ReFormat: Automatic Reverse Engineering of Encrypted Messages

Zhi Wang, Xuxian Jiang
North Carolina State University

Weidong Cui
Microsoft Research

Xinyuan Wang
George Mason University

Abstract

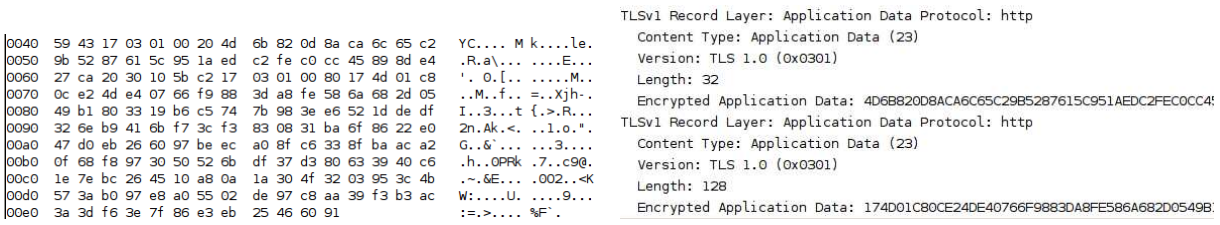
Automatic protocol reverse engineering has recently received significant attentions due to its importance to many security applications. However, previous methods are all limited in only analyzing plain-text *communications wherein the exchanged messages are not encrypted (e.g., without SSL encryption). In this paper, we propose ReFormat, a system that aims at deriving the message format even when the message is encrypted. When an encrypted input message is received, it will typically go through two main processing phases: message decryption and normal protocol processing. Based on the observation that the instructions used for message decryption is significantly different from those used for normal protocol processing, we can identify and separate these two phases by profiling the instructions executed. Further, since the plain-text messages used in normal protocol processing are generated from the message decryption phase, we can then analyze the data lifetime of run-time buffers generated from the message decryption phase to accurately pinpoint the memory locations that contain the decrypted (plain-text) message. Once it is determined, previous research efforts in analyzing plain-text protocol messages can be naturally applied to reveal the protocol message format. We have developed a prototype of ReFormat and evaluated it with four real-world protocols, HTTPS, IRC, MIME, and an unknown one used by a malware. Our experiments show that ReFormat can accurately identify decrypted message buffers and then reveal the associated message structure.*

1 Introduction

With great potentials to many security applications, protocol reverse engineering has recently received significant attentions. For example, network-based firewalls or filters [4, 23, 26] require the knowledge of protocol specifications to understand the context of a particular network communication session. Similarly, fuzz testing [13, 15, 27] 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 [3, 6, 10, 12, 18, 28] have been developed to allow for automatic protocol reverse engineering. The Protocol Informatics [3] project and Discoverer [10] 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 [6], the work [28] by Wondracek *et al.*, AutoFormat [18], and Tupni [12] take a program-based approach to find out the message format. While different in various aspects, these program-based systems share a similar insight that 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



(a) An encrypted web request message captured by TCPDUMP (b) The protocol format identified by Wireshark

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

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 key insight that *the instructions used for decrypting an encrypted message is significantly different from those used for processing a normal unencrypted protocol message*. As a result, 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 perform the *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 [6, 12, 18, 28] 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 [5]. 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

```
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: */

    ...
}
```

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

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 big 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 benefit to handle not only plain-text messages but also encrypted ones.

To fill the gap, there are several challenges: First, the memory buffers that contain the decrypted message are not known 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. In addition, these buffers can come from various sources, including global variables, the heap, or the stack. The obvious challenge is how to systematically identify the buffers that contain the decrypted message ("a needle") among all these memory buffers ("in the haystack"). 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

The goal of our system is to, given an encrypted message and an application that can decrypt the message and then process it, 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

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%

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

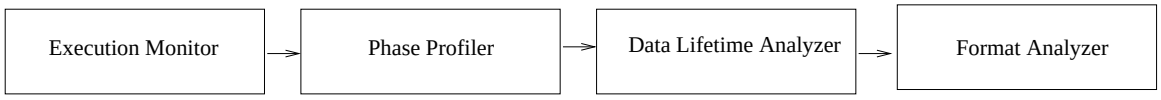


Figure 3. ReFormat System Architecture

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.

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 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 [6, 12, 28, 18], we focus on the first three steps in this paper. In our prototype, we use AutoFormat [18] as our format analyzer but other systems [6, 12, 28] should be equally applicable for the same purpose.

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 [2] processes an encrypted HTTP request issued by *wget* [30], 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. However, if the binary is 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. First, we use the cumulative percentage of arithmetic and bitwise instructions to narrow down the search range where the transition point is located. Second, we use the function-wise percentage of arithmetic and bitwise instructions to identify the last function executed in the message decryption phase. We refer to this function as the *transition function*. Therefore, the last instruction executed in this function will be the transition point between the two phases. In other words, we use the notion of transition function to indicate that no other function that executes after it will be performing any *actual* message decryption. Next, we describe these two steps in detail.

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 an output plain-text 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*. Next we narrow our search to the instructions executed between them.

To better illustrate the second step, we define a new term, *leaf function*. A leaf function contains contiguous instructions that belong to the same function. For instance, if a parent function A calls a child function B and there is no function called in B, we will have three leaf functions, 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 leaf function. For the maximum and minimum instructions identified previously, we refer to their leaf functions as the *maximum leaf*

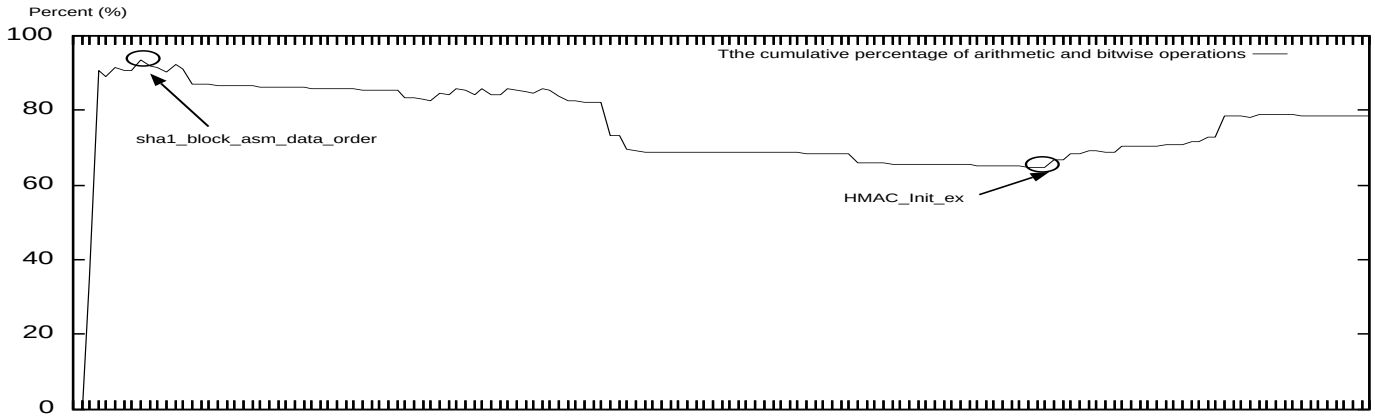


Figure 4. Phase Profiler (Step I): Calculating the cumulative percentage of arithmetic and bitwise operations in the collected shttpd-based execution trace

function and the *minimum leaf function*.

After identifying the maximum and minimum instructions based on the cumulative percentage, we compute the *function-wise* percentage of arithmetic and bitwise instructions for each leaf function between them. Here we use the function-wise percentage instead of the cumulative percentage because the leaf functions for *actual* message decryption are likely to have high function-wise percentage. Therefore we identify the last leaf function whose percentage is above a given threshold as the transition function. The last instruction executed in this function will be used as the transition point. In our prototype, we set the threshold to be 50% based on the percentages of arithmetic and bitwise operations shown in Table 1. 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 leaf function to compute the function-wise percentage of arithmetic and bitwise instructions. However, since we do not encounter such cases in our evaluation (Section 4), we plan to research the selection of the sliding window size when such a need arises.

In our running example, the cumulative percentage of arithmetic and bitwise instructions is shown in Figure 4. The X-axis is the leaf functions 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 *sha1_block_asm_data_order*. After that, the cumulative percentage keeps decreasing until it reaches the lowest value at an instruction inside the function *HMAC_Init_ex*. In Figure 5 we show the function-wise percentage of arithmetic and bitwise instructions for each leaf function 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 instructions 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)

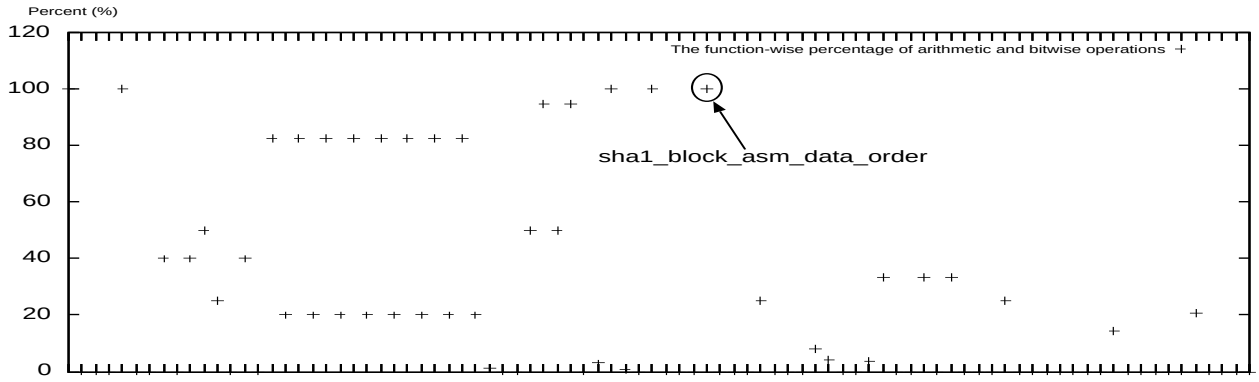


Figure 5. Phase Profiler (Step II): Calculating the function-wise percentage of arithmetic and bitwise operations within the search range

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 the buffer becomes "dead". 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 dead for the following reasons: A deallocated buffer will become invalidated right after the deallocation operation. If a buffer is being written, it will be marked dead 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 mark a buffer dead after it.

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 3.4. 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.

```
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

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

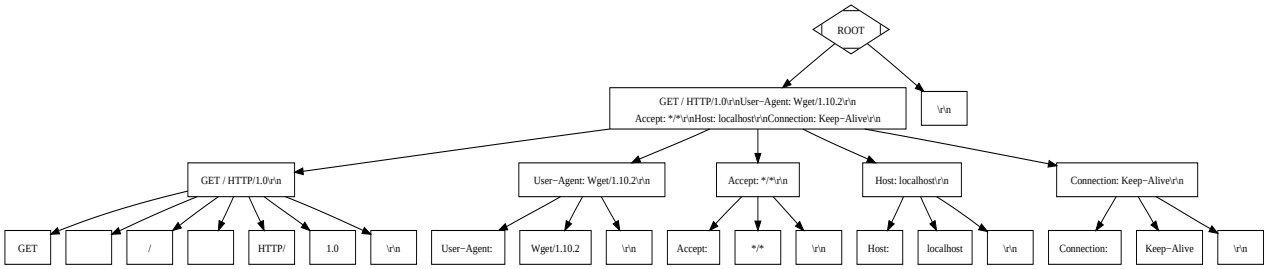


Figure 7. Format Analyzer: Revealing the HTTPS request message format

4 Implementation and Evaluation

We have implemented a ReFormat prototype based on the latest release of Valgrind [20] (version 3.2.3). Our execution monitor is built on top of some features supported in Valgrind such as instruction translation, memory marking, and propagation capabilities. Our phase profiler and data lifetime analyzer are standalone python programs. Our format analyzer uses the AutoFormat tool [18]. 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 [19] and QEMU [1] as well as other reverse engineering tools such as Polyglot [6], the system [28] by Wondracek *et al.*, and Tupni [12]. Excluding the AuotFormat 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 [38], 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. 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

Protocol	Application	Request Msg Type	Msg Size(B)	*write set*	*read set*	$write\ set \cap read\ set$
HTTPS	SHTTPD (version: 1.38)	Linux Wget	97	18	2	1
		Linux Firefox	362	5	4	1
		Windows IE	283	5	3	1
		Google Chrome	431	6	3	1
	Apache (version: 2.0.63)	Linux Wget	102	13	9	1
		Linux Firefox	475	6	18	1
		Windows IE	286	19	11	1
		Google Chrome	431	6	13	1
IRC	IRCD-Hybrid (version: 7.2.3)	JOIN message	16	8	2	1
		MODE message	16	8	2	1
		WHO message	15	7	2	1
MIME	Metamail (version: 2.7)	BASE64-encoded email message	1141	20	3	1
Unknown	Agobot (version: 3-0.2.1)	bot.status message	61	9	33	1
		bot.execute message	68	10	36	1
		bot.sysinfo message	62	9	33	1

Table 2. Summary of experiments

message and the effectiveness of the format analyzer tool. Since we uses AutoFormat in our prototype and its effectiveness was evaluated in [18], 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 in detail.

4.1 Experiments with Known Protocols

HTTPS: We have experimented with two different HTTPS servers, SHTTPD [2] (version 1.38) and Apache [31] (version 2.0.36). In Section 3, we have used the SHTTPD web server as our running example and presented our experimental results. In this section, we focus on the Apache results. In our Apache experiments, we monitored the execution of an Apache web server and collected its execution trace when it processed an incoming HTTPS request. To generate different HTTPS requests, we have used four different clients: *GNU wget*[30], *Mozilla Firefox*[32], *Microsoft IE*[33], and *Google Chrome*[34]. In the following, we describe the results of the experiment with Google Chrome. Other experimental results are summarized in Table 2.

In Figure 8, we show the cumulative percentage of arithmetic and bitwise instructions in the collected trace. The cumulative percentage reaches its highest value when the function $md5\_block\_asm\_host\_order$ is executed and drops to its lowest value when the function $strncasecmp$ is executed. After computing the function-wise percentage of arithmetic and bit instructions for each leaf function executed between these two functions, we found that $md5\_block\_asm\_host\_order$ is the only function that has a high percentage ($> 90\%$) and all other functions have their percentages less than 10%. Therefore, the transition function is $md5\_block\_asm\_host\_order$. Note that this transition function is different from the transition function $sha1\_block\_asm\_data\_order$ in our running example (see Figure 4). Our manual investigation shows that it is because the former case uses the RC4 encryption algorithm with the MD5 checksum while the latter uses the AES encryption algorithm with the SHA1 checksum.

Our data lifetime analysis found 6 memory buffers in the write set (shown in Figure 9(a)) and 13 memory buffers in the read set (shown in Figure 9(b)). The intersection of the two sets has a single buffer at the address

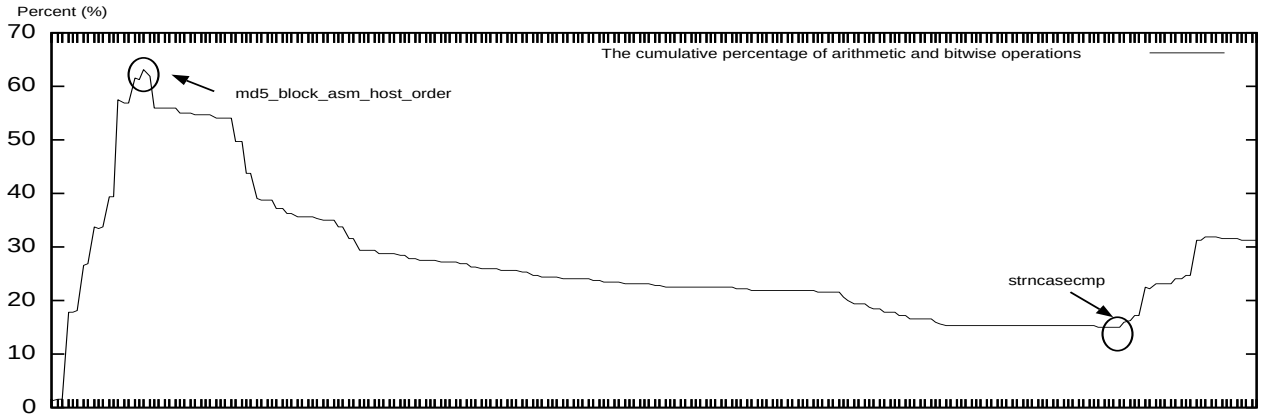


Figure 8. The cumulative percentage of arithmetic and bitwise operations in the collected Google Chrome-related execution trace

```
453ede5 447: GET / HTTP/1.1..User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13
             (KHTML, like Gecko) Chrome/0.2.149.29 Safari/525.13..Accept-Language: en-US,en..Accept: text/xml,
             application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5..
             Cache-Control: max-age=0..Accept-Charset: ISO-8859-1,*,utf-8..Accept-Encoding: gzip,deflate,bzip2
             ..Host: 172.16.237.128..Connection: Keep-Alive....
454bf80 452: .....$l....q.G.nUt2.0._.......%...D..bgP.k..$.T.j..(..7/...._F.....sI..GJ...;........ .eOc.~....n}
             .~..............5...Vv......v.hk.....&%(lv...}6K...U.A.[...l.'...3...@.......a....j......m....w...
             P;.~.....p.6.@...%.QSq....Y..)T...3.W/....I..F....Dn......26@..c<...G.l.u..X...>.#^/D.......FJ...*F
             ._.O.^!.*.l...a_..M.......uF....l'..wf.".RV;.FF...9..S..t.....xr......i..8)....>.Ke9...[x.i%.L....O
             ...4B.."..B.G......U.W<......g....H........!#3.%K.......
 455a888  16: #..>.8._...... 9
 455a8a0  16: X__TJ(F.d...b.Y.
beb82268  16: X__TJ(F.d...b.Y.
beb8242c  16: #..>.8._...... 9
```

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

```
 45345c8 431: GET / HTTP/1.1..User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13
              (KHTML, like Gecko) Chrome/0.2.149.29 Safari/525.13..Accept-Language: en-US,en..Accept: text/xml,
              application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5..
              Cache-Control: max-age=0..Accept-Charset: ISO-8859-1,*,utf-8..Accept-Encoding: gzip,deflate,bzip2
              ..Host: 172.16.237.128..Connection: Keep-Alive....
 4538330  16: GET / HTTP/1.1..
 45383a0   8: HTTP/1.1
 45383b0 132: User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13 (KHTML, like Gecko)
              Chrome/0.2.149.29 Safari/525.13..
 4538438  27: Accept-Language: en-US,en..
 4538488 109: Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,
              */*;q=0.5..
 45384f8  26: Cache-Control: max-age=0..
 4538548  36: Accept-Charset: ISO-8859-1,*,utf-8..
 4538598  37: Accept-Encoding: gzip,deflate,bzip2..
 45385e8  22: Host: 172.16.237.128..
 4538638  24: Connection: Keep-Alive..
 4538718  14: 172.16.237.128
453ede5 431: GET / HTTP/1.1..User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.13
              (KHTML, like Gecko) Chrome/0.2.149.29 Safari/525.13..Accept-Language: en-US,en..Accept: text/xml,
              application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5..
              Cache-Control: max-age=0..Accept-Charset: ISO-8859-1,*,utf-8..Accept-Encoding: gzip,deflate,bzip2
              ..Host: 172.16.237.128..Connection: Keep-Alive....
```

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

Figure 9. Locating the decrypted message

$0x0453ede5$ with the size of 431 bytes. We highlight this buffer in both figures and find that it is indeed the memory buffer with the decrypted message. Note that six bytes at the end of the buffer in the write set is *not* included because they are not in the read set. We ran AutoFormat on the decrypted message and obtained the message structure shown in Figure 10.

IRC: In this experiment, we evaluated ReFormat with a secure IRC server. Specifically, we monitored the execution of the latest *ircd-hybrid* server[35] (version: 7.2.3), and ran *xchat*[36], an IRC client, from another physical machine to establish a secure connection. After the connection is made, we executed the IRC command

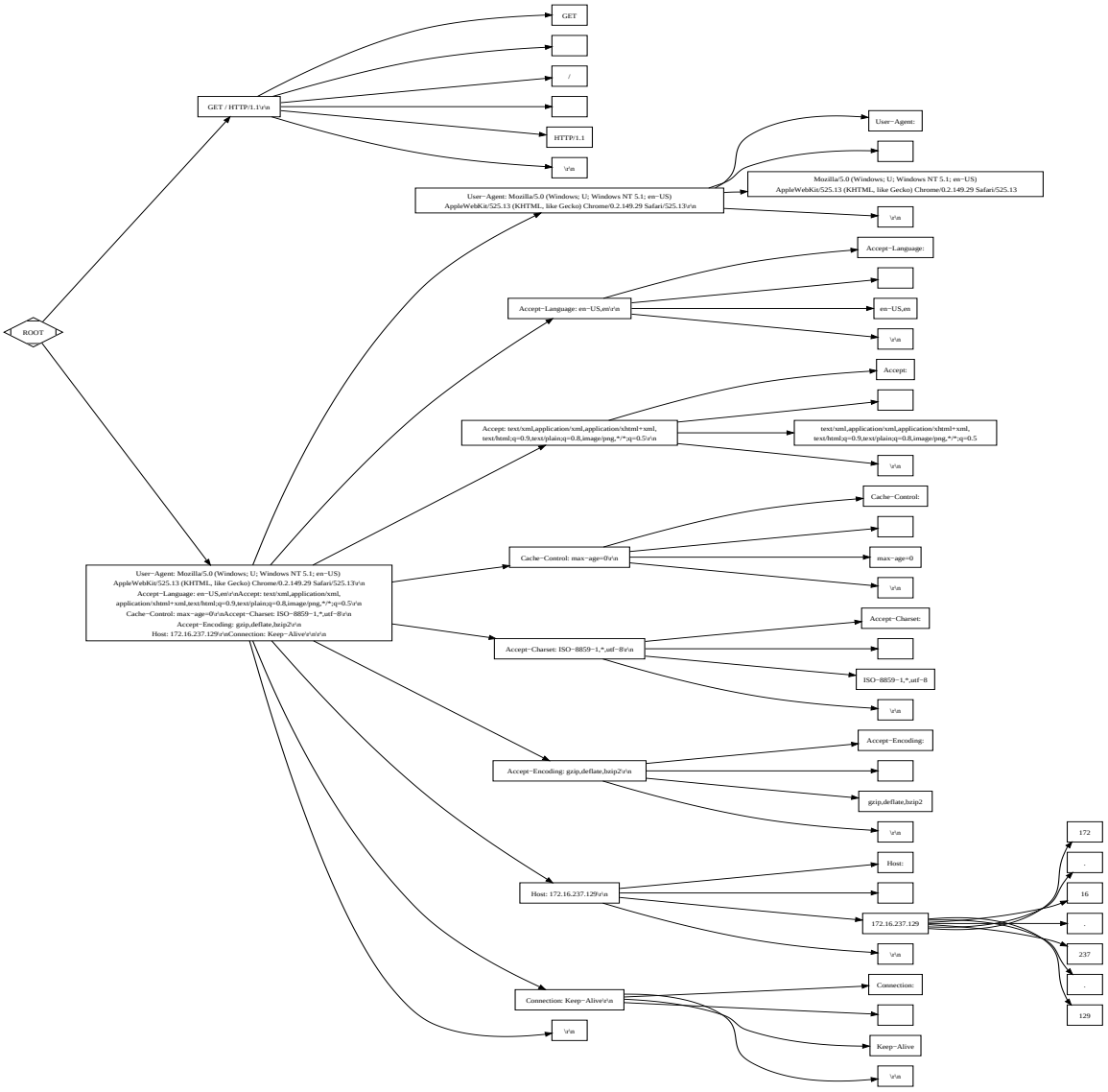


Figure 10. Revealing the Google Chrome-based HTTPS request message format

/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 11. For each message, the cumulative percentage of arithmetic and bitwise instructions reaches the highest value when the function $sh1\_block\_asm\_data\_order$ is executed and drops to the lowest value when the function $ssl3\_read\_n$ 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.

Interestingly, for each message shown in Figure 11, 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 first 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 11.

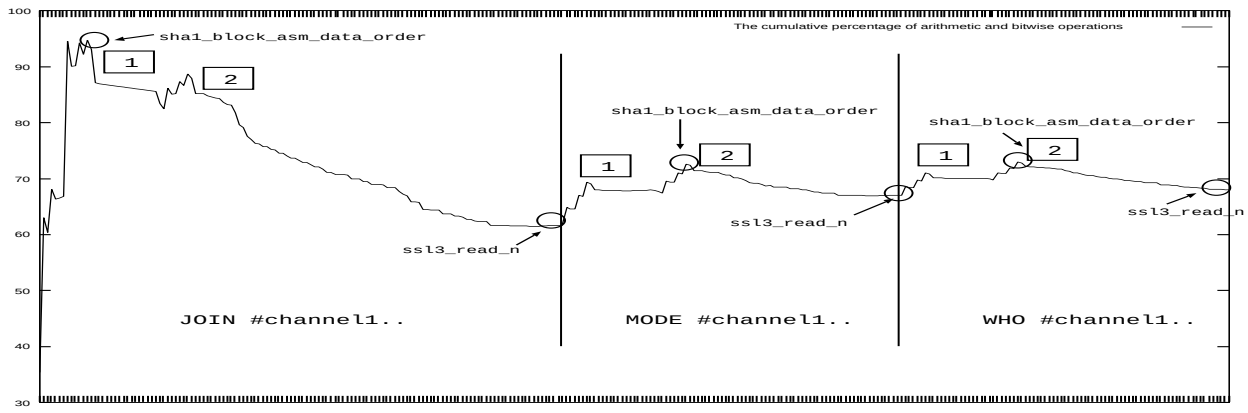


Figure 11. The cumulative percentage of arithmetic and bitwise operations in the collected *Ircd-Hybrid*-based execution trace

MIME/BASE64: In this experiment, we tested our system on an email client that is capable of reading a *BASE64*-encoded message. *BASE64* is a binary-to-text encoding scheme that works as follows: Given a message text as a string of bytes, each three 8-bit bytes are splited into four 6-bit snippets and each is then used as an offset to index into an array of 64 printable characters. To decode the message, one can simply reverse this process. Note that *BASE64* is widely used in the MIME support that extends the format of e-mail to support a variety of objects (e.g., pictures) other than the text in ASCII character sets.

In our experiment, we used a Linux-based email client called *metamail*[37] (version 2.7) and monitored its execution when it opened a *BASE64*-encoded email message. From the trace, we found that the *metamail* program wrote the decoded message into a temporary file and then created a child process to execute the */bin/less* command to display the content. Note that, since the */bin/less* command was executed as a child process, our system naturally collected its execution trace and considered it as part of the message processing instructions. In the meantime, since the decoded message, as a tainted data, was written into a local disk file, our system tracked the related "dirty" file operations. Specifically, when a piece of tainted data is being written to a file, our system will record additional information such as the file name, the current file position, the content and the number of bytes that are written into the file. In our implementation, we essentially create a virtual memory buffer to emulate the disk file. Later on, when the same file is open and the data being read falls into the recorded range, we will emulate a read operation from the virtual memory buffer and naturally re-taint the corresponding destination memory buffer. With that, ReFormat successfully discovered the transition function, i.e., *from64*. Further, ReFormat correctly identified the virtual memory buffer that emulated the disk file as the memory buffer that contains the decrypted message.

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* [38] 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 damages, 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 [38]. We want to point out that such manual efforts are simply for our controlled experiments and ReFormat is exactly designed to automatically reverse engineer the command format.

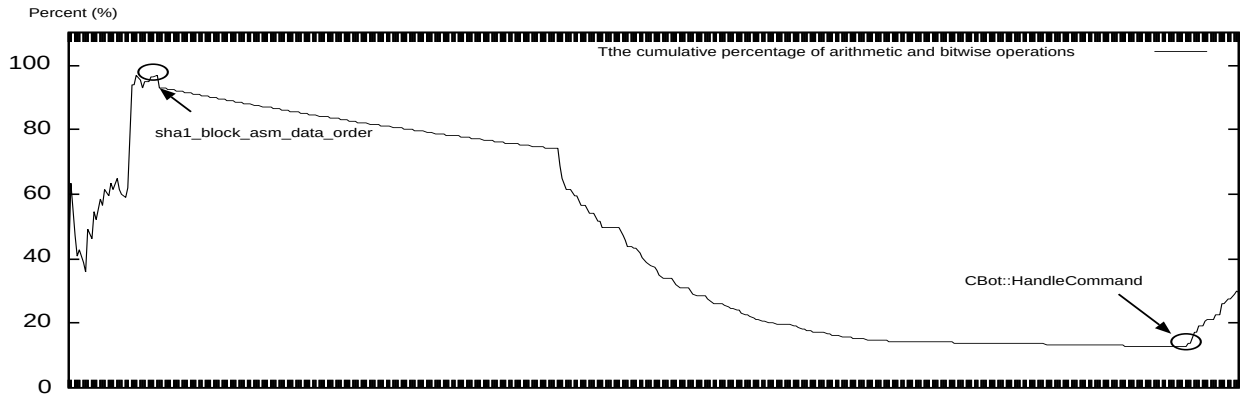


Figure 12. The cumulative percentage of arithmetic and bitwise operations in the collected trace when *agobot* handles the *.bot.execute /bin/ps* command

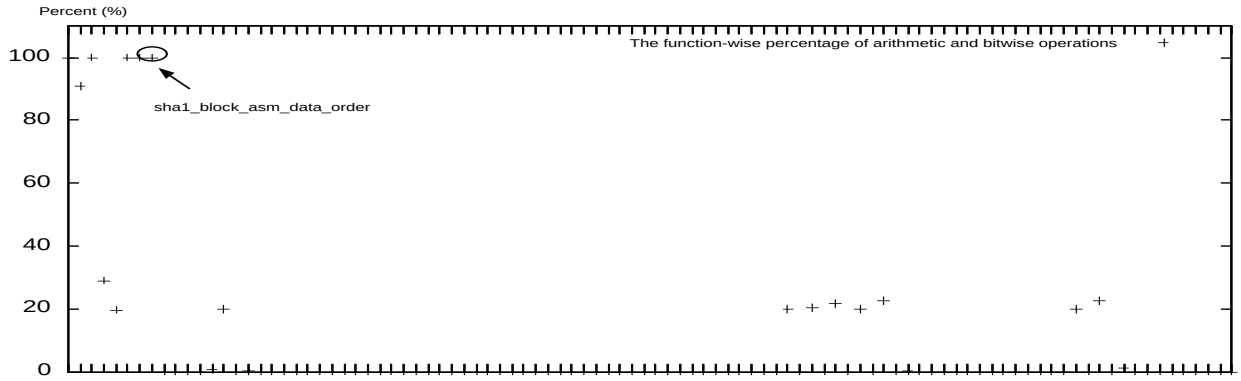


Figure 13. The function-wise percentage of arithmetic and bitwise operations when *agobot* handles the *.bot.execute /bin/ps* command

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 12 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 function-wise percentage of arithmetic and bit instructions (shown in Figure 13), we identified that *sha1_block_asm_data_order* is the transition function. The write set and the read set are shown in Figure 14(a) and 14(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 15.

5 Related Work

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

```
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
42c5188 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

Figure 14. Locating the decrypted message for the *.bot.execute* command

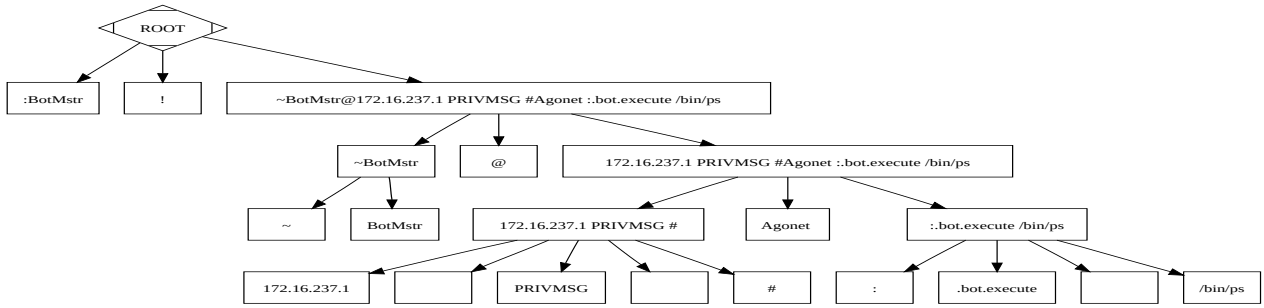


Figure 15. Revealing the *.bot.execute* command message format

on taint analysis [7, 9, 14, 22, 24, 25, 29].

As mentioned earlier, automatic protocol reverse engineering has recently received significant attentions due to its importance to many security applications. The Protocol Informatics (PI) project [3] and Discoverer [10] 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 [6], the system in [28], AutoFormat [18], and Tupni [12] 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. However, these systems work only for 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 runtime 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 [11] and ScriptGen [16, 17] replay a recorded network protocol session with another entity by identifying and updating certain input fields that are embedded in the recorded session. Replayer [21] uses binary analysis to replay an application-level dialog. None of these systems can han-

dle encrypted application-level communications. Protocol analyzers such as Wireshark [5] 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 [7, 8] 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 allows 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.

References

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

[2] SHTTP: an embeddable web server. *http://shttpd.sourceforge.net/*.

[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] 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.

[7] J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole-System Simulation. In *Proceedings of the 13th USENIX Security Symposium (Security'04)* 2004.

[8] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *Proceedings of the 14th USENIX Security Symposium (Security'05)* 2005.

[9] 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.

[10] 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.

[11] 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.

[12] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. *Proceedings of the 15th ACM Conferences on Computer and Communication Security (CCS'08)*, Oct. 2008.

[13] 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.

[14] 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.

[15] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[16] 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.

[17] 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.

[18] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[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] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Gnu wget. *http://www.gnu.org/software/wget/*.

[31] Apache Web Server. *http://www.apache.org/*.

[32] Mozilla Firefox. *http://www.mozilla.com/en-US/firefox/*.

[33] Microsoft Internet Explorer. *http://www.microsoft.com/windows/products/winfamily/ie/default.mspx*.

[34] Google Chrome. *http://www.google.com/chrome*.

[35] IRCD-Hybrid – High Performance Internet Relay Chat. *http://ircd-hybrid.com/*.

[36] XChat. *http://www.xchat.org/*.

[37] Metamail. *http://www.apache.org/*.

[38] Know Your Enemy: Tracking Botnets - Bot-Commands. *http://www.honeynet.org/papers/bots/botnet-commands.html*.