ReFormat: Automatic Reverse Engineering of Encrypted Messages

Zhi Wang[1], Xuxian Jiang[1], Weidong Cui[2], Xinyuan Wang[3], and Mike Grace[1]

[1] North Carolina State University {zhi_wang,xjiang4,mcgrace}@ncsu.edu
[2] Microsoft Research wdcui@microsoft.com
[3] George Mason University xwangc@gmu.edu

Abstract. Automatic protocol reverse engineering has recently received significant attention due to its importance to many security applications. However, previous methods are all limited in analyzing only plain-text communications wherein the exchanged messages are not encrypted. In this paper, we propose ReFormat, a system that aims at deriving the message format even when the message is encrypted. Our approach is based on the observation that an encrypted input message will typically go through two phases: message decryption and normal protocol processing. These two phases can be differentiated because the corresponding instructions are significantly different. Further, with the help of data lifetime analysis of run-time buffers, we can pinpoint the memory locations that contain the decrypted message generated from the first phase and are later accessed in the second phase. We have developed a prototype and evaluated it with several real-world protocols. Our experiments show that ReFormat can accurately identify decrypted message buffers and then reveal the associated message structure.
Keywords: Security, Reverse Engineering, Network Protocols, Data Lifetime Analysis, Encryption

1 Introduction

With great potentials to many security applications, protocol reverse engineering has recently received significant attention. For example, network-based firewalls or filters [1, 2] require the knowledge of protocol specifications to understand the context of a particular network communication session. Similarly, fuzz testing [3] of unknown protocols can utilize the same knowledge to improve the fuzzing process by generating interesting inputs more efficiently.

Traditionally, protocol reverse engineering was mostly a manual process that is time-consuming and error-prone. To alleviate this situation, a number of systems [4–9] have been developed to allow for automatic protocol reverse engineering. The Protocol Informatics [4] project and Discoverer [6] take a network-based approach and locate field boundaries from a large amount of network traces by leveraging the sequence alignment algorithm that has been used in bioinformatics for pattern discovery. Other systems, such as Polyglot [5], the work [9] by Wondracek *et al.*, AutoFormat [8], Tupni [7], and Prospex [10], take a program-based approach to find out the message format. While different in various regards, these program-based systems all operate using the same insight: how a program parses and processes a message reveals rich information about the message format.

Despite all the advances made by these systems, there still exists one major common limitation: they are unable to analyze encrypted messages. Particularly, network-based approaches are unable to identify the format of encrypted messages because the collected network traces are in the form of cipher-text, which completely destroys message field boundaries and thus unlikely exhibits any common patterns at the network packet level. Existing program-based approaches are also unable to achieve their goals on encrypted messages because it is not the input message whose format we try to discover, but the decrypted one that is generated at run-time. Unfortunately, none of the existing program-based approaches is able to accurately locate the run-time memory buffers that contain the decrypted plain-text message. From another perspective, we need to point out that, once the decrypted message is determined, we can still apply the very same insight behind these program-based approaches to extract the corresponding protocol format, i.e., by analyzing how the plain-text message is parsed in the normal protocol processing phase.

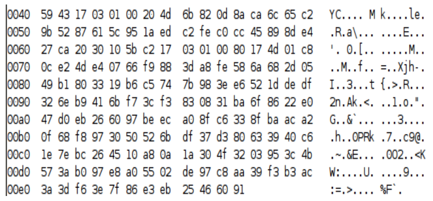
In this paper, we propose ReFormat, a program-based system that can accurately identify the run-time buffers that contain the decrypted message. Our approach is based on the observation that an encrypted input message will typically go through two main processing phases: message decryption and normal protocol processing. And *the instructions used for decrypting an encrypted message are significantly different from those used for processing a normal unencrypted protocol message.* As such, we can identify and separate the message decryption phase from the normal protocol processing phase based on the distribution of executed instructions. Further, we observe that decrypted messages are first generated from the message decryption phase and then processed in the normal protocol processing phase. Based on this observation, we can accordingly perform *data lifetime analysis* of run-time buffers that are generated from the message decryption phase to pinpoint the memory buffers that contain the decrypted message. Once the decrypted message is identified, we can take one of previous approaches [5, 7–9] to analyze how it is being handled to discover its format.

We have implemented a prototype of ReFormat and evaluated it with four protocols that encrypt (or encode) their network communications: HTTPS, IRC, MIME, and one unknown protocol used by a real-world malware. For all these test cases, ReFormat can pinpoint with high accuracy the run-time buffers that contain the decrypted message, and then identify its format.

The rest of the paper is organized as follows. In Section 2, we describe the problem scope as well as associated challenges. We present the system design and key techniques for identifying run-time buffers of the decrypted message in Section 3. In Section 4, we show the evaluation results. After discussing the related work in Section 5, we examine limitations of ReFormat and suggest possible improvement in Section 6. Finally, we conclude this paper in Section 7.

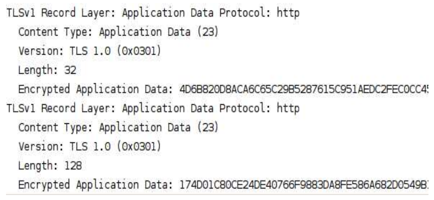
2 Problem Overview

To achieve the goal of automatic protocol reverse engineering, an important step is to derive the protocol message structure. As mentioned earlier, existing approaches have explored various solutions to uncover the structure of plain-text messages. However, they cannot be applied to understand the structure of encrypted messages. As a concrete example, Figure 1 shows an encrypted web request message that is captured in a typical HTTPS session. Specifically, Figure 1(a) shows the raw data of the web request message and Figure 1(b) illustrates the message fields decoded by Wireshark. These figures show that the request message is encapsulated in the Transport Layer Security (TLS) record layer and fragmented into two TLS encryption records. However, what we want to reverse engineer is the HTTP request (shown in Figure 1(b)) encrypted in this message. Recall that all previous protocol reverse engineering methods can only recover the format of plain-text message. One gap in recovering the format of encrypted message is how to recover the plain-text message from the cipher-text message. The goal of ReFormat is to fill this gap so that all previous program-based approaches can handle encrypted messages as well as plain-text ones.



```
0040  59 43 17 03 01 00 20 4d  6b 82 0d 8a ca 6c 65 c2   YC.... M k....le.
0050  9b 52 87 61 5c 95 1a ed  c2 fe c0 cc 45 89 8d e4   .R.a\... ....E...
0060  27 ca 20 30 10 5b c2 17  03 01 00 80 17 4d 01 c8   '. 0.[.. .....M..
0070  0c e2 4d e4 07 66 f9 88  3d a8 fe 58 6a 68 2d 05   ..M..f.. =..Xjh-.
0080  49 b1 80 33 19 b6 c5 74  7b 98 3e e6 52 1d de df   I..3...t {.>.R...
0090  32 6e b9 41 6b f7 3c f3  83 08 31 ba 6f 86 22 e0   2n.Ak.<. ..1.o.".
00a0  47 d0 eb 26 60 97 be ec  a0 8f c6 33 8f ba ac a2   G..&`... ...3....
00b0  0f 68 f8 97 30 50 52 6b  df 37 d3 80 63 39 40 c6   .h..0PRk .7..c9@.
00c0  1e 7e bc 26 45 10 a8 0a  1a 30 4f 32 03 95 3c 4b   .~.&E... .0O2..<K
00d0  57 3a b0 97 e8 a0 55 02  de 97 c8 aa 39 f3 b3 ac   W:....U. ....9...
00e0  3a 3d f6 3e 7f 86 e3 eb  25 46 60 91               :=.>.... %F`.
```

(a) An encrypted web request message captured by TCPDUMP



```
TLSv1 Record Layer: Application Data Protocol: http
  Content Type: Application Data (23)
  Version: TLS 1.0 (0x0301)
  Length: 32
  Encrypted Application Data: 4D6B820D8ACA6C65C29B5287615C951AEDC2FEC0CC4
TLSv1 Record Layer: Application Data Protocol: http
  Content Type: Application Data (23)
  Version: TLS 1.0 (0x0301)
  Length: 128
  Encrypted Application Data: 174D01C80CE24DE40766F9883DA8FE586A682D0549B
```

(b) The protocol format identified by Wireshark

Fig. 1. An encrypted web request message and its protocol format identified by Wireshark

To fill the gap, there are several challenges: First, the memory buffers that contain the decrypted message are not known a priori as they can be dynamically allocated from the heap or the stack. This is different from the previous cases with plain-text messages where the memory buffers of the input message can be easily identified and monitored — as they are typically associated with particular system calls such as *sys_read*. Second, even worse, the target buffers can be buried in hundreds or thousands of other memory buffers inside the same memory space of a running process. Intuitively, we can reduce the number of target buffers by using taint analysis [11, 12] to locate only those tainted buffers from the input messages. Our experience indicates that it is reasonably effective but we still observe tens or even hundreds of tainted buffers (Table 2 in Section 4). In other words, new heuristics still need to be developed to further prune tainted buffers. Finally, the decrypted memory buffers may only exist for a short period of time as they could be discarded or reclaimed back for other purposes right after the processing.

3 System Design

3.1 Design Overview

Given an encrypted message and an application that can decrypt and process it, our system aims to output the content and format of the decrypted message. Since an encrypted input message will be first decrypted and then processed, there is a need to delineate these two main phases, i.e., message decryption and normal protocol processing. To achieve that, our approach is based on an intuitive observation: *The instruction distribution of the message decryption phase and the normal protocol processing phase are significantly different.* Existing cryptography algorithms such as Triple-DES, AES and RC4 typically contain a large amount of arithmetic and bitwise operations and they will be applied to all the bytes in the original messages. As an example, Figure 2 shows a code snippet of the function *AES_decrypt()* from a real-world AES-based decryption implementation in the *OpenSSL* cryptographic library. When decrypting one block of an input message, it involves at least nine rounds of calculation and each round contains a large amount of arithmetic and bitwise operations such as logical right shift and xor. In addition, this particular function will be applied to every block of the encrypted message. In comparison, in the normal protocol processing phase, we are likely to observe significantly less arithmetic and bitwise instructions. To validate this observation, we have profiled the execution of representative decryption algorithms that are implemented in the OpenSSL library and compare the results with a number of existing applications that handle unencrypted messages of known protocols (or formats). The comparison (shown in Table 1) demonstrates that there exists a significant difference in the percentage of arithmetic and bitwise operations between message decryption and normal protocol processing. On one hand, more than 80% of instructions are arithmetic and bitwise operations when an encrypted input message is being decrypted. On the other hand, less than 25% of instructions are arithmetic and bitwise operations when a normal plain-text protocol message is being processed. This empirically confirms our intuitive observation.

```
void AES_decrypt(...)
{
    ...

    /* round 1: */
    t0 = Td0[s0 >> 24] ^ Td1[(s3 >> 16) & 0xff]  ^
         Td2[(s2 >>  8) & 0xff] ^ Td3[s1 & 0xff] ^ rk[ 4];
    t1 = Td0[s1 >> 24] ^ Td1[(s0 >> 16) & 0xff]  ^
         Td2[(s3 >>  8) & 0xff] ^ Td3[s2 & 0xff] ^ rk[ 5];
    t2 = Td0[s2 >> 24] ^ Td1[(s1 >> 16) & 0xff]  ^
         Td2[(s0 >>  8) & 0xff] ^ Td3[s3 & 0xff] ^ rk[ 6];
    t3 = Td0[s3 >> 24] ^ Td1[(s2 >> 16) & 0xff]  ^
         Td2[(s1 >>  8) & 0xff] ^ Td3[s0 & 0xff] ^ rk[ 7];

    /* round 2: */
    s0 = Td0[t0 >> 24] ^ Td1[(t3 >> 16) & 0xff] ^
         Td2[(t2 >>  8) & 0xff] ^ Td3[t1 & 0xff] ^ rk[ 8];
    ...
    /* round 3: */

    ...
}
```

Fig. 2. Code snippet from the OpenSSL-based AES decryption implementation

To achieve our goal, our system takes four key steps as shown in Figure 3: (1) Execution Monitor: We first monitor the application execution and collect an

Table 1. The percentages of arithmetic and bitwise operations in typical implementations of existing decryption algorithms and normal programs that handle known plaintext protocol messages (†: As discussed in Section 3.2, we only count those instructions that operate on the input message.)

Encryption/Message Type	Message Size (B)	Arithmetic & Bitwise Instructions†	Total Instructions†	Percentage
DES	2K	68921	69112	99.72%
CAST	2K	18917	21225	89.13%
RC4	2K	2709	3042	89.05%
AES	2K	6892	8475	81.32%
HTTP request	107	429	3227	13.29%
FTP port	28	421	5898	7.14%
DNS response	46	223	1687	13.22%
RPC bind	164	186	2342	7.94%
JPEG	3224	1112	12898	8.62%
BMP	3126	229	956	23.95%

execution trace recording how the application decrypts and processes an input message. (2) Phase Profiler: We then analyze the execution trace to identify the two execution phases: *message decryption* and *normal protocol processing*. (3) Data Lifetime Analyzer: After that, we perform data lifetime analysis to locate buffers that contain the decrypted message. (4) Format Analyzer: Finally, we conduct dynamic data flow analysis on the buffers located in the previous step to uncover the format of the decrypted message. Since the last step has been extensively studied in previous work [5, 7, 9, 8], we focus on the first three steps in this paper. In our prototype, we use AutoFormat [8] as our format analyzer but other systems [5, 7, 9] should be equally applicable for the same purpose.

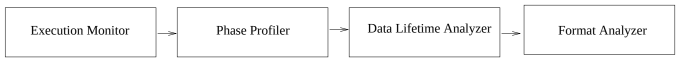


Fig. 3. ReFormat System Architecture

In the rest of this section, we will describe the execution monitor, phase profiler, and data lifetime analyzer in detail. To help illustrate our approach, we will use a running example. In the running example, an *shttpd* web server [13] processes an encrypted HTTP request issued by *wget*, an HTTP client. The raw data of the encrypted request message is shown in Figure 1(a).

3.2 Execution Monitor

Similar to other program-based approaches, by monitoring a program's execution, ReFormat aims to record how an input message is being processed by the program. In particular, by intercepting system calls that are used to read from and write to file descriptors and/or network sockets, ReFormat taints the input message and applies the well-known taint analysis technique to keep track of the instructions that access tainted memory space. By dynamically instrumenting the program execution, the taint information can be properly propagated and a trace of the instructions that operate on tainted data will be collected. We highlight that the collected trace contains *only* the instructions that operate on the marked data, rather than all executed instructions. Inside the trace, we record the address of the instruction and the current call stack when the instruction occurs. Note that the run-time call stack information is important for ReFormat. As to be shown in the following subsection, such context information is used in the phase profiler to determine the transition point between the message decryption phase and the normal protocol processing phase. In our system, to acquire the run-time call stack, we mainly traverse the current stack frames and retrieve the caller/callee information from the procedure-related activation record on the stack. If the debug information is embedded in the binary, we will derive the related function names. This works well for the program or library built with stack frame pointer support. For a binary compiled without stack frames, we can still build a shadow call stack by instrumenting the call/return instructions. Similar to previous work, we assume the boundaries of network messages can be identified, and therefore an execution trace contains the processing of a single input message.

3.3 Phase Profiler

After collecting an execution trace, we divide it into different execution phases in the phase profiler. An application usually processes an encrypted input message and responds with an encrypted output message in four phases: (1) decrypt the input message, (2) process the decrypted message, (3) generate the output message, (4) encrypt the output message. Since our goal is to identify the decrypted message (and then uncover its format), we only need to recognize the boundary between the first two phases. For simplicity of presentation, we refer to the first phase as the "message decryption" phase, and refer to the last three phases aggregately as the "normal protocol processing" phase. To divide an execution trace into these two phases, we search for the transition point between them, i.e., the last instruction executed in the message decryption phase.

We perform the search in two steps. Our first step is to use the cumulative percentage of arithmetic and bitwise instructions to narrow down the search range where the transition point is located. Here, the cumulative percentage of arithmetic and bitwise instructions at the n-th instruction is defined to be the percentage of arithmetic and bitwise instructions in the first n instructions. Note that an application may still use a large amount of arithmetic and bitwise operations to encrypt the output message at the end of an execution trace. However, the cumulative percentage during encryption is likely to be lower than the percentage in the message decryption phase. The reason is that, before the output message is encrypted, the application, when processing the decrypted message and then generating a plain-text output message, will likely introduce a significant amount of instructions that are neither arithmetic nor bitwise. As such, we expect the cumulative percentage to reach its peak value in the message decryption phase and to drop to its lowest value in the normal protocol processing phase. In other words, the transition point must be between the instruction with the maximum cumulative percentage and the one with the minimum percentage. After identifying these two instructions in the execution trace, we refer to them as the *maximum instruction* and the *minimum instruction*.

After identifying the maximum and minimum instructions based on the cumulative percentage, our second step is to compute the percentage of arithmetic and bitwise instructions for each *function fragment* between them. Here, a *function fragment* is defined to contain contiguous instructions that belong to the same function and are executed in the same context (or under the same runtime stack frame). For instance, if a parent function A calls a child function B and there is no function called in B, we will have three function fragments, F_{A1}, F_B, and F_{A2}, where F_{A1} contains all instructions in A executed before B is called and F_{A2} contains all instructions in A executed after B returns. An important property is that each instruction in the execution trace belongs to one and only one function fragment. For the maximum and minimum instructions identified previously, we refer to their function fragments as the *maximum function fragment* and the *minimum function fragment*.

We point out that our second step uses the fragment-wise percentage instead of the cumulative percentage because the function fragments for *actual* message decryption are likely to have high fragment-wise percentage. Therefore we identify the last function fragment whose percentage is above a given threshold as the transition function. The last instruction executed in this fragment will be used as the transition point. In our prototype, based on the percentages of arithmetic and bitwise operations shown in Table 1, we set the threshold to be 50%[4]. As to be shown in Section 4, this threshold works well in all test cases.

Meanwhile, we anticipate that, in certain applications, there may not exist a function boundary between the message decryption phase and the normal protocol processing phase. For example, some protocol implementation may put message decryption and processing into a single big function. In this case, we can alternatively compute the percentage on a sliding window to determine the transition point. Specifically, we can have a sliding window on each instruction and then treat each sliding window as a function fragment to compute the fragment-wise percentage of arithmetic and bitwise instructions. However, since we do not encounter such cases in the evaluation, we have not explored the selection of the sliding window size in this paper.

In our running example, the cumulative percentage of arithmetic and bitwise instructions is shown in Figure 4. The X-axis is the fragments in the temporal order, and the Y-axis is the cumulative percentage. At the very beginning, there is a steady increase of the cumulative percentage of arithmetic and bitwise instructions until it reaches the peak value at an instruction inside the function fragment *sha1_block_asm_data_order*. After that, the cumulative percentage keeps decreasing until it reaches the lowest value at an instruction inside the function fragment *HMAC_Init_ex*. In Figure 5 we show the fragment-wise percentage of arithmetic and bitwise instructions for function fragment executed between *sha1_block_asm_data_order* and *HMAC_Init_ex*. Given our threshold, we identify the last invocation of *sha1_block_asm_data_order* as the transition function, which is consistent with our manual analysis of the shttpd source code. Also, in this running example, we found that more than 99% of arithmetic in-

[4] In fact, any value between 25% and 80% works the same way in our evaluation.

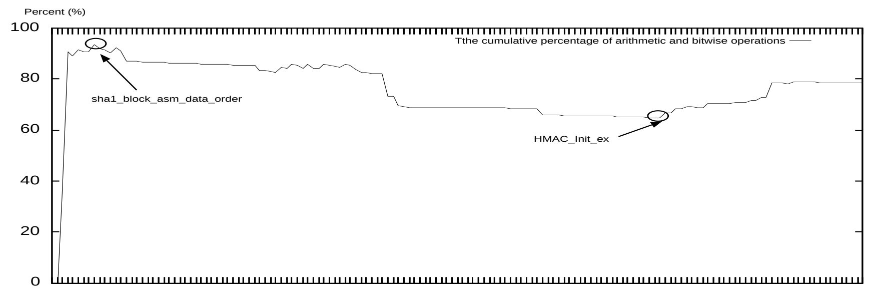


Fig. 4. Phase Profiler (Step I): Calculating the cumulative percentage of arithmetic and bitwise operations in the collected shttpd-based execution trace

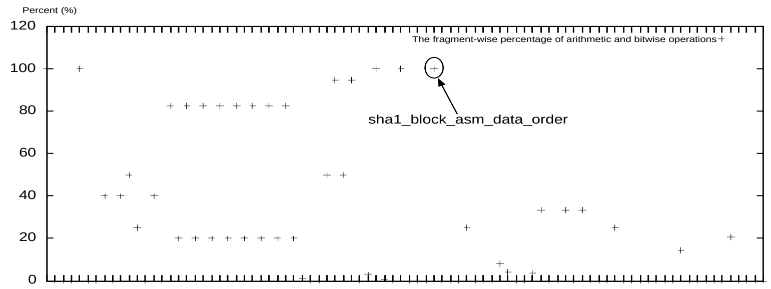


Fig. 5. Phase Profiler (Step II): Calculating the fragment-wise percentage of arithmetic and bitwise operations within the search range

structions and more than 90% of bitwise instructions actually occurred in the message decryption phase.

3.4 Data Lifetime Analyzer

After determining the message decryption phase and the normal protocol processing phase, our next step is to locate the memory buffers that contain the decrypted message. The basic idea is to identify the buffers (data) passed from the message decryption phase to the normal protocol processing phase. Specifically, the buffers must be written in the former phase and read in the latter phase. To identify such buffers, we analyze the *lifetime* of memory buffers.

Before describing our algorithm, we first define the liveness of a memory buffer. Note that a buffer is a contiguous memory block, and we only care about *tainted* buffers. When an application starts, we mark all buffers pre-allocated for global variables as *live*. Then, in the message decryption phase, after a buffer is allocated in the heap or the stack, we mark it as live; after a live buffer is deallocated from the heap or the stack (i.e., when a stack frame is popped), we clear the "live" mark associated with the buffer and it becomes invalid. After the application enters the normal protocol processing phase, we handle the liveness of memory buffers differently. Specifically, after a buffer is deallocated or accessed (either read or write operations), it becomes invalid for the following reasons: A deallocated buffer will become invalidated right after the deallocation operation.

```
 41748f8  97: GET / HTTP/1.0..User-Agent: Wget/1.10.2..Accept: */*..Host: localhost..Connection: Keep-Alive....
 417e0b5 133: .....GET / HTTP/1.0..User-Agent: Wget/1.10.2..Accept: */*..Host: localhost..Connection: Keep-Alive
              ......m...1.q..D.%....u............
 4197bc0  20: ......d...6T../.b.f.
 4197c58  20: .@].l...Y...7T...!.k
 4197cf0  20: O.#..31.r.^......T.
 4197d0c  20: .".Rxvj.Ns.1...‘".~W
 4197d88  20: ......d...6T../.b.f.
 4197e20  20: .@].l...Y...7T...!.k
 4197eb8  20: .m...1..D..q..%..u..
 4197ee0  52: TEGaH / /PTT.0.1esU.gA-r:tneegW .1/t2.01cA..tpec/* :
bee82cfc  20: ..k..w....b......J.K
bee82de0  16: ...V.31..|....$.
bee832f0  20: ......\...}.....m...
bee83348  56: ....TEGaH / /PTT.0.1esU.gA-r:tneegW .1/t2.01cA..tpec/* :
bee833cc  20: m........CG.q..AX.G.
bee83408  20: .....\...}........m
bee834d0  20: 1S....VY....-.M....T
bee835dc  20: ..m...1.q..D.%....u.
```

(a) The *write set* in the message decryption phase

```
41748f8 97: GET / HTTP/1.0..User-Agent: Wget/1.10.2..Accept: */*..Host: localhost..Connection: Keep-Alive....
4197f50 97: GET / HTTP/1.0..User-Agent: Wget/1.10.2..Accept: */*..Host: localhost..Connection: Keep-Alive....
```

(b) The *read set* in the normal protocol processing phase

Fig. 6. Data Lifetime Analyzer: Obtaining the *write set* and *read set*

If a buffer is being written to, it will be marked invalid as the buffer's content is not from the message decryption phase any more. For read operations, we only need to care about the first read operation and will not consider further reads.

Based on the liveness definition, we identify the memory buffers that contain the decrypted message in three steps. First, we search for all the buffers that were written to in the message decryption phase and are still live when the application enters the normal protocol processing phase. We refer to this set of buffers as the *write set*. Second, we search for all the buffers that are live when they are being first read from in the normal protocol processing phase. We refer to this set of buffers as the *read set*. Finally, we identify the buffers in the intersection of the two sets as those that contain the decrypted message.

If the intersection of the write and read sets has only a single buffer, this buffer will be used as the decrypted input message for the format analysis. If multiple buffers are found in the intersection, we first sort them based on the temporal order of the first read operations on them. Then, we treat the sorted buffers as a virtual single buffer that contains the whole decrypted message.

In our running example, the write and read sets we identified are shown in Figure 6. After intersecting the two sets, we find only one common buffer that starts at $0x041748f8$ with the following content: $GET/HTTP/1.0..$
$UserAgent : Wget/1.10.2..Accept : */*..Host : localhost..Connection : KeepAlive....$
Based on the knowledge of the HTTP protocol, we know that it is *the* buffer that contains the decrypted message. After identifying the decrypted message buffer, we then apply the AutoFormat tool as the format analyzer and the result is shown in Figure 7.

4 Implementation and Evaluation

We have implemented a ReFormat prototype based on the latest release of Valgrind (version 3.2.3). Our execution monitor is built on top of some features supported in Valgrind such as instruction translation, memory marking, and

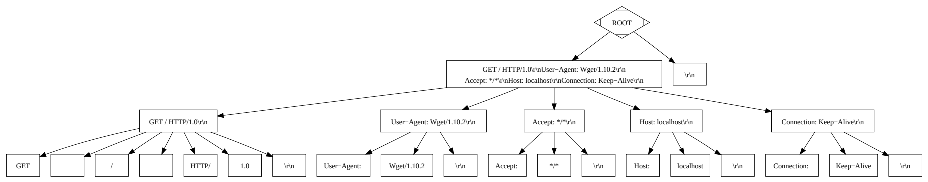

ROOT
GET / HTTP/1.0\r\nUser−Agent: Wget/1.10.2\r\n Accept: */*\r\nHost: localhost\r\nConnection: Keep−Alive\r\n
\r\n
GET / HTTP/1.0\r\n
User−Agent: Wget/1.10.2\r\n
Accept: */*\r\n
Host: localhost\r\n
Connection: Keep−Alive\r\n
GET
/
HTTP/
1.0
\r\n
User−Agent:
Wget/1.10.2
\r\n
Accept:
/
\r\n
Host:
localhost
\r\n
Connection:
Keep−Alive
\r\n

Fig. 7. Format Analyzer: Revealing the HTTPS request message format

propagation capabilities. Our phase profiler and data lifetime analyzer are standalone python programs. Our format analyzer uses the AutoFormat tool [8]. We note that our system is not tightly coupled with Valgrind and AutoFormat and can be implemented using other binary instrumentation tools such as Pin and QEMU as well as other reverse engineering tools such as Polyglot [5], the system [9] by Wondracek *et al.*, and Tupni [7]. Excluding the AutoFormat code, our ReFormat prototype has 4626 lines of C and 1392 lines of Python.

In our evaluation, we performed two sets of experiments. The first set of experiments involves input messages from three known protocols, HTTPS, IRC, and MIME. The second set of experiments was conducted on an unknown protocol used by agobot [14], a real-world malware. Table 2 shows the list of protocols we tested and the programs we used. These programs are obtained either directly from the standard OS distribution or by compiling the source code with the default configuration. For each experiment, Table 2 lists the decrypted (plain-text) message size, the total number of tainted buffers, and the size of write set, read set and their intersection. Notice the number of tainted buffers is larger than the total size of both the write set and the read set. This is because new tainted buffers generated in normal protocol processing phase are not included in the write set or the read set, but are counted in the tainted buffer set. In each experiment, we ran our prototype to obtain the decrypted message and its format. The format accuracy is dependent on two factors: the accuracy of the decrypted message and the effectiveness of the format analyzer tool. Since we uses AutoFormat in our prototype and its effectiveness was evaluated in [8], we focus on the accuracy of the decrypted message in our experiments. By accuracy, we measure whether the buffers we found after the data lifetime analysis contains the *complete* decrypted input message and *nothing else*. For completeness, we show the formats reverse engineered by AutoFormat. In all our experiments, ReFormat accurately identified the decrypted message. In the rest of this section, we describe our experimental results for IRC and agobot in detail. Due to space constraint, we omit the detailed results for HTTPS and MIME. Interested readers are referred to our technical report[15].

4.1 Experiments with Known Protocols

IRC: In this experiment, we evaluated ReFormat with a secure IRC server. Specifically, we monitored the execution of the latest *ircd-hybrid* server[16] (version: 7.2.3), and ran *xchat*, an IRC client, from another physical machine to

Table 2. Summary of experiments

Protocol	Application	Msg Type	Size(B)	*Tainted set*	*write set*	*read set*	*write set* ∩ *read set*
HTTPS	SHTTPD (version: 1.38)	Linux Wget	97	40	18	2	1
		Linux Firefox	362	38	5	4	1
		Windows IE	283	83	5	3	1
		Google Chrome	431	112	6	3	1
	Apache (version: 2.0.63)	Linux Wget	102	57	13	9	1
		Linux Firefox	475	51	6	18	1
		Windows IE	286	91	19	11	1
		Google Chrome	431	96	6	13	1
IRC	IRCD-Hybrid (version: 7.2.3)	JOIN message	16	59	8	2	1
		MODE message	16	42	8	2	1
		WHO message	15	53	7	2	1
MIME	Metamail (version: 2.7)	BASE64-encoded email message	1141	31	20	3	1
Unknown	Agobot (version: 3-0.2.1)	bot.status message	61	172	9	33	1
		bot.execute message	68	144	10	36	1
		bot.sysinfo message	62	174	9	33	1

establish a secure connection. After the connection is made, we executed the IRC command */join #channel1* to log into a specific channel. This command triggered three IRC messages to be sent: **JOIN #channel1\r\n**, **MODE #channel1\r\n**, and **WHO #channel1\r\n**. Instead of showing our analysis on each message separately, we combine the traces and show the phase profile analysis results collectively in Figure 8. For each message, the cumulative percentage of arithmetic and bitwise instructions reaches the highest value when the function *sh*1*_block_asm_data_order* is executed and drops to the lowest value when the function *ssl*3*_read_n* is executed. For each message, we show at the bottom the decrypted message identified by ReFormat. It is clear that ReFormat identified all three decrypted messages accurately.

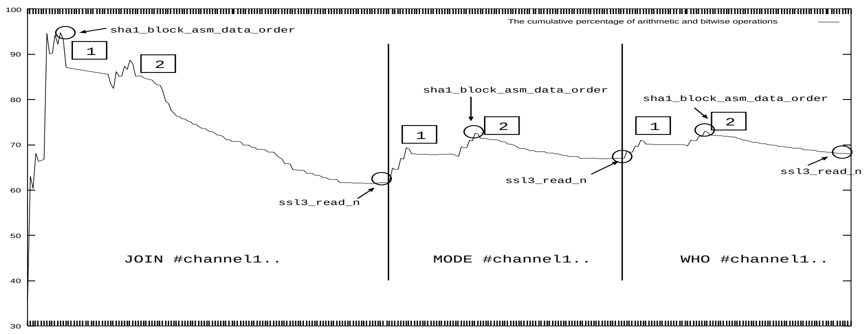


Fig. 8. The cumulative percentage of arithmetic and bitwise operations in the collected *Ircd-Hybrid*-based execution trace

Interestingly, for each message shown in Figure 8, there are two peaks (marked as 1, and 2 in the figure) in the cumulative percentage of arithmetic and bitwise operations. Further investigation reveals that an encrypted message such as the one corresponding to **WHO #channel1\r\n** is encapsulated into two 32-byte SSL record layers and each SSL record layer will be independently decrypted before being combined together for normal protocol processing. In other words, for each encrypted message, it will go through two rounds of decryption, hence leading to two peak values in the corresponding portion of the curve in Figure 8.

4.2 Experiments with Unknown Protocols

We now present our second set of experiments to show that ReFormat is able to uncover the format of encrypted protocol messages used by a real world bot program. Specifically, we monitored the execution of a bot software called *agobot* [14] and this particular bot contains its own (proprietary) SSL implementation. When the bot runs, it persistently attempts to connect to a pre-specified IRC server and log into a hard-coded channel. To confine potential damage, we performed a controlled experiment where the bot's connection request was redirected to a local IRC server under our control. In addition, we used the *xchat* program to connect to the IRC server, join the secure channel, and issue commands to the bot. In the meantime, we collected the execution trace of the agobot. We learned about the channel name and control commands from our own manual analysis and other reverse engineering efforts [14]. We want to point out that such manual efforts are simply for our controlled experiments and ReFormat is used to demonstrate the capability in automatically uncovering the command format.

By analyzing the execution trace, we found that the agobot received 15 messages in total: two messages for the SSL handshake, seven messages for establishing the secure connection to the IRC server and logging into a specified IRC channel, and six messages for the commands received from our own botmaster. In our experiment, we focused on a single command message: *.bot.execute /bin/ps.*

Figure 9 shows the cumulative percentage of arithmetic and bitwise instructions. According to the cumulative percentage, we identified the functions *sha1_block_asm_data_order* and *CBot::HandleCommand* as the maximum and minimum functions. Further, based on the fragment-wise percentage of arithmetic and bit instructions, we identified that *sha1_block_asm_data_order* is the transition function. The write set and the read set are shown in Figure 10(a) and 10(b), respectively. The intersection of the two sets has only one buffer at the address 0x04285b8d. We find its content is the same as the command issued by our *xchat* program, We then applied AutoFormat to uncover the format of this decrypted message and the result is shown in Figure 11.

5 Related Work

In this section, we describe the related work and compare it with ReFormat. Note that the execution monitor in ReFormat leverages generic techniques of dynamic taint analysis, which has been widely investigated. In this section, we omit detailed discussion on this area. Interested readers are referred to a number of recent efforts on taint analysis [11, 12].

As mentioned earlier, automatic protocol reverse engineering has recently received significant attention due to its importance to many security applications.

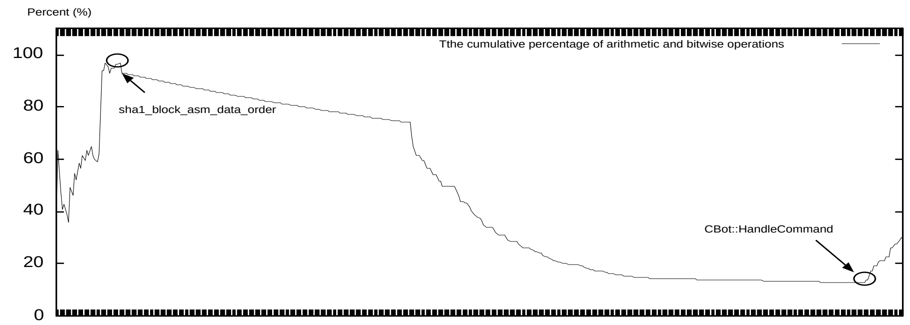


Fig. 9. The cumulative percentage of arithmetic and bitwise operations in the collected trace when *agobot* handles the *.bot.execute /bin/ps* command

```
4285b88  5: .  . . ‘
4285b8d 96: :BotMstr!~BotMstr@172.16.237.1 PRIVMSG #Agonet :.bot.execute /bin/ps...8.C@...M.3....2...2..B...
4b6edcc 20: _.[........_..._.[.
429aeb0 16: ........_..._.[.
42c50d8 60: :F..MtoB!rtstoB~rtsM271@.61..732RP 1SMVIA# Genog.: t.tobcexe
4b6ed24 60: :F..MtoB!rtstoB~rtsM271@.61..732RP 1SMVIA# Genog.: t.tobcexe
42c50d4 16: ....4...f...;.&9
4b6ed20 16: ....4...f...;.&9
42c50b8 20: C.8....@.3.M2...2...
4b6ee90 20: .8.C@...M.3....2...2
```

(a) The *write set* in the message decryption phase

```
4285b8d 68: :BotMstr!~BotMstr@172.16.237.1 PRIVMSG #Agonet :.bot.execute /bin/ps
42c51c8 32: :BotMstr!~BotMstr@172.16.237.1
... ...
42c6228  9: BotMstr!~
42c6230 21: ~BotMstr@172.16.237.1
42c6440 59: ~BotMstr@172.16.237.1 PRIVMSG #Agonet :.bot.execute /bin/ps
... ...
42c677d 15: :.bot.execute /
42c68c8 12: 172.16.237.1
42c6908 12: 172.16.237.1
42c6948 32: :BotMstr!~BotMstr@172.16.237.1 P
... ...
42c6fc8 14: .bot.execute /
42c70b8 14: .bot.execute /
4b6afca 68: :BotMstr!~BotMstr@172.16.237.1 PRIVMSG #Agonet :.bot.execute /bin/ps
```

(b) The *read set* in the normal protocol processing phase

Fig. 10. Locating the decrypted message for the *.bot.execute* command

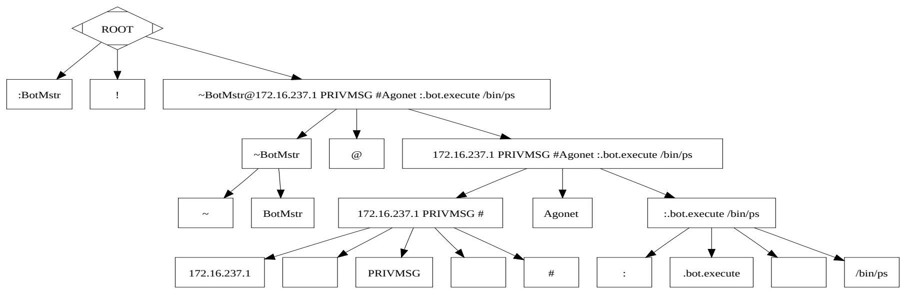


Fig. 11. Revealing the *.bot.execute* command message format

The Protocol Informatics (PI) project [4] and Discoverer [6] aim at extracting protocol format from collected network traces. They have the advantage of conveniently collecting network traces when a parsing program is unavailable. However, they become less effective in the face of encrypted network traffic. Unlike the PI and Discoverer projects, several systems such as Polyglot [5], the system in [9], AutoFormat [8], and Tupni [7] share the key insight that how a program parses and processes a message reveals rich information about the message format. Based on this insight, they reverse engineer input message formats by using dynamic data flow analysis to understand how a program consumes an input message. Prospex [10] makes a step further to uncover protocol specification. In comparison, these systems are mainly designed to work with plain-text input messages. ReFormat complements these systems by providing an effective scheme to discern the protocol processing phase from the message decryption phase and then pinpoint the run-time memory buffers that contain the decrypted message. And naturally, the above program-based systems can be integrated in ReFormat to reverse engineer the format of the decrypted message.

In addition, there has been related work that studies reverse engineering for specific applications such as application-level replay. For example, RolePlayer [17] and ScriptGen [18] replay a recorded network protocol session with another entity by identifying and updating certain input fields that are embedded in the recorded session. None of these systems can handle encrypted application-level communications. Protocol analyzers such as Wireshark have the capability of properly formatting a protocol message, but they require prior knowledge about those protocols and are of less use when analyzing unknown or encrypted protocols.

ReFormat relies on another general technique, i.e., data lifetime analysis, to locate the decrypted memory buffers. Along with dynamic taint analysis, this technique has been proposed in another different problem context [11, 19] that aims to detect potential leakage of sensitive data such as passwords and social security numbers in the memory. ReFormat differs from them by focusing on the identification of the run-time memory buffers of the decrypted message.

6 Limitations and Future Work

In this section, we discuss the limitations in ReFormat and suggest possible improvements for future work.

First, ReFormat relies on the observation that the instruction distribution for message decryption is significantly different from normal protocol processing. While this observation holds true for many applications as we have shown in previous sections, it may not be the case when the normal protocol processing would be essentially doing some intensive decryption-like operations. In other words, when the processing of a message content involves significant arithmetic and bitwise operations, our system may not work properly. One possible way to solve these problems is to uncover other characteristics of the message decryption phase and use such characteristics to differentiate it from the normal protocol processing phase.

Second, ReFormat is designed to handle benign programs and malware that do not intentionally obfuscate their executions to thwart program analysis. In other words, the analysis of ReFormat can be potentially evaded if a program deliberately introduces redundant instructions to manipulate the distribution, e.g., embedding unnecessary arithmetic or bitwise operations in normal protocol processing or injecting unnecessary non-arithmetic or non-bitwise instructions into message decryption. How to make ReFormat applicable to obfuscated programs still remains a technical challenge.

Third, ReFormat assumes an application first decrypts an encrypted message and then processes the decrypted message. If an application does not follow this assumption, e.g., it decrypts *part* of the message and processes it before decrypting and processing the rest, ReFormat may not identify the whole decrypted message correctly. To handle such applications, we would need to divide an execution trace into multiple decryption and processing phases. We leave this to future work.

Finally, ReFormat analyzes one input message at a time and does not correlate multiple messages in the same protocol session. Extending ReFormat to further reconstruct the entire protocol state machine is part of our future work.

7 Conclusion

We have presented ReFormat, a system that enables existing automatic protocol reverse engineering tools to handle encrypted messages. ReFormat is based on the insight that the instructions used for message decryption is substantially different from those for normal protocol processing. By analyzing the percentage of arithmetic and bitwise instructions, ReFormat can discern the message decryption phase and the normal protocol phase. Furthermore, with the insight that the decrypted message is generated in the message decryption phase and handled in the normal protocol processing phase, ReFormat can analyze the data lifetime of run-time buffers to accurately pinpoint the memory buffers that contain the decrypted message. We have implemented a prototype of ReFormat and evaluated it with a variety of protocol messages from real-world (known or unknown) protocols. Our experimental results show that ReFormat achieves high accuracy in locating the decrypted message buffers and extracting the related message structure.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This work was supported in part by US National Science Foundation (NSF) under Grants 0852131 and 0855297. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

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

2. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In: Proceedings of ACM SIGCOMM '04. (2004) 193–204
3. Cui, W., Peinado, M., Wang, H.J., Locasto, M.: Shieldgen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In: Proceedings of 2007 IEEE Symposium on Security and Privacy, Oakland, CA (May 2007)
4. : The Protocol Informatics Project. http://www.baselineresearch.net/PI/
5. Caballero, J., Song, D.: 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)
6. Cui, W., Kannan, J., Wang, H.J.: Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In: Proceedings of the 16th USENIX Security Symposium (Security'07), Boston, MA (August 2007)
7. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: Automatic Reverse Engineering of Input Formats. Proceedings of the 15th ACM Conferences on Computer and Communication Security (CCS'08) (October 2008)
8. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08) (February 2008)
9. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic Network Protocol Analysis. Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08) (February 2008)
10. Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: Protocol Specification Extraction. In: Proceedings of 2009 IEEE Symposium on Security and Privacy, Oakland, CA (May 2009)
11. Chow, J., Pfaff, B., Christopher, K., Rosenblum, M.: Understanding Data Lifetime via Whole-System Simulation. In: Proceedings of the 13th USENIX Security Symposium, San Diego, CA (2004)
12. Newsome, J., Song, D.: 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)
13. : SHTTP: An Embeddable Web Server. http://shttpd.sourceforge.net/
14. : Know your Enemy: Tracking Botnets - Bot-Commands. http://www.honeynet.org/papers/bots/botnet-commands.html
15. Wang, Z., Jiang, X., Cui, W., Wang, X.: Reformat: Automatic Reverse Engineering of Encrypted Messages. (Department of Computer Science Technical Report, North Carolina State University, TR-2008-26) (2008)
16. : Ircd-hybrid – High Performance Internet Relay Chat. http://ircd-hybrid.com/
17. Cui, W., Paxson, V., Weaver, N., Katz, R.H.: 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)
18. Leita, C., Mermoud, K., Dacier, M.: ScriptGen: An Automated Script Generation Tool for Honeyd. In: Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05), Washington, DC, USA (2005) 203–214
19. Chow, J., Pfaff, B., Garfinkel, T., Rosenblum, M.: Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In: Proceedings of the 14th USENIX Security Symposium, Baltimore, Maryland (2005)