MP trees. Travel all branches of mpt_2, search the corresponding branch in mpt_1, and accumulate the contribution degree for mpt_2.

Algorithm 2. AutoBoundary-AccumulateContribution

Input. [mpt_1]: the first MP tree; [mpt_2]: the second MP tree
Output. [$matchContribution$]: accumulated contribution degrees for mpt_1

```
for each branch br_i in mpt_1 do
    (br_i', score) = AutoBoundary-FindCorrespondingBranch(br_i, mpt_2)
    if score > getThreshold(br_i, br_i') then
        matchContribution += getContribution(br_i)
    end if
end for
Return (matchContribution);
```

Algorithm 3. AutoBoundary-CalculateTreeSimilarity

Input. [mpt_1]: the first MP tree; [mpt_2]: the second MP tree
Output. [*similarity*]: the similarity between input MP trees
1: $matchCont_1$ = AutoBoundary-AccumulateContribution(mpt_1, mpt_2)
2: $matchCont_2$ = AutoBoundary-AccumulateContribution(mpt_2, mpt_1)
3: $similarity = (matchCont_1 \times getLength(mpt_1) + matchCont_2 \times getLength(mpt_2))$
4: $similarity = similarity/(getLength(mpt_1) + getLength(mpt_2))$
5: Return (*similarity*);

At last, calculate the similarity between two MP trees, based on their contribution degrees. As Algorithm 3 described, we obtain the result similarity through merging two accumulated contribution degrees.

Message Byte Clustering. Message byte clustering is an iterative process, from the first message byte to the last one. As shown in Algorithm 4, each iteration handles two adjacent message bytes. AutoBoundary finds MP trees for these two bytes, and then calculates the similarity between MP trees through the approach described above.

Algorithm 4. AutoBoundary-ClusterByte

Input. [*message*]: the ICN protocol message, including n bytes
[$mpt_1, mpt_2, ..., mpt_n$]: MP trees, one tree mapping one byte
Output. [*cluster*]: the result clusters
1: **for** i from 1 to n **do**
2: mpt_i = getMPTree($byte_i$) // $byte_i$ means ith byte in *message*
3: mpt_{i+1} = getMPTree($byte_{i+1}$)
4: $similarity_i$ = AutoBoundary-CalculateTreeSimilarity(mpt_i, mpt_{i+1})
5: **end for**
6: **for** i from 1 to $n-1$ **do**
7: k =sizeof(short)
8: $pre = (1 > i - k/2)?1 : (i - k/2)$
9: $post = (i + k/2 > n - 1)?(n - 1) : (i + k/2)$
10: **for** j from pre to $post$ **do**
11: $neighbourMean$ += $similarity_j$
12: **end for**
13: $neighbourMean = neighbourMean/(post - pre + 1)$
14: // check whether $similarity_i$ is larger than the mean value of k neighbors
15: **if** $similarity_i \geqslant neighbourMean$ **then**
16: divide $byte_i$ and $byte_{i+1}$ into a cluster
17: **end if**
18: **end for**
19: Return (clusters);

After that, $n-1$ similarities are obtained. According to these similarities, AutoBoundary divides message bytes into clusters. To evaluate whether a similarity is high enough, we need a reference value. Therefore, for each one out of $n-1$ similarities, AutoBoundary finds its k neighbors, and calculates the mean value. If a similarity is larger than its k-neighbor mean value, two message bytes related to the similarity are divided into a cluster. At last, every message byte is covered by one and only one cluster. While a cluster is identified as a protocol field, field boundaries is set between clusters.

4 Evaluation

We have implemented an AutoBoundary prototype in 20,500 lines of source code on Linux 3.16 (Debian 8.5.0). The execution monitor module extends the instrumentation tool Pin [12] (version 2.14-71313). However, we note that our design is not tightly coupled with Pin, and can be implemented using other instrumentation tools, e.g., Valgrind [15]. The MP tree generation module takes a memory propagation record file that is the outcome of execution monitor module as input, and outputs MP trees. Based on MP trees, the field boundary identification module infers message formats.

We will present two sets of experiments. The first set of experiments involves 20 kinds of prototype messages from 3 known ICN protocols, including *Modbus/TCP*, *IEC 60870-5-104* and *ISO 9506*. The second set of experiments involves 10 protocol messages in a closed ICN protocol used by Siemens PLCs, namely *S7*. These messages are either for conveying commands from the engineer station or for retrieving I/O data from the controller.

In the first set of experiments with known ICN protocols, we can quantitatively evaluate the effectiveness of AutoBoundary. We compare our results with the results from a popular network protocol analyzer – Wireshark. We present the set of message fields as F, and the number of F as $|F|$. We count $|F|$ in both Wireshark and AutoBoundary results. We also count the number of fields in protocol specifications, which will be taken as benchmarking. Because both Wireshark and AutoBoundary may consolidate multiple protocol fields as one coarse-grained field, we count the total number of coarse-gained fields as $|E_c|$. On the other hand, they may divide a protocol field into multiple overly-fine-grained fields. We count the number of overly-fine-grained fields as $|E_o|$. Table 1 reports the results. In the following, we describe our experiments in greater detail.

4.1 Modbus TCP Request

In this experiment, we monitor the execution of a Modbus/TCP server implemented by libmodbus v3.0.6, and trace 20 Modbus/TCP messages (8 types). The results in Table 1 show that AutoBoundary identifies all protocol fields, as there is one overly-fine-grained field discovered by Wireshark. For the error ratio, AutoBoundary performs better. As a detailed example, for the "Write Single Coil" sub-messages, Wireshark reports $|E_c| = 0$ and $|E_o| = 1$ while AutoBoundary

Table 1. Protocol field comparison between Wireshark and AutoBoundary

Protocol	Message type	Specification	Wireshark			AutoBoundary		
		$\|F\|$	$\|F\|$	$\|E_c\|$	$\|E_o\|$	$\|F\|$	$\|E_c\|$	$\|E_o\|$
Modbus/TCP	Write Single Coil	7	8	0	1	7	0	0
	Write Multiple Coils	9	9	0	0	9	0	0
	Write Single Register	7	7	0	0	7	0	0
	Write Multiple Registers	9	9	0	0	9	0	0
	Read Coils	7	7	0	0	7	0	0
	Read Discrete Inputs	7	7	0	0	7	0	0
	Read Holding Registers	7	7	0	0	7	0	0
	Read Input Registers	7	7	0	0	7	0	0
IEC 60870-5-104	U format STARTDT	5	4	1	0	4	1	0
	U format STOPDT	5	4	1	0	4	1	0
	I format Interrogation	16	15	1	0	11	5	0
	I format C_SC_NA_1	19	17	2	0	12	7	0
	I format C_DC_NA_1	18	17	1	0	11	7	0
	I format C_SE_NB_1	22	17	5	0	12	10	0
	I format C_RD_NA_1	15	14	1	0	10	5	0
	I format C_CS_NA_1	28	23	5	0	13	15	0
ISO 9506	getNameList	16	4	12	0	11	4	0
	Read	23	9	14	0	23	0	0
	Write	26	11	15	0	28	0	2
	defineNamedVariableList	29	12	17	0	31	0	2

shows $|E_c| = 0$ and $|E_o| = 0$, comparing with the Modbus/TCP specification. The reason for having an overly-fine-grained field in Wireshark result is that the low byte of "output value" field is erroneously identified as a padding of Ethernet frame. And if a Modbus/TCP server does not use the default port 502 (e.g., the example program of libmodbus uses 1502 as the default port), Wireshark cannot identify any Modbus/TCP fields. AutoBoundary works well no matter which port number is used by the Modbus/TCP server. Therefore we believe that AutoBoundary outperforms Wireshark, when confronting Modbus/TCP.

4.2 IEC 60870-5-104 Request

In this experiment, we monitor the execution of an IEC 60870-5-104 server implemented by lib60870 v0.9.4, and trace 20 messages (8 types) in control direction. Table 1 shows the existence of coarse-grained fields both in the Wireshark and AutoBoundary results. More specifically, Wireshark identifies 86.7% of protocol fields, while AutoBoundary only discovers 60.1% of them. To find out the root cause, we make in-deep analysis against "U format – STARTDT" sub-messages and "I format – Interrogation" sub-messages.

For the "U format – STARTDT" sub-messages, Wireshark reports $|E_c| = 1$ and $|E_o| = 0$ while AutoBoundary shows $|E_c| = 1$ and $|E_o| = 0$. According to IEC 60870-5-104 specification, a "STARTDT" message has 1 start field, 1 length field and 4 control fields. The first control field is divided into two parts: bit 1–2 are always 1 (format type), bit 3–8 represents one function (TESTFR, STOPDT or STARTDT). Therefore, the first control field should be treated as 2 different fields. Because the last 3 control fields are always 0, they are combined as 1 field. Wireshark does not identify the last field, i.e. 3-byte 0x00. AutoBoundary treats the first control field as 1 field. It does not discover the format type field (bit 1–2 of the first control field) and function field (bit 3–8 of the first control field).

For the "I format – Interrogation" sub-messages, Wireshark reports $|E_c| = 1$ and $|E_o| = 0$ while AutoBoundary shows $|E_c| = 5$ and $|E_o| = 0$. An "Interrogation" message has 16 protocol fields, including start field, length field, two format type fields (bit 1 of CFO[1] 1 and bit 1 of CFO 3), send sequence number field, receive sequence number field, type identification field, SQ field (bit 8 of VSQ[2]), number field, test flag field (bit 8 of COT[3]), P/N field (bit 7 of COT), cause field, originator address field, common address field, information object address field and qualifier of interrogation field. Wireshark only discovers one format type field, while AutoBoundary does not identify SQ, test flag, P/N and format type fields.

There is a main reason behind the coarse-grained field: for AutoBoundary, the granularity of dynamic taint analysis is byte but not bit. So it cannot identify field boundaries which are not byte-aligned, such as function field in the "STARTDT" message, SQ field in the "Interrogation" message and so on. It is easy to modify the granularity of dynamic taint analysis from byte to bit, but the resource consumption will be increasing exponentially. To balance out the costs and benefits, we keep the byte-granularity.

4.3 ISO 9506 Request

In this experiment, we monitor the execution of an ISO 9506 (MMS) server implemented by libiec61850 v1.0.1, and trace 12 messages (4 types) in control direction. Table 1 shows that, there are coarse-grained fields both in the Wireshark results and AutoBoundary results. Wireshark only identifies 39.3% of protocol fields. ISO 9506 (MMS) uses Abstract Syntax Notation One (ASN.1) to encode request PDUs. The Basic Encoding Rules (BER) of ASN.1 has three parts: identifier, length and content. Wireshark only identifies the content fields. This is the root cause for the poor result. On the other hand, AutoBoundary discovers 94.7% of protocol fields. As a detailed example, for "getNameList" sub-messages, AutoBoundary reports $|F| = 11$ and $|E_c| = 4$. Through static analysis against the source code, we find that the implementation of the protocol ignores specific messages fields. So these fields cannot be inferred by AutoBoundary.

[1] Control Field Octet.
[2] Variable Structure Qualifier.
[3] Cause Of Transmission.

For "write" sub-messages, AutoBoundary reports $|F| = 28$ and $|E_o| = 2$. The reason behind overly-fine-grained fields is that the "itemID" field consists of multiple parts – a logical node name "LLN0", a functional constraint "ST", and a data name "Health". The implementation of the protocol parses them respectively. Therefore, AutoBoundary regards one "itemID" field as three different fields. The same thing happens to "defineNamedVariableList" sub-messages.

Table 2. Protocol field comparison between S7 Wireshark dissector and AutoBoundary

Protocol	Message function code	S7 Wireshark dissector	AutoBoundary		
		$\|F\|$	$\|F\|$	$\|E_c\|$	$\|E_o\|$
Siemens S7 Protocol	Setup communication	11	11	0	0
	Upload	16	16	0	0
	PLC Stop	10	10	0	0
	Write Variable	20	20	0	0
	(Multiple) Read Variable	48	48	0	0

4.4 Siemens S7 Messages

We present our second set of experiments showing that AutoBoundary can uncover the field boundaries of a closed ICN protocol (S7) message used by Siemens PLCs. To verify the AutoBoundary results, we use a great project – S7 Wireshark dissector, and compare results between AutoBoundary and S7 Wireshark dissector.

We monitor the execution of an S7 server implemented by Snap7, and trace 10 messages (5 types) in control direction. As shown in Table 2, for each field boundary identified by S7 Wireshark dissector, there is an identical field boundary automatically discovered by AutoBoundary.

5 Limitations and Future Work

The first limitation of AutoBoundary is the granularity of dynamic taint analysis. To balance out the costs and benefits, we choose 1-byte as the minimum unit when tracing taint data. But some ICN protocol fields are not byte-aligned, such as IEC 60870-5-104 requests mentioned in Sect. 4.2. In other words, if a protocol field is not byte-aligned, AutoBoundary cannot infer the filed boundary accurately. Secondly, AutoBoundary is the dynamic trace dependency. If the implementation of an ICN protocol ignores some message fields, AutoBoundary cannot discover the boundaries of these fields, just like what happened for ISO 9506 requests mentioned in Sect. 4.3. Fixing the above problems is a part of our future work. And we plan to extend execution monitoring process to the protocol client (e.g., HMI application), which is easier to be obtained. In addition, identifying entire structure of a message is another part of our future work.

6 Related Works

The methods of automatic protocol reverse engineering can be classified into network trace analysis and dynamic analysis approaches.

The network trace analysis approaches for protocol reverse engineering take as input a network capture and use clustering techniques to determine protocol information. *Protocol Informatics* project [2] aims to employ Smith Waterman algorithm to infer protocol formats from a set of protocol network packets. *Discoverer* [7] leverages recursive clustering and type-based sequence alignment to infer message formats. *RolePalyer* [8] can mimic both the server side and the client side of the session for application protocols. *Biprominer* [19] and *ProDecoder* [20] are two automatic protocol reverse engineering tools, which use statistical methods to find keywords and probable keyword sequences. *AutoReEngine* [13] adopts a similar method but measuring keyword location from the beginning of a message as well as the end. *ReverX* [1] uses a speech recognition algorithm to identify delimiters, and then finds keywords within protocol messages by identifying the frequency of byte sequences.

The dynamic analysis approaches monitor the execution of a software binary that implements the communication protocol to identify the protocol message fields. *Polyglot* [6] depends on the existence of loops in which tainted data is iteratively compared to a constant value. *Dispatcher* [5] targets transmitted messages. To extract the message format of sent messages, it leverages the intuition that the structure of the output buffer represents the inverse of the structure of the sent message. *AutoFormat* [11] treats consecutive bytes that are run in the same execution context as message fields, and then exposes a tree of hierarchal fields. *Tupni* [9] can reverse engineer an input format with a rich set of information. Its key property is that it identifies arbitrary record sequences by analyzing loops in a program. *Wondracek* et al. [21] presented a approach to extract information about the fields of individual messages, and aggregate this information to determine a general specification of the message format.

7 Conclusion

We have proposed MP tree to analyze the movement trace of message bytes, and described the way how to build, compress, and compare MP trees. Based on MP tree, we have presented AutoBoundary, a system for automatic protocol field boundary identification. We have implemented a prototype of AutoBoundary and evaluated it with a variety of ICN protocol messages. Our experimental results show that AutoBoundary achieves high accuracy in ICN protocol filed boundary identification.

Acknowledgments. This work was supported by the National Key Research and Development Program of China (No. 2017YFB0801900). We would like to thank all anonymous reviewers for helping us make this paper better.

References

1. Antunes, J., Neves, N., Verissimo, P.: Reverse engineering of protocols from network traces. In: Working Conference on Reverse Engineering, WCRE 2011, pp. 169–178. IEEE, Limerick (2011)
2. Beddoe, M.A.: Network protocol analysis using bioinformatics algorithms (2012). http://www.4tphi.net/~awalters/PI/pi.pdf. Accessed 31 Aug 2016
3. Borisov, N., Brumley, D.J., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: A generic application-level protocol analyzer and its language. In: 14th Symposium on Network and Distributed System Security, NDSS 2014. The Internet Society, San Diego, February 2007
4. Brumley, D., Caballero, J., Liang, Z., Newsome, J., Song, D.: Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: 16th USENIX Security Symposium, pp. 213–228. USENIX Association, Boston, August 2007
5. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: ACM Conference on Computer and Communications Security, CCS 2009, pp. 621–634. ACM, Chicago (2009)
6. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: ACM Conference on Computer and Communications Security, CCS 2007, pp. 317–329. ACM, Alexandria (2007)
7. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol reverse engineering from network traces. In: 16th USENIX Security Symposium, pp. 199–212. USENIX Association, Berkeley, August 2007
8. Cui, W., Paxson, V., Weaver, N.C., Katz, R.H.: Protocol-independent adaptive replay of application dialog. In: Network and Distributed System Security Symposium, NDSS 2006, pp. 487–490. The Internet Society, San Diego (2006)
9. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: automatic reverse engineering of input formats. In: ACM Conference on Computer and Communications Security, CCS 2008, pp. 391–402. ACM, Alexandria (2008)
10. Egele, M., Kruegel, C., Kirda, E., Yin, H., Song, D.: Dynamic spyware analysis. In: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference, ATC 2007, pp. 18:1–18:14. USENIX Association, Santa Clara (2007). http://dl.acm.org/citation.cfm?id=1364385.1364403
11. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: Network and Distributed System Security Symposium, NDSS 2008. The Internet Society, San Diego, February 2008
12. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200. ACM, Chicago, June 2005
13. Luo, J.Z., Yu, S.Z.: Position-based automatic reverse engineering of network protocols. J. Netw. Comput. Appl. **36**(3), 1070–1077 (2013)
14. Needleman, S.B., Wunsch, C.D.: A general method applicable to search for similarities in amino acid sequence of 2 proteins. J. Mol. Biol. **48**(3), 443–453 (1970)
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Not. **42**(6), 89–100 (2007)

16. Oehlert, P.: Violating assumptions with fuzzing. IEEE Secur. Priv. **3**(2), 58–62 (2005)
17. Pang, R., Paxson, V., Sommer, R., Peterson, L.: Binpac: a yacc for writing application protocol parsers. In: ACM SIGCOMM Conference on Internet Measurement, pp. 289–300. ACM, Rio De Janeriro, October 2006
18. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Krgel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: Network and Distributed System Security Symposium, NDSS 2007. The Internet Society, San Diego, February–March 2007
19. Wang, Y., Li, X., Meng, J., Zhao, Y., Zhang, Z., Guo, L.: Biprominer: automatic mining of binary protocol features. In: 12th International Conference on Parallel and Distributed Computing. Applications and Technologies, PDCAT 2011, pp. 179–184. IEEE, Gwangju (2011)
20. Wang, Y., et al.: A semantics aware approach to automated reverse engineering unknown protocols. In: 20th IEEE International Conference on Network Protocols, ICNP 2012, pp. 1–10. IEEE, Austin (2012)
21. Wondracek, G., Comparetti, P.M., Krgel, C., Kirda, E.: Automatic network protocol analysis. In: Network and Distributed System Security Symposium, NDSS 2008. The Internet Society, San Diego, February 2008