Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution *

Zhiqiang Lin†, Xuxian Jiang‡, Dongyan Xu†, Xiangyu Zhang†

†Department of Computer Science
Purdue University
{zlin, dxu, xyzhang}@cs.purdue.edu

‡Department of Information and Software Engineering
George Mason University
xjiang@ise.gmu.edu

Abstract

Protocol reverse engineering has often been a manual process that is considered time-consuming, tedious and error-prone. To address this limitation, a number of solutions have recently been proposed to allow for automatic protocol reverse engineering. Unfortunately, they are either limited in extracting protocol fields due to lack of program semantics in network traces or primitive in only revealing the flat structure of protocol format. In this paper, we present a system called AutoFormat that aims at not only extracting protocol fields with high accuracy, but also revealing the inherently "non-flat", hierarchical structures of protocol messages. AutoFormat is based on the key insight that different protocol fields in the same message are typically handled in different execution contexts (e.g., the runtime call stack). As such, by monitoring the program execution, we can collect the execution context information for every message byte (annotated with its offset in the entire message) and cluster them to derive the protocol format. We have evaluated our system with more than 30 protocol messages from seven protocols, including two text-based protocols (HTTP and SIP), three binary-based protocols (DHCP, RIP, and OSPF), one hybrid protocol (CIFS/SMB), as well as one unknown protocol used by a real-world malware. Our results show that AutoFormat can not only identify individual message fields automatically and with high accuracy (an average 93.4% match ratio compared with Wireshark), but also unveil the structure of the protocol format by revealing possible relations (e.g., sequential, parallel, and hierarchical) among the message fields.

*Part of this research has been supported by the National Science Foundation under grants CNS-0716376 and CNS-0716444. The bulk of this work was performed when the first author was visiting George Mason University in Summer 2007.

1 Introduction

The knowledge about network protocol specification is valuable to many security applications: Network-based intrusion detection systems (e.g., Snort [4] and Bro [24]) or vulnerability-specific filters (e.g., Shield [28]) require knowledge about protocols to perform deep packet inspection; Network management software depends on such knowledge to correctly recognize and classify monitored network traffic; Fuzz testing (e.g., Packet Vaccine [29] and ShieldGen [14]) can take advantage of such knowledge to improve the fuzzing process by generating "malicious" inputs more efficiently.

In practice, however, it is mostly a manual and error-prone process to derive protocol specifications. For an open protocol (e.g., HTTP), the specification can be acquired from public documentation such as RFCs. For a closed protocol (e.g., SMB or Skype), the specification has to be reverse engineered manually and complexities arise: (1) A single protocol message usually contains a large number of fields (e.g., the Samba NTLMSSP_AUTH message contains about 50 fields); (2) An individual field may not be static and may have varying size; and (3) More importantly, there may exist complex relationships (e.g., sequential, parallel, and hierarchical) or dependencies among the fields. As such, protocol reverse engineering is widely known as a challenging task and existing manual approaches tend to be tedious, time-consuming, and error-prone. As an example, after numerous trials and errors, it took 12 years for the open-source Samba project to reverse engineer the Microsoft SMB protocol [1].

To address this challenge, new solutions, including Protocol Informatics (PI) [3], Discoverer [12], and Polyglot [9], have recently been proposed to automate the process of reverse engineering network protocols. The PI project adopts sequence alignment – a technique widely used in bioinformatics to find certain patterns in large sequences of strings

– to infer protocol format from network traces. Discoverer takes a step further by applying recursive clustering techniques to group messages with similar formats such that, with the help of a type-based sequence alignment algorithm, it can produce more concise results in the revealed protocol format. However, as pointed out in [9], the lack of protocol semantics in network traces fundamentally limits the accuracy of the extracted protocol format. Moreover, any network trace-based approach becomes ineffective when network traffic is encrypted.

From another perspective, Polyglot recognizes the fact that the way a protocol is implemented for handling incoming protocol messages reveals a wealth of information about protocol format. Therefore, the protocol implementation can be naturally analyzed to uncover protocol format. Specifically, Polyglot proposes a dynamic binary analysis approach that exploits the semantics of payload-processing instructions to identify detailed message fields. Although instruction-level semantic information is indeed useful in extracting message fields, it is still limited in only revealing the "flat" structure of protocol format. To reverse engineer network protocols more accurately and thoroughly, in addition to the extraction of detailed protocol fields, it is equally important to expose inherently hierarchical structures of "non-flat" protocol messages and reveal cross-field relations.

We note that the above protocol information is naturally specified in protocol specifications. For example, the *Backus - Naur Form* (BNF), which has been widely used to express network protocol syntax, is designed to be expressive enough in describing the hierarchical structure of a protocol message and cross-field relations within the message. Meanwhile, a number of existing techniques can benefit from the richer knowledge about protocol format. For example, fuzz testing can greatly reduce the fuzzing space with the knowledge of possible cross-field relations; Discoverer and PI can leverage the knowledge to achieve better alignment among collected traces.

In this paper, we present AutoFormat, a new host-based approach that aims at uncovering not only detailed protocol fields in a protocol message, but also the inherent hierarchical structure as well as cross-field relations. AutoFormat is based on the key observation that *different protocol fields in the same message are typically handled in different execution contexts* such as the run-time call stack and location of the instruction being executed. In other words, adjacent message bytes belonging to the same protocol field are usually handled in the same execution context. Therefore, by monitoring program execution, we can collect execution context information for every message byte annotated with its offset in the entire message, and then cluster them to discover protocol fields. Further, based on the same context information, we can uncover the structural hierarchy of the message format as well as possible cross-field relations in the message.

We have implemented a proof-of-concept prototype and evaluated it with more than 30 protocol messages from seven protocols, including two text-based protocols (HTTP and SIP), three binary-based protocols (DHCP, RIP, and OSPF), one hybrid protocol (CIFS/SMB), and one unknown protocol used by a real-world malware. The experimental results are encouraging: For the six known protocols, AutoFormat is able to identify protocol fields automatically with high accuracy (an average 93.4% match ratio compared with the message fields derived by Wireshark [5]). Furthermore, it unveils the hierarchical structure of the entire message as well as cross-field relations. For the unknown malware protocol, AutoFormat results match well with our manual static analysis results. AutoFormat does not require accessing protocol source code and is therefore applicable to analyzing proprietary or unknown protocols.

The rest of the paper is organized as follows. Section 2 describes the problem scope and defines the terminologies used in the paper. The system design and key techniques for the extraction of protocol format will be presented in Section 3. In Section 4, we show the evaluation results. The related work will be discussed in Section 5. We examine limitations of AutoFormat and suggest possible improvement in Section 6. Section 7 concludes this paper.

2 Problem Scope and Terminologies

In this section, we first discuss the general goals of network protocol reverse engineering and outline our specific problem scope. We then define the terminologies that will be used throughout the paper.

2.1 Problem Scope

In network protocol reverse engineering, there exist two main challenging tasks: (1) The first task focuses on each individual protocol message and aims at identifying the boundary (or size) of every single protocol field as well as the entire structure built on the fields; (2) The second task involves multiple protocol messages and the goal is to build the entire protocol state machine, which includes various protocol-specific states and their transitions. Since the first task lays the foundation for protocol reverse engineering and its accuracy and completeness affects the second task, we in this paper focus on the first task and leave the second one as future work.

To articulate the challenges that arise from the first task, we use a real-world example. Figure 1 shows the standard BNF structure of the HTTP Request message documented in RFC2616 ("Hypertext Transfer Protocol – HTTP/1.1"). Particularly, an HTTP Request message contains multiple

```
Request = Request-Line
        *(( general-header
        | request-header
        | entity-header ) CRLF) CRLF
        [ message-body ]
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Figure 1. The BNF structure of the HTTP Request message documented in RFC2616.

high-level fields: `Request-Line`, `general-header`, `request-header`, `entity-header`, `CRLF`, and optionally `message-body`. Notice that (1) A field can contain multiple sub-fields. For example, the `Request-Line` field contains the method (`Method`) to be applied to the requested resource, the identifier (`Request-URI`) of the requested resource, and the protocol version (`HTTP-Version`) in use, as well as several separators such as `SP` and `CRLF`. (2) The "*" (iterative) and "|" (alternative) symbols in Figure 1 reflect the intrinsic *parallel* relationship among certain high-level fields, e.g., `general-header`, `request-header`, and `entity-header`. As such, a solution to protocol reverse engineering should not only identify the boundary of each field in the protocol message, but also structure the identified fields so that the overall message skeleton and the relations among message fields can be uncovered.

2.2 Terminologies

Considering the recursive nature of defining a protocol field, this paper uses the term *finest-grained field* to represent the smallest subsequence that cannot be further divided into smaller sub-fields. For ease of representation, we use $\Phi(x)$ to represent a field x in a protocol format ($\Phi(x)$ can be thought of as the name of field x), and its value is a set which contains all the offsets in the protocol message belonging to $\Phi(x)$ (hence, the size of field x can simply be denoted as $|\Phi(x)|$). Using the same example, we use $\Phi($`Request-Line`$)$ and $\Phi($`Method`$)$ to represent two different fields even though $\Phi($`Method`$)$ is a sub-field of $\Phi($`Request-Line`$)$. In the following, we define three important types of relation between protocol fields and the determination of these relations is a main focus of this paper.

Hierarchical: The hierarchical relation reflects the fact that a field can be further divided into multiple sub-fields. As mentioned earlier, $\Phi($`Request-Line`$)$ in Figure 1 contains a number of sub-fields including $\Phi($`Method`$)$, $\Phi($`SP`$)$, $\Phi($`Request-URI`$)$, $\Phi($`SP`$)$, $\Phi($`HTTP-Version`$)$, and $\Phi($`CRLF`$)$. For simplicity, we call the field with a number of sub-fields a hierarchical field. $\Phi($`Request-Line`$)$ is a hierarchical field.

Sequential: The sequential relation captures the ordering between two adjacent fields in a protocol message. For example, in the hierarchical field $\Phi($`Request-Line`$)$, $\Phi($`Method`$)$ is always followed by $\Phi($`SP`$)$ which is in turn followed by $\Phi($`Request-URI`$)$. We call such adjacent fields sequential fields.

Parallel: The parallel relation reflects the fact that the positions of two or more fields are exchangeable in the protocol specification. As an example, Figure 1 shows that in an HTTP Request message, the positions of the following three fields, $\Phi($`general-header`$)$, $\Phi($`request-header`$)$ and $\Phi($`entity-header`$)$ can be exchanged without affecting the message's semantics. In this paper, we call such position-exchangeable fields parallel fields.

3 System Design

The intuition behind our approach is simple but effective: A binary implementing the protocol is programmed to recognize the protocol format of received messages. As such, the specific way of handling an incoming message can be examined to uncover its format. As an example, Figure 2 shows the code snippet from a real-world web server (i.e. the NullHTTPd server) that processes the header fields of an HTTP Request message. Based on the BNF format shown in Figure 1, the first line of the request payload (received in the context of the $sgets()$ function – line 129) contains the $\Phi($`Request-Line`$)$ field, which is divided into multiple sub-fields $\Phi($`Method`$)$, $\Phi($`Request-URI`$)$, and $\Phi($`HTTP-Version`$)$ (line 137). In other words, the fields $\Phi($`Method`$)$, $\Phi($`Request-URI`$)$, and $\Phi($`HTTP-Version`$)$ are sequential fields and their combination forms a hierarchical field. Similarly, the next few lines (lines 147-162) handle various other header fields and the way of handling them leads to the exposure of several parallel fields – $\Phi($`Cookie`$)$ (lines 154-155), $\Phi($`Host`$)$ (lines 156-157), $\Phi($`If-Modified-Since`$)$ (lines 158-159), and $\Phi($`User-Agent`$)$ (lines 160-161).

AutoFormat is interested in how field-specific execution context information can be collected and analyzed to extract protocol format. Figure 3 shows an architectural overview of AutoFormat, which has two main components: a *context-aware execution monitor* and a *protocol field identifier*. Given a binary that implements the protocol to be analyzed, AutoFormat works as follows: (1) On receiving an incoming protocol message, it first marks the received data and keeps track of their propagation at the byte granularity; (2) Once a message byte is read, the execution monitor logs that particular byte, its offset in the entire message, and the run-time execution context at that moment, which includes the call stack and the location of the instruction being ex-

```
119 int read_header(int sid) {
...
     /* read a line with no more than size(line)-1 bytes; the return character '\n' signals the end of the line. */
129  sgets(line, sizeof(line)-1, conn[sid].socket);
...
     /* break down the first line of the HTTP Request message, i.e., the f(Request-Line) field, into different message fields. */
137  if (sscanf(line, "%[^ ] %[^ ] %[^ ]", conn[sid].dat->in_RequestMethod, conn[sid].dat->in_RequestURI, conn[sid].dat->in_Protocol)!=3)
138          printerror(sid, 400, "Bad Request", "Can't Parse Request.");
...
147  while (strlen(line)>0) {

         /* read the next line, which is either a general-header, request-header, or entity-header. */
148      sgets(line, sizeof(line)-1, conn[sid].socket);
...
         /* break down the line into more specific subfields, e.g., f(Cookie), f(Host), f(If-Modified-Since), and f(User-Agent). */
154      if (strncasecmp(line, "Cookie: ", 8)==0)
155         strncpy(conn[sid].dat->in_Cookie, (char *)&line+8, sizeof(conn[sid].dat->in_Cookie)-1);
156      if (strncasecmp(line, "Host: ", 6)==0)
157         strncpy(conn[sid].dat->in_Host, (char *)&line+6, sizeof(conn[sid].dat->in_Host)-1);
158      if (strncasecmp(line, "If-Modified-Since: ", 19)==0)
159         strncpy(conn[sid].dat->in_IfModifiedSince, (char *)&line+19, sizeof(conn[sid].dat->in_IfModifiedSince)-1);
160      if (strncasecmp(line, "User-Agent: ", 12)==0)
161         strncpy(conn[sid].dat->in_UserAgent, (char *)&line+12, sizeof(conn[sid].dat->in_UserAgent)-1);
162  }
...
187 }
```

Figure 2. Code snippet in a real-world web server parsing the incoming HTTP Request message

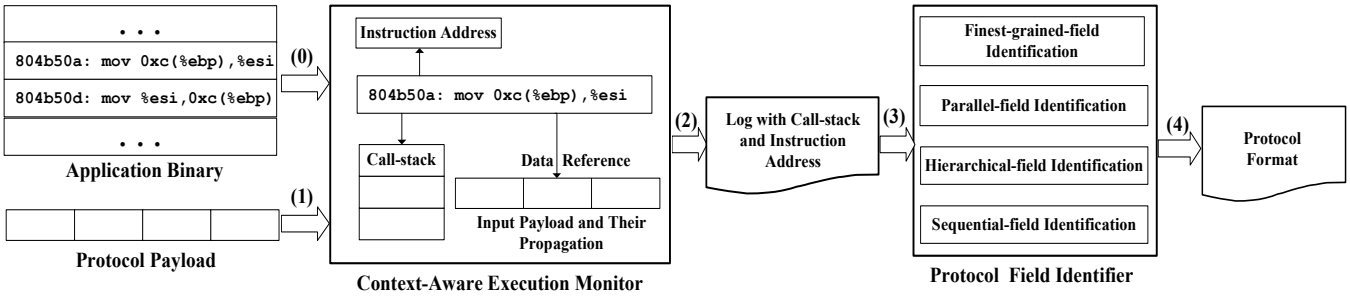


Figure 3. AutoFormat: An architectural overview

ecuted; (3) With the collected context information, the offline *protocol field identifier* is invoked to identify protocol fields and extract the structural layout of the message.

3.1 Context-Aware Execution Monitor

By monitoring program execution, we can intercept the network-related system calls (e.g., *sys_socket*), mark the messages received, and annotate every message byte with its offset in the entire message. Moreover, throughout the message processing life-time, we instrument the data movement instructions (e.g., `mov`) as well as arithmetic/logic instructions (e.g., `add`, `mul`, `and`) to propagate the annotation. More specifically, for a data movement instruction, we check whether the source operand is marked. If yes, we will annotate the destination operand, which can be a register or a memory location, with the source operand's annotation, i.e. its offset in the original message. If the source operand is not marked, we will simply unmark the destination operand. If two marked operands appear in the same instruction, we will union their annotations (e.g., for the `add` operation, the result is the union of the operands if they are both marked). Note that the marking and propagation operations are based on the taint analysis technique, which has been widely adopted. We refer interested readers to related literature such as [8, 10, 11, 15, 23, 26, 27, 31].

AutoFormat is interested in two types of execution context information: the run-time call stack and the address of the instruction that accesses a marked memory location. By monitoring program execution, we can easily record the instruction location when it is referencing a marked memory location. However, to acquire the run-time call stack information, we need to traverse the stack frames: For each function stack frame, we can obtain the return address inside the frame and, if the symbol information is available, we can further derive the function name from the return address. Note that such technique works except when the program is compiled without the stack frame pointer support, which prevents us from traversing the stack. However, we can overcome this problem by instrumenting the function call and return instructions and maintaining a shadow stack frame inside the execution monitor. The shadow stack frame contains the return addresses for all the functions called so far. From this shadow stack frame, we are able to derive the run-time call stack.

(a) A raw HTTP Request message captured by TCPDUMP

(b) The protocol format identified by Wireshark

Figure 4. A raw HTTP Request message and the protocol format identified by Wireshark

Algorithm 1 *Protocol Field Tree Generation*

1: **Input**: the log array log (with total N records). For the i^{th} record $log[i]$, it has members: (1) $log[i].o$ – the byte offset, (2) $log[i].s$ – the call-stack, and (3) $log[i].l$ – the instruction location;
2: **Output**: $ftree$ – the protocol field tree.
3: **Field_Tree_Creation** (log) {
4: $ftree \leftarrow ROOT$; /* Create the ROOT node, which contains all the offsets of input data */
5: $ROOT \leftarrow \{0, 1, 2, ...m-1\}$;
6: **Get** $log[0].o, log[0].s$; /* Process the 1^{st} record */
7: $p \leftarrow \{log[0].o\}$;
8: **for** (i$\leftarrow$1; $i < N$; i++){ /* Process the i^{th} record */
9: **Get** $log[i].o, log[i].s$;
10: $q \leftarrow \{log[i].o\}$;
11: **if** (($log[i].o$ == ($log[i-1].o$)+1) && ($log[i].s == log[i-1].s$))
12: $p \leftarrow$**UNION**(p, q);
13: **else**{
14: **Create** a new node v with offset interval p;
15: **Find** a node u in $ftree$, such that u, but not its children (if any), subsumes (the offset interval of) v;
16: **if**(u has children) **move** those children whose offset intervals are each a subset of v as children of v;
17: **Insert** v as the child of u;
18: $p \leftarrow q$;
19: }
20: }
21: **Return** $ftree$;
22: }

3.2 Protocol Field Identifier

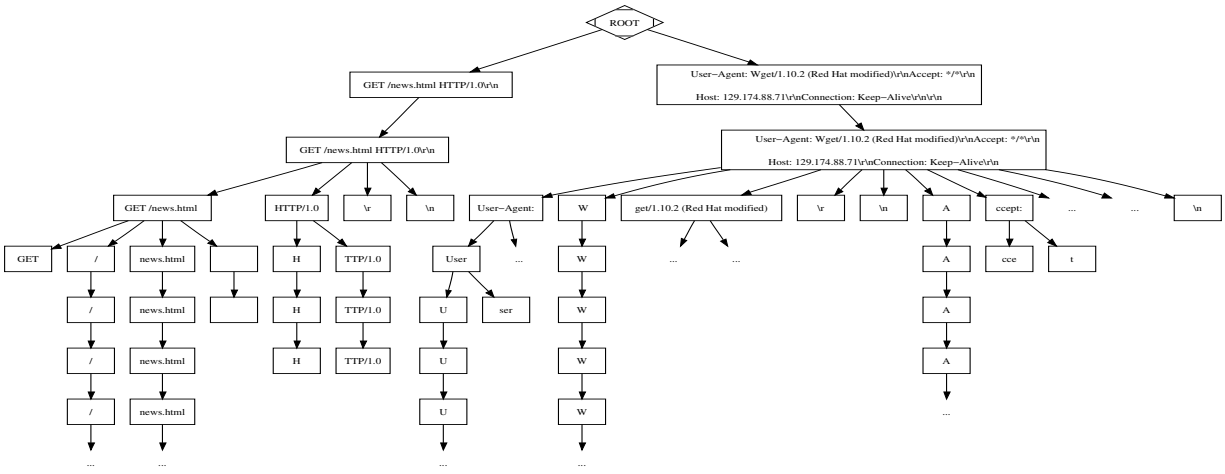
In this subsection, we walk through an example to demonstrate how AutoFormat works. This example is related to an HTTP Request message that is sent by the Linux *wget* command to ask for an HTML file named *news.html* from an Apache-based (version 2.0.59) web server. Figure 4(a) shows the raw request payload in the TCPDUMP form and Figure 4(b) shows the message format identified by Wireshark [5]. Note that the execution of the web server is monitored by AutoFormat.

3.2.1 Identifying Finest-Grained Fields and Hierarchical Fields

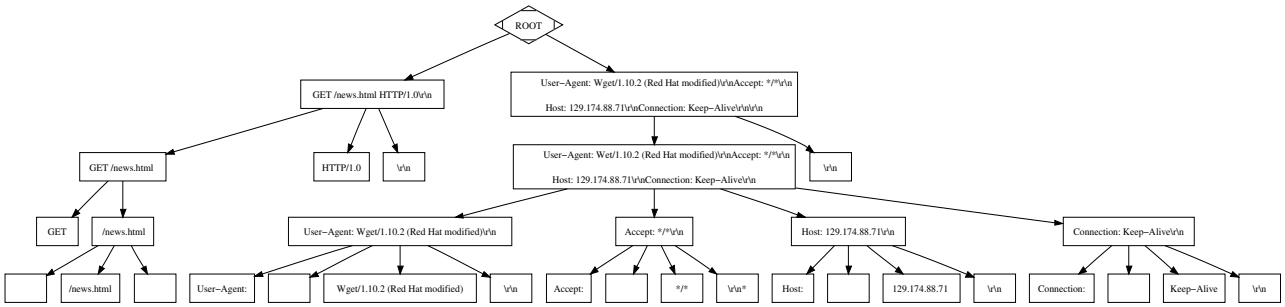
When a marked memory location is being read, AutoFormat will log the execution context and save it as a record in the form of $< o, c, s, l >$, where o is the offset of the referenced memory in the entire message, c is its content, s is the run-time call stack when the memory reference occurs, and l is the location of the memory reference instruction. When processing the log file, we can simply consider the log file as an array, log, with N elements and each element, say $log[i]$, has four members: $log[i].o$, $log[i].c$, $log[i].s$, and $log[i].l$. Because of the locality property of program execution, certain offset may be intensively referenced and there may exist several continuous log records that are identical. In that case, we will first pre-process the log file to discard all but one of the successive identical log records. Some pre-processed log records are shown in Appendix I.

Our next step is to build a protocol field tree $ftree$ and use the protocol field tree to store the identified fields and express possible hierarchical relations among them. More specifically, each node of $ftree$ represents either a finest-grained field or a hierarchical field. Each field is associated with an *offset interval* denoted by the starting position and the size of the field. A node is a child to a parent if and only if the former's offset interval is a subset of the latter's offset interval. Our protocol field tree generation algorithm is shown in **Algorithm 1**.

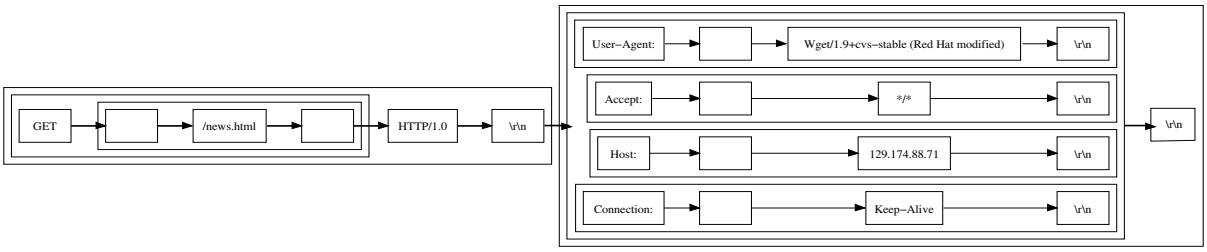
Essentially, Algorithm 1 scans the entire log file and checks whether two successive records ($log[i]$ and $log[i-1]$) are related to two consecutive offsets (line 12: $log[i].o == (log[i-1].o) + 1$) of the input data and have the same ex-



(a) Step 1: Building a protocol field tree from the raw execution traces



(b) Step 2: Refining the protocol field tree with the identified *finest-grained* fields and *hierarchical* fields



(c) Step 3: Presenting the protocol format with the discovered *parallel* and *sequential* fields

Figure 5. Steps of AutoFormat when reverse engineering the HTTP Request message format

ecution context (line 12: $log[i].s == log[i-1].s$)[1]. If so, we merge the corresponding offset intervals into one (line 13: $p \leftarrow$ **UNION**(p, q)). If not, a new protocol field node will be created (line 14). To link the new node to the tree, an existing node will be chosen as the new node's parent node (lines 15-18). This chosen node – but not its child (if any) – should contain the new node's offset(s). If we cannot find such a node, we will insert the new node as a child of

[1]We note that, when analyzing text protocols, we use $log[i].s$ as execution context. However, when analyzing binary protocols, we replace $log[i].s$ with $log[i].l$ – the current instruction location – as execution context.

the $ROOT$ node which contains all offsets (line 5). For the parent of the new node, some of its children may be moved down to become the new node's children (lines 16-17). The reason for the move is to maintain the hierarchical property of the protocol field tree. The result of running Algorithm 1 on the collected log for the `wget` request message is shown in Figure 5(a).

Ideally, we would like to have each leaf node as a finest-grained field. However, Figure 5(a) shows that not all the leaf nodes are finest-grained fields. Furthermore, the protocol field tree built by Algorithm 1 raises the following three issues: (1) Some leaf nodes might be of overly fine granularity. The reason is that the implementation code typically contains functions that do not always reference all the field offsets – examples are `strcmp`, `strcasecmp`, and `strlen`. In the HTTP Request example, the "culprit" function is `ap_rgetline_core`, which, in the Apache-2.0.59 implementation, reversely reads an input line and naturally introduces a few *overly-fine-grained fields*. These factors thus lead to some strange nodes in the tree such as `H`, `U`, `W`, `A`, '`\r`' and '`\n`'; (2) Certain fields may be referenced multiple times at different time instances (e.g.,'`\`', `news.html`, `H`, `U`) and thus they are redundant; (3) There exist some fields that may not be referenced at all. One example is the `space` in the " \" field. For each of the above issues, we propose a corresponding heuristic to refine the protocol field tree.

- **Tokenization**: Text-based protocols usually have delimiters to separate protocol fields and the characters in each field can usually form a token. As such, if the content in two neighboring child nodes can form one token, we will merge these two child nodes into one. However, this heuristic may not be applicable to binary protocols.
- **Redundant node deletion**: An internal node in the field tree is redundant if and only if it has only one child (e.g., the left child of `ROOT` and the parent nodes with only one leaf node child in Figure 5(a)). Such redundancy can be removed by merging the internal node with its child. This refinement continues until no further elimination can be conducted.
- **New node insertion**: If the offsets of all children do not exactly match the offsets of their parent, we will insert a new child node with the missing offsets (the tokenization heuristic may apply). There exists another possibility of node insertion, which occurs when identifying parallel fields. We defer its description to Section 3.2.2.

Based on the above three heuristics, we can refine the protocol field tree and the result is shown in Figure 5(b). It is encouraging to notice that all the identified finest-grained fields are now leaf nodes of the tree. Thus, a simple tree traversal algorithm on the leaf nodes can immediately reveal the flat structure of the protocol message. To unveil the "non-flat" nature of the same protocol message, we need to identify the hierarchical fields. In fact, many existing protocol analyzers including Wireshark have provided this feature to facilitate the understanding of the protocol. Conveniently, the way we build the protocol field tree readily provides such information. We can simply perform a breadth-first traversal on the protocol field tree: Any non-leaf node represents a hierarchical field.

3.2.2 Identifying Parallel and Sequential Fields

Parallel fields are those whose positions are exchangeable in the message structure. The identification of parallel fields is particularly useful for protocol fuzz testing and for field alignment in PI [3]. Parallel fields are typically processed in a loop (an example is shown in Figure 2), thus these fields share certain execution history. To discover the parallel relations among the identified fields, AutoFormat utilizes the execution context history, defined as a sequence of execution contexts for a particular offset. Details are described in **Algorithm 2**.

Algorithm 2 *Parallel Field Identification*

```
1: /* log: the log file with N records; ftree: the protocol field tree*/
2: Parallel_Field_Identification (ftree, log){
3:     Breadth-first traverse ftree{
4:       for each node v traversed {
5:         for each of v's child child[i] {
               /* accumulate the history for the lowest offset of child[i]*/;
6:           Let lo be the lowest offset of child[i];
7:           for (k←0; k < N; k++){
8:             Get log[k].o, log[k].s;
9:             if (log[k].o == lo) child[i].history += log[k].s;
10:          }
11:        }
12:        Identify those children of v with similar execution history and mark
           each of them as a parallel field (or the start of a new parallel field);
13:      }
14:    }
15: }
```

Specifically, we first collect the execution history seen by the lowest offset (i.e. the first byte) of each node in the refined protocol field tree[2] (lines 5-11). For a parent node, if some of its child nodes share similar execution history[3], we will mark each of them as a parallel field (line 12). If there exist non-marked child node(s) between two marked ones, the marked one on the left (with a smaller offset) will join the non-marked child node(s) to form a new parallel field

[2]The reason for choosing the lowest offset is that not all offsets in a node are processed in the same capacity and the first byte of a field tends to be the most processed among all.

[3]We consider two execution histories as "similar" if they share common history prefix (as sequence of stack frames). The shared prefix is at least $h\%$ of the entire history and h is a tunable parameter. In our experiments, we set $h = 80$.

Protocol	Program	Binary Size	Protocol	Program	Binary Size
HTTP	Apache-2.0.59	253K	DHCP	Dhcp-3.0.5	1.86M
SIP	Asterisk-1.4.4	8.93M	RIP	Ripd (Zebra-0.95a)	234K
CIFS/SMB	Samba-3.0.8	2.77M	OSPF	Ospfd (Zebra-0.95a)	1.10M

Table 1. Six known protocols for the evaluation of AutoFormat

(recall the *new node insertion* heuristic in Section 3.2.1). In this case, a hierarchical node will be inserted into the tree to represent the new parallel field. Note that the size of the parallel fields may vary and they may have different number of sub-fields.

In the HTTP Request example, Algorithm 2 will identify the following four parallel fields: "`User-Agent: Wget...\r\n`", "`Accept: */*\r\n`", "`Host: 129.174.88.71\r\n`", and "`Connection: ...\r\n`". However, no new hierarchical nodes will be inserted into the protocol field tree as these fields have been represented by existing nodes in the tree.

Based on the protocol field tree and the parallel fields identified, AutoFormat further identifies the sequential fields: It first performs a pre-order traversal of the tree but only lists the leaf nodes and those internal nodes that each represents a *parent* of multiple parallel fields. The result of this traversal is a list of sequential fields. Recursively, the same traversal is performed on the sub-trees each rooted at a hierarchical node that represents a parallel field. This way we are able to identify all lists of sequential fields in the protocol field tree. We point out that the identification of sequential fields is also useful to protocol fuzz testing as the sequential fields should be treated atomically for each fuzz test.

Finally, after identifying both the parallel fields and the sequential fields, we can conveniently derive the BNF specification of the protocol message, guided by the protocol field tree. As an example, Figure 5(c) shows the BNF definition of the HTTP Request message: The leaf nodes are mapped to the smallest boxes while the internal nodes are mapped to the larger boxes. The topology of the tree naturally determines the *nesting* of the boxes. The sequential fields are connected by solid arrows, while the parallel fields are laid out in parallel. A comparison between AutoFormat's result in Figure 5(c) and the actual BNF specification confirms the correctness of the former. More evaluation results using real-world protocols will be presented in the next section.

4 Evaluation

We have implemented an AutoFormat prototype that extends the latest release of Valgrind [21] (version 3.2.3). This version provides memory marking and propagation capabilities necessary to enable context-aware execution monitoring. However, we note that our design is not tightly coupled with Valgrind and can be implemented using other binary instrumentation tools such as Pin [19] and QEMU [2].

We will present two sets of experiments. The first set of experiments involve 21 protocol messages from six *known* network protocols. Table 1 shows the list of protocols. Specifically, we choose two text-based protocols (HTTP and SIP), three binary-based protocols (DHCP, RIP, OSPF), and one hybrid protocol (CIFS/SMB). The program binaries are obtained either directly from the standard OS distribution or by compiling the source code with the default configuration. The second set of experiments involve 11 protocol messages in an *unknown* protocol used by the Slapper worm [25]. These messages are either for synchronization among infected hosts to form a P2P attack network or for conveying commands from the attacker (e.g., a bot master).

In the first set of experiments with known protocols, we are able to quantitatively evaluate the effectiveness of AutoFormat. More specifically, we compare our results with the results from the latest version (0.99.6) of a popular network protocol analyzer – Wireshark [5]. We represent the sets of finest-grained fields, hierarchical fields, and parallel fields as F, H, and P, respectively and we count $|F|$, $|H|$, and $|P|$ in both Wireshark and AutoFormat results. For each set, we also count how many Wireshark-identified fields are automatically discovered by AutoFormat and calculate the corresponding exact match ratio R_e. For the three sets, the ratios are denoted as $R_e(F)$, $R_e(H)$, and $R_e(P)$, respectively. In addition, since AutoFormat may divide a Wireshark-identified field into multiple fields, each of which is counted as an overly-fine-grained field, we count the total number of overly-fine-grained fields as $|F_o|$. Similarly, AutoFormat may consolidate multiple Wireshark-identified fields as one finest-grained field, which we call a coarse-grained field. We also count the total number of coarse-grained fields namely $|F_c|$.

Table 2 reports our results. We take the averages of $R_e(F)$, $R_e(H)$, and $R_e(P)$ and obtain the following: $\overline{R_e(F)} = 88.5\%$, $\overline{R_e(H)} = 98.0\%$, $\overline{R_e(P)} = 100.0\%$. If we do not differentiate the field types, the total average match rate in our experiments would be $\overline{R_e} = (\overline{R_e(F)}*21+\overline{R_e(H)}*16+\overline{R_e(P)}*7)/(21+16+7) = 93.4\%$, where 21, 16, 7 are the numbers of valid messages (excluding the "-" items in Table 2) used in our calculation of $\overline{R_e(F)}$, $\overline{R_e(H)}$, and $\overline{R_e(P)}$, respectively. In the following, we describe our experiments in greater detail.

Protocol	Request Msg Type	Wireshark			AutoFormat						Analysis of F	
		$\|F\|$	$\|H\|$	$\|P\|$	$\|F\|$	$R_e(F)$	$\|H\|$	$R_e(H)$	$\|P\|$	$R_e(P)$	$\|F_o\|$	$\|F_c\|$
HTTP	Linux Wget	8	1	0	23	**8/8**	9	**1/1**	4	-	15	0
	Linux Firefox	15	1	0	51	**15/15**	16	**1/1**	11	-	36	0
	Windows Firefox	12	1	0	39	**12/12**	13	**1/1**	8	-	27	0
	Windows IE	11	1	0	35	**11/11**	12	**1/1**	7	-	24	0
SIP	SIP REGISTER	20	9	11	116	**20/20**	37	**9/9**	12	**11/11**	96	0
	STATUS 200 OK	20	9	11	110	**20/20**	39	**9/9**	12	**11/11**	90	0
	SIP SUBSCRIBE	21	9	12	118	**21/21**	41	**9/9**	13	**12/12**	97	0
DHCP	DHCP BOOTP Request	39	8	7	34	**28/39**	9	**7/8**	7	**7/7**	1	5
RIP	RIPv2 Request	8	0	0	6	**5/8**	0	-	0	-	0	1
	RIPv2 Response	9	0	0	9	**9/9**	0	-	0	-	0	0
OSPF	OSPF Hello Packet	15	0	0	15	**15/15**	0	-	0	-	0	0
	OSPF DB Descr.	20	1	0	17	**15/20**	1	**1/1**	0	-	0	2
	OSPF LS Request	11	0	0	10	**9/11**	0	-	0	-	0	1
	OSPF LS ACK	16	0	0	13	**11/16**	0	-	0	-	0	2
CIFS/SMB	Negotiate Request	26	9	6	24	**23/26**	9	**9/9**	6	**6/6**	0	1
	NTLMSSP_NEGO	35	8	2	37	**28/35**	7	**7/8**	2	**2/2**	7	2
	NTLMSSP_AUTH	49	14	6	51	**42/49**	13	**13/14**	6	**6/6**	7	2
	Tree Connect	22	2	0	20	**19/22**	2	**2/2**	0	-	0	1
	NT Create	30	2	0	28	**27/30**	2	**2/2**	0	-	0	1
	Read	23	2	0	21	**20/23**	2	**2/2**	0	-	0	1
	Close	16	2	0	14	**13/16**	2	**2/2**	0	-	0	1

Table 2. Protocol format comparison between Wireshark and AutoFormat (Note $P \subset F \bigcup H$).

HTTP GET Query: Since the variances of protocol messages of the same type is useful in inferring the generic protocol format, we have used various web-clients (e.g., *wget*, *Firefox*, and *IE*) to generate different HTTP request messages. The results in Table 2 show that for each field identified by Wireshark, there is an identical field automatically discovered by AutoFormat. Moreover, AutoFormat performs deeper field discovery by revealing more finest-grained fields ($|F| = 23$ for AutoFormat; $|F| = 8$ for Wireshark). Though our evaluation unfavorably considers such situation as having overly-fine-grained fields, we believe our results outperform Wireshark.

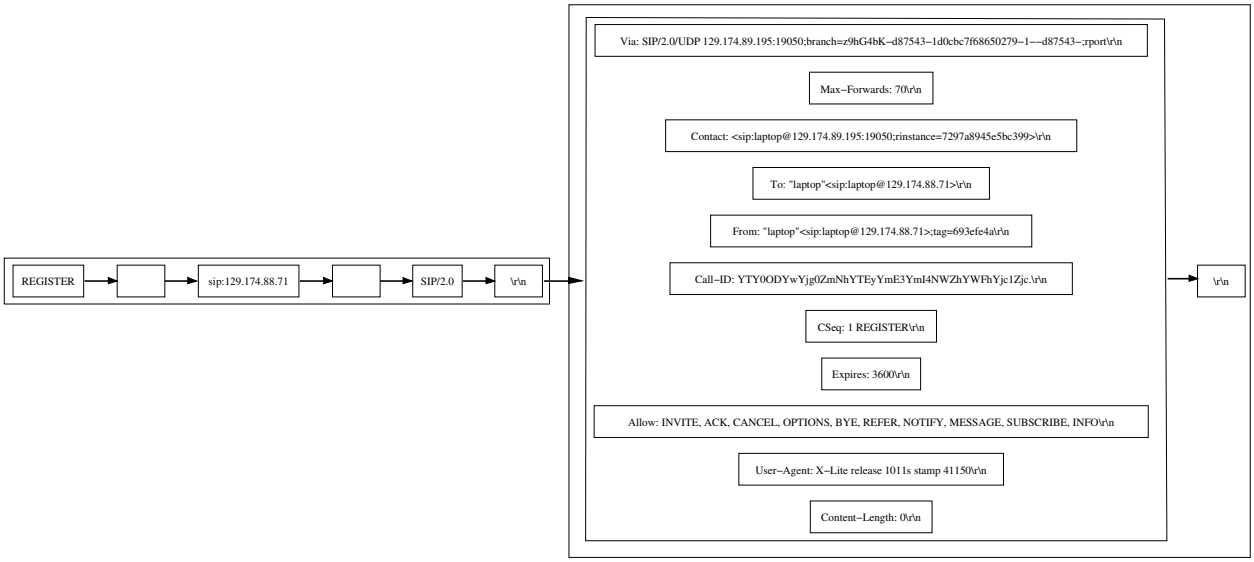
More specifically, using the `wget` request message as an example, the Wireshark result (shown in Figure 4(b)) reports $|F| = 8$, $|H| = 1$, and $|P| = 0$. Wireshark only considers the first line as a hierarchical field (with three finest-grained sub-fields) and identifies all other lines as finest-grained fields, which, based on the standard BNF definition (Figure 1), needs to be further divided into multiple sub-fields. Moreover, no parallel field has been identified despite the fact that, as dictated by both the "*" and "|" symbols in the standard BNF (Figure 1), the fields starting with keywords `User-Agent`, `Accept`, `Host`, and `Connections` are actually parallel fields.

In comparison, AutoFormat (shown in Figure 5(c)) reports $|F| = 23, |H| = 9, |P| = 4$. For each field identified by Wireshark, we can find an exact match in AutoFormat, hence $R_e(F) = R_e(H) = R_e(P) = 100\%$ (due to the fact that $|P| = 0$ in Wireshark, we mark the corresponding table entry with "-"). Detailed analysis further shows that $|F_o| = 15$ and $|F_c| = 0$. The high value of $|F_o|$ is due to the need of separating each line in the message (which is considered as one single field in Wireshark) into multiple meaningful sub-fields. As an example, AutoFormat divides the first line into six finest-grained fields while Wireshark only highlights three of them; for the second, third, and fourth lines, AutoFormat identifies four finest-grained fields in each line while Wireshark only has one field per line. Moreover, the hierarchical structure exposed by AutoFormat further reveals the overall skeleton of the message. Particularly, the identification of four parallel fields, not reported by Wireshark, is important in understanding the protocol format.

SIP REGISTER Request: In this experiment, we monitor the execution of Asterisk and trace 18 SIP messages during a successful registration session. In these 18 SIP messages, we have nine SIP REGISTER sub-messages, four SIP STATUS sub-messages, and five SIP SUBSCRIBE sub-messages. Since the structure of each sub-message of the same type is similar, we randomly choose one for each sub-message type in our evaluation.

Similar to the HTTP scenario, for each parallel or hierarchical field identified by Wireshark, there is an identical field automatically discovered by AutoFormat. As a detailed example, for the SIP REGISTER message, AutoFormat reports $|F| = 109$, $|H| = 19$, and $|P| = 12$ while Wireshark shows $|F| = 20$, $|H| = 9$, and $|P| = 11$. The vast difference is due to the fact that Wireshark does not perform deep field identification. Instead, Wireshark extracts protocol fields for the SIP protocol in a way similar to the HTTP protocol[4]. For comparison, Figures 6(a) and 6(b) show the protocol formats derived by AutoFormat and Wireshark, respectively. We also show the entire refined protocol field tree generated by AutoFormat in Figure 7.

[4]In fact, the design of the SIP protocol is heavily based on the HTTP protocol.



(a) AutoFormat result showing depth-two traversal of the refined protocol field tree ($|F| = 109$, $|H| = 19$, and $|P| = 12$)

```
Session Initiation Protocol
⊟ Request-Line: REGISTER sip:129.174.88.71 SIP/2.0
    Method: REGISTER
    [Resent Packet: False]
⊟ Message Header
  ⊟ Via: SIP/2.0/UDP 129.174.89.195:60140;branch=z9hG4bK-d87543-c6786158b34d2b08-1--d87543-;rport
      Transport: UDP
      Sent-by Address: 129.174.89.195
      Sent-by port: 60140
      Branch: z9hG4bK-d87543-c6786158b34d2b08-1--d87543-
      RPort: rport
    Max-Forwards: 70
  ⊟ Contact: <sip:laptop@129.174.89.195:60140;rinstance=3b2059786ca1bec9>
    ⊟ Contact Binding: <sip:laptop@129.174.89.195:60140;rinstance=3b2059786ca1bec9>
      ⊟ URI: <sip:laptop@129.174.89.195:60140;rinstance=3b2059786ca1bec9>
          SIP contact address: sip:laptop@129.174.89.195:60140
  ⊟ To: "laptop"<sip:laptop@129.174.88.71>
      SIP Display info: "laptop"
      SIP to address: sip:laptop@129.174.88.71
  ⊟ From: "laptop"<sip:laptop@129.174.88.71>;tag=a3178d77
      SIP Display info: "laptop"
      SIP from address: sip:laptop@129.174.88.71
      SIP tag: a3178d77
    Call-ID: Nzk2MmM3M2Y3ZGE2MGNiZGJkMDZkOGFlZjc4ZjFjMzI.
  ⊟ CSeq: 1 REGISTER
      Sequence Number: 1
      Method: REGISTER
    Expires: 3600
    Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
    User-Agent: X-Lite release 1011s stamp 41150
    Content-Length: 0
```

(b) Wireshark result ($|F| = 20$, $|H| = 9$, $|P| = 11$)

Figure 6. Comparison of AutoFormat and Wireshark results for the SIP Register message

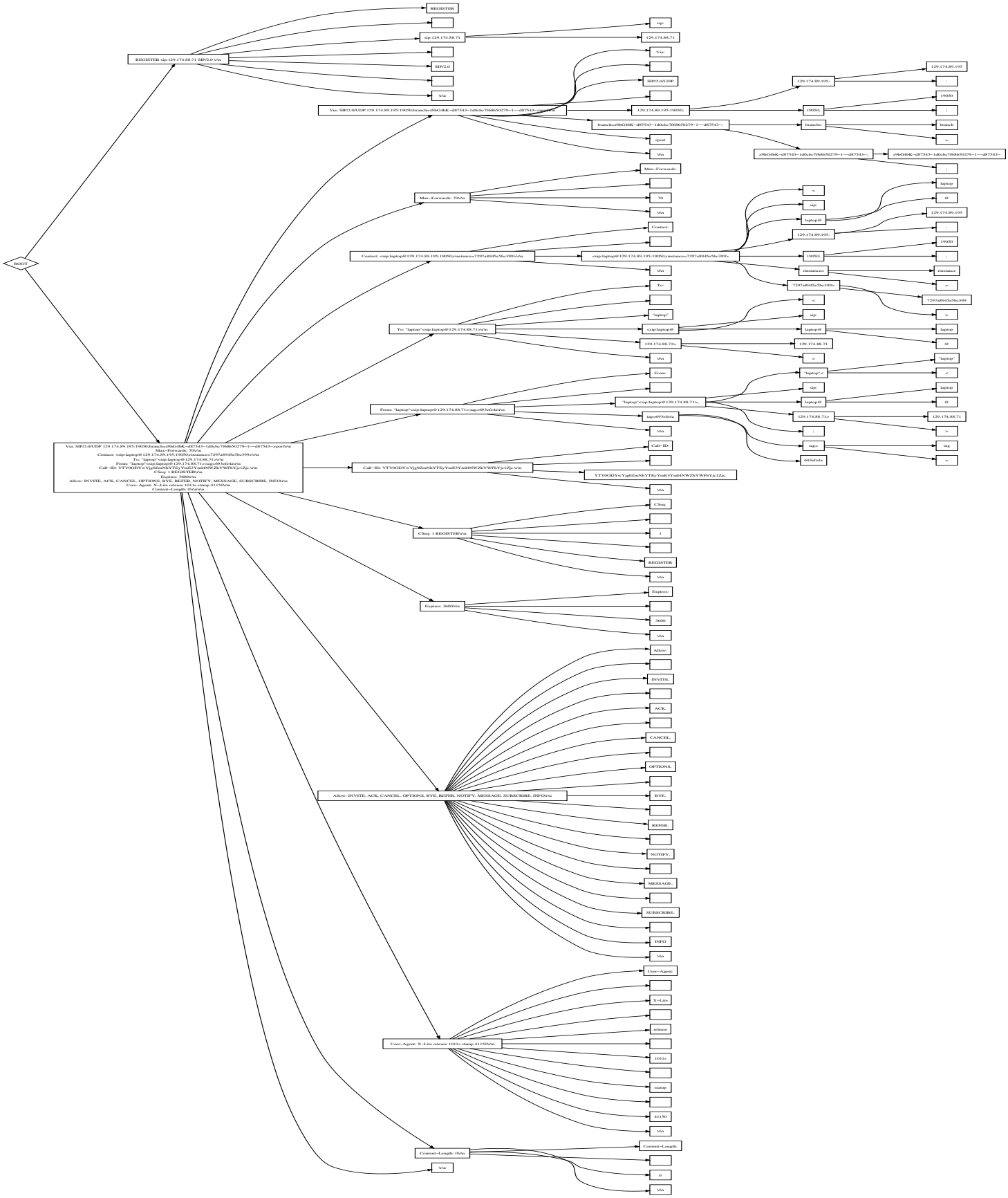


Figure 7. The refined protocol field tree for the SIP REGISTER message generated by AutoFormat

Wireshark		AutoFormat		Wireshark		AutoFormat	
Field Name	Size	Size	Match?	Field Name	Size	Size	Match?
Message type	1	1	√	Value	2	2	√
Hardware type	1	1	√	Option	1	1	√
Hardware address length	1	1	√	Length	1	1	√
Hops	1	1	√	Value	4	4	√
Transaction ID	4	6	Coarse-grained	Option	1	1	√
Seconds elapsed	2			Length	1	1	√
Bootp flags	2	2	√	Value	4	4	√
Client IP address	4	4	√	Option	1	1	√
Your (client) IP address	4	8	Coarse-grained	Length	1	1	√
Next server IP address	4			Value	14	14	√
Relay agent IP address	4	4	√	Option	1	1	√
Client hardware address	6	6	√	Length	1	20	Coarse-grained
		10	Overly-fine-grained	Value	19		
Server host name	64	192	Coarse-grained	Option	1	1	√
Boot file name	128			Length	1	8	Coarse-grained
Magic cookie	4	4	√	Hardware type	1		
Option	1	1	√	Client MAC address	6		
Length	1	1	√	End Option	1	1	√
Value	1	1	√	Padding	242	242	√
Option	1	1	√				
Length	1	1	√				

Table 3. Detailed comparison between Wireshark and AutoFormat on the finest-grained fields identified for the DHCP BOOTP Request message ($|F_{Wireshark}| = 39, |F_{AutoFormat}| = 34$).

DHCP BOOTP Request: In this experiment, we monitor the execution of the dhcpd daemon and study the BOOTP request message. Table 2 shows that AutoFormat reports $|F| = 34$, $|H| = 9$, and $|P| = 7$ while Wireshark reports $|F| = 39$, $|H| = 8$, and $|P| = 7$. Note that AutoFormat and Wireshark have identified almost identical hierarchical fields and parallel fields. But they differ in the extraction of the finest-grained fields. More specifically, among the 39 finest-grained fields by Wireshark, only 28 of them are discovered by AutoFormat. To understand the reason, we perform a detailed comparison between these finest-grained fields and the results are shown in Table 3.

The table shows the existence of coarse-grained fields as well as an overly-fine-grained field in the AutoFormat results. To find out the root cause, we map the message payload to the corresponding application handling code and find out that there are two main reasons behind the coarse-grained fields: (1) Certain fields of the payload are simply zeroed place-holders and the server code does not need to handle them. For example, the fields Φ(`Your (Client) IP Address`), Φ(`Next Server IP Address`), Φ(`Server host name`), and Φ(`Boot file name`) are place-holders within this particular request message. As a result, they are usually zeroed out. (2) The application code is programmed to ignore certain fields in the message payload. For example, when handling the BOOTP request message, the daemon analyzed simply ignores the fields such as Φ(`Transaction ID`) and Φ(`Seconds elapsed`). Note that the second reason is implementation-specific and the coarse-grained fields identified could be useful for a variety of applications, such as protocol compliance checking [7], application fingerprinting [8], and fuzz testing [29].

The existence of the overly-fine-grained field ($|F_o| = 1$), interestingly, exposes the wrong interpretation of the BOOTP Request message in the current Wireshark implementation. Note that the application source code does declare the field as a 16-byte field and our input data is six none-zero bytes followed by 10 zero bytes. When Wireshark handles this field, it only interprets the first six bytes and declares a Φ(`Client hardware address`) field with only six bytes. According to RFC 2131 for DHCP, the Φ(`Client hardware address`) field is indeed 16 bytes long.

SMB Negotiate Request: In this experiment, we collect 30 SMB messages in a user session when a Windows-based client successfully opens a directory in a remote Linux server. For the SMB Negotiate Request message, AutoFormat reports $|F| = 24$, $|H| = 9$, and $|P| = 6$ while Wireshark reports $|F| = 26$, $|H| = 9$, and $|P| = 6$. For comparison, we show the detailed fields in Table 4. The only difference is one coarse-grained field: the three finest-grained fields identified by Wireshark – Φ(`Process ID High`), Φ(`Signature`) and Φ(`Reserved`) – are consolidated into one field by AutoFormat. Similar to the previous experiment, the main reason is that the server program simply ignores these fields.

RIP and OSPF messages: We experiment with RIP and OSPF protocols by monitoring the execution of the Zebra routing software. More specifically, we analyze two RIP messages and four OSPF messages. As shown in Table 2, we have identical results for the `RIPv2 Response`

Wireshark		AutoFormat		Wireshark		AutoFormat	
Field Name	Size	Size	Match?	Field Name	Size	Size	Match?
Server Component	4	4	√	Byte Count	2	2	√
SMB Command	1	1	√	Buffer Format	1	1	√
NT Status	4	4	√	Name	23	23	√
Flags	1	1	√	Buffer Format	1	1	√
Flags2	2	2	√	Name	10	10	√
Process ID High	2	12	Coarse-grained	Buffer Format	1	1	√
Signature	8			Name	28	28	√
Reserved	2			Buffer Format	1	1	√
Tree ID	2	2	√	Name	10	10	√
Process ID	2	2	√	Buffer Format	1	1	√
User ID	2	2	√	Name	10	10	√
Multiplex ID	2	2	√	Buffer Format	1	1	√
Word Count	1	1	√	Name	11	11	√

Table 4. Detailed comparison between Wireshark and AutoFormat on the finest-grained fields identified for the Samba Negotiate Request message ($|F_{Wireshark}| = 26, |F_{AutoFormat}| = 24$).

message and the `OSPF Hello` message. For the `RIPv2 Request` message, we have one coarse-grained field, which corresponds to three sub-fields in Wireshark. Similarly, we have one coarse-grained field for the `OSPF LS Request`. For the other two message types, the AutoFormat results contain two coarse-grained fields. The reason for these unmatched coarse-grained fields is the same as the DHCP case: The corresponding message bytes are not further processed by the software implementation. Note that when we experiment with the RIP protocol, we deliberately compile it without the stack-frame pointer support and we strip all the symbols in the generated binary. AutoFormat still works thanks to its execution context capture technique (Section 3.1).

Slapper Worm Messages: We now present our second set of experiments showing that AutoFormat fully uncovers the message format of an unknown protocol used by the Slapper worm.

It has been reported that the Slapper worm can self-organize infected hosts into a P2P attack network [16]. However, to the best of our knowledge, the exact protocol format used by the Slapper worm has not been fully investigated or reported. In this experiment, we use AutoFormat to reverse engineer the message format of the unknown protocol. Specifically, we launch the Slapper worm inside a virtual worm playground environment [16] and a special master program [6] capable of issuing commands to the attack network is deployed, connecting to only one infected host. Meanwhile, it is interesting that in each Slapper worm propagation session, it will replicate itself with a copy of its source code. As such, we can statically analyze the source code and manually identify all the message fields, which will be used to verify the AutoFormat results.

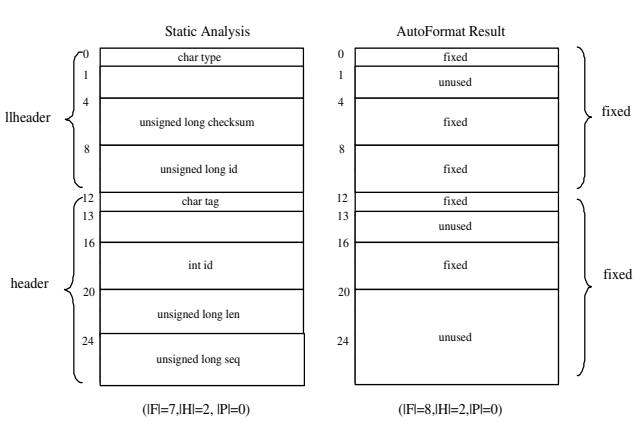
AutoFormat has identified 11 message types and the results are presented in Table 5. It is encouraging that AutoFormat results match our static analysis results well. In the following, we detail our results for two specific message types. These two types of messages are sent by the master program when issuing the following commands: (1) The *LIST* command is used to list all members in the P2P attack network; (2) The *DNS Flood* command is used to launch a DNS flooding attack. The detailed format of these two message types, derived by static analysis and by AutoFormat, are shown in Figure 8.

Our static analysis shows that for the *LIST* command, the message has two high-level fields that are defined with two data structures: *struct llheader* and *struct header*. *struct llheader* has three member fields: *char type*, *unsigned long checksum*, and *unsigned long id*. It is interesting to point out that the compiler automatically pads the first *char* member to 4 bytes, which is identified by AutoFormat as two finest-grained fields (one at offset 0 and another at offsets 1-3). The same reason holds for the first member – *char tag* – in the *struct header* data structure. The only remaining difference is the coarse-grained field at offsets 20-27. By checking its source code, we find out that the Slapper worm simply ignores this field.

For the *DNS Flood* message, there are two differences in the static analysis and AutoFormat results: AutoFormat identifies two additional finest-grained fields (one at offset 1-3 and another at offsets 13-15) and misses one hierarchical field (at offsets 12-25). As explained earlier, the two finest-grained fields are introduced by the compiler. The missing hierarchical field is related to the *struct header* data structure which is embedded as a member within a higher-level *struct df_rec* data structure. Note that the higher-level hierarchical field (offset 12-31) is correctly uncovered by AutoFormat. The main reason for missing the nested *struct header* field is that the worm binary uses the same execution context to handle all members in the *df_rec* data structure.

Protocol	Request Msg Type	Static Analysis			AutoFormat						Analysis of F	
		$\|F\|$	$\|H\|$	$\|P\|$	$\|F\|$	$R_e(F)$	$\|H\|$	$R_e(H)$	$\|P\|$	$R_e(P)$	$\|F_o\|$	$\|F_c\|$
Unknown protocol used by Slapper worms	LIST command	7	2	0	8	**5/7**	2	**2/2**	0	-	0	1
	INFO command	9	3	0	10	**7/9**	2	**2/3**	0	-	0	1
	SH Command	8	2	0	10	**8/8**	2	**2/2**	0	-	0	0
	UDP Flood	11	3	0	12	**9/11**	2	**2/3**	0	-	0	1
	TCP Flood	10	3	0	11	**8/10**	2	**2/3**	0	-	0	1
	DNS Flood	9	3	0	11	**9/9**	2	**2/3**	0	-	0	0
	Email Scan	8	3	0	8	**6/8**	2	**2/3**	0	-	0	1
	GetMyIP*	8	3	0	7	**5/8**	2	**2/3**	0	-	0	1
	Incoming Client*	7	2	0	5	**4/7**	2	**2/2**	0	-	0	1
	Route*	10	3	0	13	**10/10**	2	**2/3**	0	-	0	0
	Info Packet*	18	3	0	19	**16/18**	2	**2/3**	0	-	0	1

Table 5. Comparison between static analysis and AutoFormat results for the unknown protocol used by Slapper worm: The rows marked by "*" represent those messages exchanged between worm-infected hosts; while other unmarked rows represent the messages exchanged between a worm-infected host and the special master program. (Note $P \subset F \bigcup H$).



(a) Message format comparison for the *LIST* command

(b) Message format comparison for the *DNS Flood* command

Figure 8. Comparison of Slapper worm message formats derived by static analysis and by AutoFormat

5 Related Work

In this section, we describe related work and compare it with AutoFormat. Note that AutoFormat relies on the generic technique of dynamic taint analysis. Since the technique itself has been widely investigated and there exists a large body of recent work in taint analysis, we omit detailed discussion in this section. Interested readers are referred to [10, 11, 15, 23, 26, 27, 31].

Sharing the same goal of automating protocol reverse engineering, the Protocol Informatics (PI) project [3], Discoverer [12], Polyglot [9], and [30] by Wondracek et al. are closely related to our work. Particularly, from a network perspective, both PI and Discoverer aim at extracting protocol format from collected network traces. They have the advantage of convenient trace collection. But the lack of program semantics in network traces significantly limits the accuracy of extracted protocol formats. Moreover, they become ineffective in the face of encrypted network traffic.

From a host perspective, both Polyglot [9] and the solution in [30] share the key insight that the way a protocol is implemented to recognize and handle protocol messages provides a wealth of information about the protocol format. AutoFormat differs from them in its way of extracting protocol format. By recognizing and leveraging the field-specific execution context, AutoFormat collects and analyzes run-time execution context information to infer protocol format. Nonetheless, it is possible to integrate all the above host-based solutions to achieve better accuracy and completeness in protocol reverse engineering.

There exist other related techniques designed for other purposes but having the ability, at least to some extent, of

extracting certain protocol format. For example, application dialog replay systems such as RolePlayer [13], ScriptGen [17, 18], and Replayer [22] share the need of identifying and updating certain input fields that are embedded in a captured protocol session. Particularly, by leveraging byte-wise sequence alignment from network traces, RolePlayer and ScriptGen heuristically identify and adjust some specific fields; Replayer instead leverages host-based binary analysis to enable automatic protocol replay. These systems aim at discovering only partial protocol format to serve the purpose of replaying the protocol dialog, whereas AutoFormat aims at extracting the entire protocol message format and revealing possible cross-field relations.

Protocol identification proposed by Ma et al. [20] also has the ability of inferring partial protocol format (e.g., the format of the first 64 bytes of a protocol session data). However, their approach is purely based on network traces thus sharing the same limitations as the other network trace-based approaches [3, 12]. Protocol analyzers such as Wireshark [5] have the capability of properly formating a protocol message, but they require prior knowledge about those protocols and thus are of less use when analyzing unknown protocols.

6 Limitations and Future Work

The first limitation of AutoFormat is the dynamic trace dependency. Since AutoFormat relies on execution traces, the protocol format it derives depends on the diversity of input recorded in the execution trace. In other words, if a certain message format never occurs in the trace, it is impossible for AutoFormat to infer that format.

Secondly, our current prototype only analyzes protocol format at the byte granularity and is thus unable to discover protocol fields at the bit level. However, this limitation can be removed by re-implementing existing memory marking and propagation technique at the bit-level. Our current prototype is not flexible enough in distinguishing between text and binary protocols. However, the tokenization process proposed in Discoverer can potentially be adopted for automatic recognition.

Thirdly, AutoFormat analyzes each message in isolation and does not correlate multiple messages in the same protocol session. Extending AutoFormat to further reconstruct the entire protocol state machine is part of our future work.

Finally, AutoFormat cannot yet handle obfuscated binaries. However, since such binaries still need to interpret and handle incoming messages, it is inevitable that certain data and control dependencies still exist in the obfuscated code. Such dependencies may be leveraged to overcome the difficulties caused by the obfuscation techniques.

7 Conclusion

We have presented AutoFormat, a system for automatic protocol format extraction. AutoFormat is based on the insight that a protocol implementation is programmed to recognize the protocol format and usually creates protocol field-specific execution contexts. By instrumenting and monitoring program execution, we can obtain the execution context information and use it as clustering criteria to identify protocol fields and their relations. We have implemented a prototype of AutoFormat and evaluated it with a variety of protocol messages from seven real-world (known or unknown) protocols. Our experimental results show that AutoFormat achieves high accuracy in protocol field identification and message format reconstruction.

References

[1] How Samba Was Written. *http://samba.org/ftp/tridge/misc/french_cafe.txt*.

[2] QEMU: an open source processor emulator. *http://www.qemu.org/*.

[3] The Protocol Informatics Project. *http://www.baselineresearch.net/PI/*.

[4] The SNORT network intrusion detection system. *http://www.snort.org*.

[5] Wireshark: The World's Most Popular Network Protocol Analyzer. *http://www.wireshark.org/*.

[6] PUD: Peer-To-Peer UDP Distributed Denial of Service. *http://www.packetstormsecurity.org/distributed/pud.tgz*, 2002.

[7] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A generic application-level protocol analyzer and its language. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, San Diego, CA, February 2007.

[8] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *In Proceedings of the 16th USENIX Security Symposium (Security'07)*, Boston, MA, August 2007.

[9] J. Caballero and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, 2007.

[10] J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole-system simulation. 2004.

[11] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

[12] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, Boston, MA, August 2007.

[13] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS'06)*, San Diego, CA, February 2006.

[14] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *In Proceedings of 2007 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

[15] M. Egele, C. Kruegel, E. Kirda, H. Yin, , and D. Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (Usenix'07)*, June 2007.

[16] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual Playgrounds for Worm Behavior Investigation. *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, Sept. 2005.

[17] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots. In *9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, pages 185–205, 2006.

[18] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 203–214, Washington, DC, USA, 2005.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200, 2007.

[20] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected means of protocol inference. In *Proceedings of the 6th ACM SIGCOMM on Internet measurement (IMC'06)*, pages 313–326, New York, NY, USA, 2006. ACM Press.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, USA, 2007. ACM Press.

[22] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM Conference on Computer and and Communications Security (CCS'06)*, 2006.

[23] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)*, San Diego, CA, February 2005.

[24] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks, 31(23-24):2345-2463*, 1999.

[25] F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf*.

[26] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, Boston, Massachusetts, 2004.

[27] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, San Diego, CA, February 2007.

[28] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM '04*, pages 193–204, 2004.

[29] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS'06)*, pages 37–46, New York, NY, USA, 2006. ACM Press.

[30] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic Network Protocol Analysis. *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[31] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

Appendix I

Figure 9 shows an excerpt of the pre-processed log file generated by AutoFormat.

Offset	Content	Stack	Location
0	'G'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
1	'E'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
2	'T'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
3	' '	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
4	'/'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
5	'n'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
6	'e'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
7	'w'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core->ap_get_brigade->0x2D2CE->ap_get_brigade->0x2D667 ->apr_brigade_split_line->memchr	0x4BA56A2
...			
24	'\n'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core	0x26187
23	'\r'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core	0x26322
22	'0'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_rgetline_core	0x261B3
0	'G'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_getword_white	0x1F7F3
1	'E'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_getword_white	0x1F7F3
2	'T'	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_getword_white	0x1F7F3
3	' '	main->ap_mpm_run->0x15C57->0x15B38->0x15941->ap_process_connection->ap_run_process_connection ->0xF5A8->ap_read_request->ap_getword_white	0x1F7F3
...			

Figure 9. Sample log entries collected and cleansed by AutoFormat when monitoring the Apache web server processing an incoming HTTP Request message