PAPER

FCReducer: Locating Symmetric Cryptographic Functions on the Memory

Ryoya FURUKAWA[†,††], ***Nonmember***, **Ryoichi ISAWA**[†††a)], ***Member***, **Masakatu MORII**[††], ***Senior Member***, **Daisuke INOUE**[†††], ***Member***, ***and*** **Koji NAKAO**[†††], ***Fellow***

SUMMARY Malicious software (malware) poses various significant challenges. One is the need to retrieve plain-text messages transmitted between malware and herders through an encrypted network channel. Those messages (e.g., commands for malware) can be a useful hint to reveal their malicious activities. However, the retrieving is challenging even if the malware is executed on an analysis computer. To assist analysts in retrieving the plain-text from the memory, this paper presents *FCReducer* (Function Candidate Reducer), which provides a small candidate set of cryptographic functions called by malware. Given this set, an analyst checks candidates to locate cryptographic functions. If the decryption function is found, she then obtains its output as the plain-text. Although existing systems such as CipherXRay have been proposed to locate cryptographic functions, they heavily rely on fine-grained dynamic taint analysis (DTA). This makes them weak against under-tainting, which means failure of tracking data propagation. To overcome under-tainting, FCReducer conducts coarse-grained DTA and generates a typical data dependency graph of functions in which the root function accesses an encrypted message. This does not require fine-grained DTA. FCReducer then applies a community detection method such as InfoMap to the graph for detecting a community of functions that plays a role in decryption or encryption. The functions in this community are provided as candidates. With experiments using 12 samples including four malware specimens, we confirmed that FCReducer reduced, for example, 4830 functions called by Zeus malware to 0.87% as candidates. We also propose a heuristic to reduce candidates more greatly.

key words: *malware, dynamic taint analysis, binary analysis, sandbox, community detection*

1. Introduction

There exists a type of malicious software (*malware*) specimens that encrypt the network channel between malware specimens and their herders. Since the channel is encrypted, analysts do not easily reveal their communication to answer, for example, *what commands are sent to the malware?* and *which kinds of contents are transmitted?* Examples of such malware include Zeus malware, which was used at a serious cyber banking fraud [1]. Zeus has been changed for Advanced Persistent Threats (APTs) [2], and its variants aim to seek and steal confidential information from the target in APTs, receiving commands from their herders on an encrypted channel. *Bot* [3] also uses encrypted channels to communicate with their herders.

When this type of malware is captured before/after carrying out cyber-attacks, one of essential steps for analysts is to reveal the malware's activities to prevent/mitigate the incident caused by the malware. In this context, our research focuses on how to retrieve the plain text transmitted between the malware and its herder on an encrypted channel. More precisely, we execute the captured malware on an analysis computer, and then we try to retrieve the transmitted plain-text messages from the memory of the computer. The plain text representing their communication is a useful hint to reveal the malware's activities, and it leads to making prevention/mitigation.

Retrieving the plain text is not easy even though the malware is executed on an analysis computer. One reason is that the malware usually calls too many functions to recognize which function is the cryptographic function. If it can be recognized, its output is a plain text in the case of the decryption function. However, it is impractical to check whether or not each function is the cryptographic function even if an analyst checks each within a few minutes. The number of called functions can peak at thousands, and she ends up spending much time on checking all the called functions.

Existing systems such as ReFormat [4] and CipherXRay [5] have been proposed to locate cryptographic functions. To do so, an idea shared between them is to use dynamic taint analysis [6], which is for tracking the propagation of an encrypted message on the memory. CipherXRay, for example, locates a function as the cryptographic function that causes *avalanche effect* [7], [8]. The avalanche effect is an essential property of cryptographic algorithms, which means that a bit data affects many other bits. This observation requires a fine-grained taint analysis at bit level granularity.

One drawback of these existing systems is that they heavily rely on dynamic taint analysis, and typical dynamic taint analysis can cause under-tainting [9], [10], which means failure of tracking data propagation. Under-tainting results in failing to pick up the correct cryptographic function because the existing systems need to track the propagating data at fine-grained granularity. Even worse, under-tainting is known to be easily caused by just implementing programs with some tricks such as pointer indirection (described in Sect. 2.2). This means that effectiveness of the

Manuscript received April 27, 2017.
Manuscript revised October 31, 2017.
Manuscript publicized December 14, 2017.

[†]The author is with PwC Cyber Services LLC, Tokyo, 100–0004 Japan.
[††]The authors are with Graduate School of Engineering, Kobe University, Kobe-shi, 657–8501 Japan.
[†††]The authors are with National Institute of Information and Communications Technology, Koganei-shi, 184–8795 Japan.
a) E-mail: isawa@nict.go.jp

DOI: 10.1587/transinf.2017EDP7143

systems can be degraded easily.

In this paper, we propose *FCReducer* (Function Candidate Reducer), a system for reducing candidates of cryptographic function on the memory. Its main purpose is to provide a small candidate set without relying on fine-grained dynamic taint analysis. To this end, it generates a data dependency graph between functions as follows. Given a program, FCReducer takes called functions that access the data propagated originally from an encrypted message, and it links the functions that share any data with each other in the data dependency graph. This does not require fine-grained taint analysis because FCReducer just cares about whether or not functions share any data. It then takes all the functions in the graph as candidates because they access the data propagated from an encrypted message.

FCReducer reduces those candidates because many candidates can be nominated in that manner. With *community detection* [11], FCReducer divides the functions in the graph into some groups called *community*, based on the density of the links. The decryption function can exist in a community roughly corresponding to the decryption phase of a given program. FCReducer considers the functions in the community as candidates. To more greatly reduce the candidates, another heuristic can be added to FCReducer. We use a heuristic focusing on an I/O (input-output) size of each function in this paper. This is based on the fact that the I/O size of symmetric cryptographic functions is the same. Due to this heuristic, this paper targets only symmetric cryptographic functions.

With experiments using eight testing programs linked to open source cryptographic libraries and four real world malware specimens, we measured effectiveness of five community detection methods and the I/O-size heuristic. Examples of those community detection methods include InfoMap [12] and FastGreedy [13]. The experiments confirmed that FCReducer reduced 1736 functions to 0.24% (4.1 functions) on average with two false negative cases out of 12 samples by combining InfoMap and the I/O-size heuristic. With no false negative cases, FCReducer reduced 1736 called functions to 4.63% (80.3 functions) on average by using FastGreedy. If an analyst can check each function in a candidate set within a few minutes, she can find the cryptographic function in a practical amount of time. Thus we can tell that FCReducer supports analysts to obtain and examine the input or output as plain-text messages.

We more clearly define a goal in this paper: FCReducer aims to provide a small function-candidate set that an analyst can check within a practical amount of time for a malware specimen. As an example, suppose that a set contains 50 candidates and that the elapsed time for checking one candidate is from one minute to five minutes. Under this assumption, an analyst spends from 50 minutes to around four hours to find out the cryptographic function. As a worst case, we accept *four hours* as a practical amount of time, considering results of a simple method focusing on the order of called functions. Actually, we show that FCReducer was able to provide smaller candidate sets than a set of 50 candidates in Sect. 4.6, and in that section we also show that it is more effective than that simple method.

Our contributions in this paper are follows:

- We propose FCReducer, a novel system to reduce candidates of cryptographic function. In particular, it is strong against programs that cause under-tainting.
- We add a heuristic based on I/O-size of functions to FCReducer. This shows extensibility of FCReducer. Like this heuristic, readers also apply their heuristics to specialize FCReducer against their target malware.
- We measured effectiveness of five community detection methods. This measurement helps analysts to choose the best method among them. There is a trade-off between reduction rate and false negative rate.

The rest of the paper is organized as follows. Section 2 introduces background knowledge of this research, and Sect. 3 presents FCReducer. Section 4 then evaluates effectiveness of FCReducer using eight testing programs and four real world malware specimens. After that, Sect. 5 briefly summarizes related work to clearly show the differences between FCReducer and existing systems. Finally, Sect. 6 concludes this paper.

2. Background Knowledge

2.1 How to Trace Functions

FCReducer traces all functions called by a given program during the execution of the program. This can be implemented by customizing virtualization solutions such as QEMU [14], PEMU [15], VirtualBox [16], and Xen [17] for monitoring each executed instruction (e.g., `add`, `push`, `jmp`, and `call`) at hypervisor level. It can be also implemented by using dynamic binary instrumentation tools such as Intel Pin [18] and Valgrind [19]. In this paper, we implement FCReducer based on QEMU because we are more familiar with it than the other solutions and tools.

QEMU provides an ability to flexibly call *user-defined functions* in the middle of execution of a given program. This kind of user-defined functions is named *helper*[†]. FCReducer traces functions with a helper that logs the behavior of functions called by a given program. For example, we create a helper that takes as input a 4-byte value of function address, and we insert this helper into the `disas_insn` function embedded in QEMU, as is shown in Fig. 1. The `disas_insn` plays a main role in executing the instructions of a given program. In the example, when the call instruction (0xe8) of a given program is executed, a helper named `gen_helper_TrackFunc` is also executed. At this time, the destination address of a call instruction is passed to `gen_helper_TrackFunc` through `tcg_const_i32(tval)`, and then it logs that address, where we define the destination address of `call` instructions as the beginning address of

[†]Please refer to *def-helper.h* of QEMU for the description of *helper*. This is contained in *qemu-2.1.2.tar.xz*, the source of QEMU 2.1.2. It can be downloaded from the web site [14].

```
static target_ulong disas_insn(CPUX86State *env,
DisasContext *s, target_ulong pc_start)
{
...
 switch (b) {
...
  case 0xe8: /* call im */
...
      tval += next_eip;
      gen_helper_TrackFunc(tcg_const_i32(tval));
```

Fig. 1 A customized code of QEMU to trace functions. `b`: an instruction of a given program to be executed, `next_eip`: the return address of a call, `tval` added by `next_eip`: the destination linear address of a call, `tcg_const_i32`(·): a function embedded in QEMU for passing a value to helper `gen_helper_TrackFunc`.

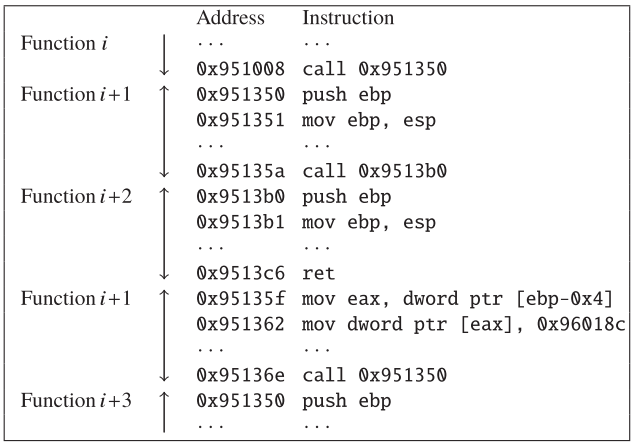

Fig. 2 Definitions of function and function ID, where $i+n$ denotes the function ID ($i, n = 0, 1, 2, \cdots$). The beginning address of a function is the destination address of a call instruction. The end of a function is the point where a ret instruction is executed (e.g., Function $i+2$ ends at `0x9513c6`).

functions. We can also monitor executed instructions (e.g., `mov` and `xchg`) of functions by inserting helpers into switch cases in `disas_insn` that correspond to instructions to be monitored.

We here define *function ID* because FCReducer assigns a function ID to each function while tracing functions in Sect. 3. Figure 2 shows an example of executed instructions. As is shown in the figure, the ID is incremented by one every time a call instruction is executed. Note that we allow different IDs to be assigned to an identical function.

2.2 Under-Tainting: Drawback of Dynamic Taint Analysis

Dynamic taint analysis (DTA for short) is a technique that enables a system to track propagation of a certain input to a given program as follows. A system marks a value of the target input with a taint tag as *tainted*. When another value is generated from the tainted value, its taint tag is propagated to the generated value. By successively marking every value that is generated from tainted values, the system can track the values that are propagated originally from the target input.

```
for (i = 0; i < 256; i++)
    array[i]=i;
...
var_out = array[var_in];
```

Fig. 3 A code snippet to cause under-tainting.

A serious drawback of typical DTA methods is *under-tainting*. This is caused if DTA methods fail to mark a value that in fact should be marked. Cavallaro et al. [20] demonstrate that under-tainting can be easily caused, for example, with the code snippet shown in Fig. 3 (a.k.a., *pointer indirection*). Under typical DTA, the taint tag of `var_in` is not propagated to `var_out` because the value of `var_in` is implicitly assigned to `var_out` through `array`. Like this, typical DTA methods do not propagate a taint tag if a value is generated implicitly from the tainted value.

2.3 Community Detection

In the science of networks [21], a graph representing a real system (i.e., a set of things working together for their purpose) is said to have *community structure*. That is, there are many edges that link vertices in the same community and comparatively few edges that link vertices in different communities. Such communities can be considered as fairly independent compartments of a graph, and vertices in a community play a similar role [11].

Community detection methods such as InfoMap [12] and WalkTrap [22] have been proposed, and they are implemented as modules in various programming languages, for example in the python-igraph module [23], [24]. By using such modules, we can detect communities in a graph.

3. Function Candidate Reducer

In the rest of the main body of this paper, we explain about the approach to *decryption function* alone for simplicity. To encryption function, we summarize that in Appendix A.

3.1 Assumptions about Execution Phases

FCReducer takes functions that access the data propagated from an encrypted message for generating the data dependency graph of functions. It then divides the functions in the graph into communities with community detection under the following assumptions.

We assume that network applications using an encrypted channel usually process an encrypted, input message and respond with an encrypted output message through the four execution phases [4]: decrypt the input message, process the decrypted message, generate the output message, and encrypt the output message. Network applications include a malware type of bot, which is our target in this paper. We also assume that the data dependency graph of functions has the community structure. That is, functions in the same community plays a similar role.

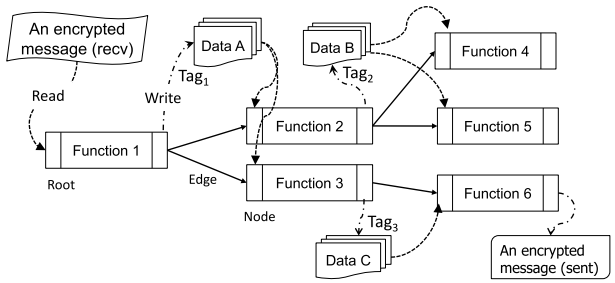

Fig. 4 An example of data dependency graph.

Under these assumptions, we expect that there is a community roughly corresponding to the decryption phase (i.e., the first phase). This community should be one of earlier communities since the decryption phase is the first phase. Thus we focus on the first community in this paper, which we describe as *root community* in Sect. 3.2.

3.2 Key Ideas

FCReducer aims to provide a small candidate set of cryptographic functions even if a given program possesses the code that causes under-tainting. More precisely, this set contains candidate addresses of the cryptographic function's beginning address on the memory.

1) Data Dependency Graph: A key idea behind FCReducer to avoid under-tainting is that it generates a data dependency graph of functions like Figure 4. In the graph, the root vertex corresponds to a function that reads an encrypted message received by the given program, and two functions are linked if a function reads the data propagated from the encrypted message written by the other function. As you can see in Fig. 4, any functions in the graph access the data propagated originally from an encrypted message. FCReducer takes all the functions in the graph as candidates, based on the fact that the decryption function must access such propagated data.

The above key idea strengthens FCReducer against under-tainting such as the code described in Sect. 2.2. In that code, the value of `var_in` is assigned to `var_out` through `array` to hinder the propagation of `var_in`'s taint tag to `var_out`. FCReducer overcomes this code because it does not have to propagate `var_in`'s taint tag to `var_out`. Suppose that `var_in` is marked with the taint tag of `function 1` and that `function 2` assigns a value of `var_in` to `var_out` through `array`. Even in this case, FCReducer just links `function 1` to `function 2`, and it marks `var_out` with the taint tag of `function 2`.

2) Community Detection: In exchange for avoiding under-tainting, coarse-grained taint analysis usually nominates many function candidates in the graph. To reduce those candidates, a key idea behind FCReducer is that it detects communities hidden in the graph with community detection methods, based on the internal edge density of vertices (i.e., functions) in subgraphs. To this end, we treat data dependency graphs as the directed graph in graph theory,

1: **procedure** GENERATEGRAPH(I_i)
2: **if** I_i = CALL **then**
3: push id on $stack$
4: $global\_id \leftarrow global\_id + 1$
5: $id \leftarrow global\_id$
6: add V_{id} to G
7: **else if** I_i = RET **then**
8: pop id from $stack$
9: **else if** I_i = WRITE $addr\ val$ **then**
10: $M_{addr} \leftarrow id$
11: **else if** (I_i = READ $addr\ val$) and (M_{addr} is tainted) **then**
12: $k \leftarrow M_{addr}$
13: add $E_{k,id}$ to G
14: **end if**
15: **end procedure**

Fig. 5 A procedure for generating a graph.

where vertices and edges correspond to functions and data dependency, respectively.

We use the internal edge density to detect communities. This is because Functions in the same community pass data to each other for their role, and functions in different communities rarely pass data outside the community. That is, the internal edge density in a community should become high due to passing data inside the community. After detecting communities, FCReducer takes as candidates the functions in the first community that contains the root vertex. We call this community *root community*.

3) I/O-size Heuristic: We adopt a heuristic that prioritizes functions in the root community whose I/O (Input and Output) size is the same. This is based on the fact that the I/O size of symmetric cryptographic functions is the same such as AES [25] and RC4 [26]. Given a candidate set, analysts first check those prioritized candidates, and then they check the rest of candidates.

3.3 Methods

We describe FCReducer in more detail and formalize some methods applied to it in this section.

1) Data Dependency Graph: FCReducer generates a directed graph representing data dependency between functions in the following manner. It executes a given program on QEMU with helpers, and it monitors each instruction to track if the given program receives an encrypted message from a server. If it is, FCReducer marks the memory area with `ID 0` where the encrypted message has been restored. Then, if `function 1` reads the memory area marked with `ID 0`, FCReducer memorizes `function 1` as the root function. When this function writes any data into a memory area, it marks this area with `ID 1`. After that, if `function 2` reads the area marked with `ID 1`, FCReducer links `function 1` to `function 2`. FCReducer then links `function 1` to `function 3` if `function 3` reads the area marked with `ID 1`.

We formalize this graph-generation algorithm as procedure *GenerateGraph* in Fig. 5, where the symbols in this procedure are listed in Table 1. Before GenerateGraph is called, graph G is initialized with empty set $\emptyset$, and variable

Table 1 List of the symbols used in the procedure in Fig. 5.

Symbol	Description
G	A directed graph G=(V,E)
V	A set of vertices representing functions of a given program
V_n	The n-th vertex representing a function
E	A set of edges representing data dependency between vertices
$E_{p,q}$	The edge linking V_p to V_q $(p, q = 1, 2, \cdots, n)$
I_i	The instruction executed for the i-th time since an encrypted message was received
id	Positive integers representing the ID of functions
$addr$	A memory address
M_{addr}	The ID of a function that writes data to the memory area of $addr$ (i.e., the memory area of $addr$ is tainted.)
$stack$	LIFO (Last In and First Out)

```
1: procedure MergeAreas(S_id)
2:     for i ← START to END do
3:         if S_{id,i} overlaps or is contiguous with S_{id,i+1} then
4:             if h_{id,i} ≤ h_{id,i+1} then
5:                 S_{id,i+1} ← [l_{id,i}, h_{id,i+1}]
6:                 S_{id,i} ← null
7:             else
8:                 S_{id,i+1} ← S_{id,i}
9:                 S_{id,i} ← null
10:            end if
11:        end if
12:    end for
13: end procedure
```

Fig. 6 A procedure for mering overlapped areas. *null*: a flag signalizing that this area is merged, $S_{id} \in \{SR_{id}, SW_{id}\}$.

global_id is set to zero. After a given program has received an encrypted message from the server, FCReducer begins and keeps passing each executed instruction to GenerateGraph for generating the graph.

2) Community Detection: Taking G as input, FCReducer detects communities hidden in G with a community detection method. The community detection can be done by using python igraph. For example, InfoMap is implemented as *community_infomap* [27]. After detecting communities, FCReducer takes as candidates the beginning addresses of the functions in the root community.

3) I/O-size Heuristic: FCReducer prioritizes functions, focusing on their I/O size. We define *input* and *output* of a function as a contiguous memory area read by the function and one written by the function, respectively. FCReducer memorizes every contiguous read area as *input* and every contiguous written area as *output*. It then takes a function as candidate if any one of *input* and any one of *output* are the same size. If not, FCReducer determines this function should not be a symmetric cryptographic function.

We formalize this heuristic as follows. For *input*, let $RR = \{RR_{id} \mid 0 \leq id \leq n-1\}$ be a set of memory areas read by function id, where n denotes the number of functions called by a given program. RR_{id} is then expressed as $\{RR_{id,j} \mid 0 \leq j \leq m-1\}$, and $RR_{id,j}$ represents the memory area read by IR_j, where IR_j and m denote a read instruction of function id executed for j-th time $(j = 0, 1, \cdots)$ and the number of read instructions of function id. Here we use notation $[l, h]$ to denote the memory area between addresses l and h, inclusive[†], and $RR_{id,j} = [l_{id,j}, h_{id,j}]$. For example, if the second read instruction of function 3 reads the memory area between 0x100 and 0x200, $RR_{3,1} = [0x100, 0x200]$ is added to RR_{id}. While monitoring each instruction, FCReducer memorizes the read memory areas in RR.

FCReducer sorts the elements of each RR_{id} by lower address l to recognize contiguous memory areas. We express the sorted RR_{id} as SR_{id}. For example, if RR_0 = {[0x150, 0x400], [0x500, 0x600], [0x100, 0x200], [0x050, 0x100]}, it is sorted as SR_0 = {[0x050, 0x100], [0x100, 0x200], [0x150, 0x400], [0x500, 0x600]}. At this time, $SR_{0,0}$ is contiguous with $SR_{0,1}$, and $SR_{0,1}$ overlaps with $SR_{0,2}$. FCReducer then merges memory areas $SR_{id,k}$ that overlaps or are contiguous with each other, according to procedure *MergeAreas* in Fig. 6. In the above example, the memory areas in SR_0 are merged as {(*null*), (*null*), [0x050, 0x400], [0x500, 0x600]}, where *null* denote a flag signalizing that this area is merged. *Output* can be formalized as SW_{id} in the same manner as *input*.

Finally, for each function candidate, FCReducer prioritizes function id if the following conditions are satisfied. First, the merged SR_{id} contains a memory area $SR_{i,k}$ whose size is equal to at least one of memory areas contained in the merged SW_{id}. Second, that size is larger than b bytes. The second condition is to avoid the situation where their size matches by accident, and we set b to eight in this paper, which is the same as CipherXray [5] uses. If SR_{id} contains [0x500, 0x600] and SW_{id} contains [0x750, 0x850], function id is prioritized.

In summary, to get a candidate set of decryption functions, FCReducer first executes a given program, and generates the data dependency graph by taking functions that access the data propagated from an encrypted message. It then detects communities in the generated graph, and takes functions in the root community as candidates. After that, among those candidates, FCReducer prioritizes functions whose I/O size is the same. Given this candidate set, an analyst first check whether or not each prioritized candidate is a decryption function, and then check the rest of candidates if the decryption function does not exist in the prioritized candidates.

3.4 Limitations

FCReducer has the following two limitations. First, the current version of FCReducer cannot deal with malware specimens that call functions without `call` and/or `ret` instructions because it recognizes functions based on a pair of those instructions. An example of such obfuscation techniques is as follows: a malware specimen executes a `push` instruction to put a return address on the stack, and then it transfers the instruction pointer (a.k.a. *program counter*) to the beginning of a function with a `jmp` instruction. After this function is finished, the malware executes a `pop` instruction to obtain the return address and transfers the instruction pointer to that address. Currently, this kind of obfuscations is out of

[†]We quote this notation from paper [28].

our scope. In our future work, we should implement special case rules for such obfuscated function calls as a pair of `push-jmp` and `pop-jmp` to defeat them.

Second, the I/O-size heuristic will definitely cause false negative cases (i.e., it will prioritize candidates that are, in fact, not decryption functions) in cases where malware authors design decryption functions that return the output whose size intentionally differs from the input. For those obfuscated functions, an analyst is forced to check the incorrectly prioritized candidates first, and then checks the rest of candidates (i.e., not-prioritized candidates). This makes her malware analysis late. However, after that, she can reach the decryption function if it is contained in the rest of candidates. In addition, the I/O-size heuristic works without depending on community detection, and so it can be replaceable and also should be replaced with another heuristic at the time when the I/O-heuristic has become clearly ineffective against malware specimens to be analyzed.

4. Experimental Evaluation

Every sample used in this evaluation contained one decryption function and one encryption function. We checked how many candidates of decryption function were reduced by FCReducer to measure its effectiveness.

4.1 Dataset and Environment

We generated eight testing programs that communicated with our C&C (Command and Control) server on an encrypted network channel. This channel was created using cryptography libraries, which were Beecrypt [29], Brian Gladman [30], Crypto++ [31], and OpenSSL [32]. Column 2 of Table 2 lists the symmetric-key algorithms implemented by those libraries. Each testing program used one algorithm of a library.

In addition, we captured four real world malware specimens on the Internet, which were Alina, Grum, Pony, and Zeus. They were all categorized into bot, and they communicated with a C&C server on an encrypted channel. Alina and Grum used an XOR operation for the encrypted channel, and Pony and Zeus used RC4. We configured those malware specimens to communicate with our C&C server.

For checking if FCReducer was strong against under-tainting, we customized the above 12 samples (i.e., the eight testing programs and the four malware specimens) with the code shown in Fig. 3 to cause under-tainting on purpose. With that code, they copied an encrypted message received from our C&C server to another memory area.

As an evaluation environment, we modified QEMU 2.1.2 to implement FCReducer in it, and installed Windows 7 (32-bit) into the modified QEMU as a guest OS. We set up this QEMU and our C&C server in an isolated network. Each of the 12 samples was executed under FCReducer until the sample sent an output message to our C&C server.

The community detection methods FCReducer used were InfoMap [12], WalkTrap [22], FastGreedy [13], MultiLevel [33], and SpinGlass [34]. They were implemented as a python library named *python igraph* [24], and we used default parameters of community detection methods set up in python igraph.

Table 2 Results of community detection methods. B-GLAD: Brian Gladman, Bfish: Blowfish, All: # of all functions called by a sample, RAF: a ratio of # of provided candidates to "All" (RAF is defined in Eq. (1)), Graph: graph G (whose RAF value is calculated as: # of functions in G divided by # of all functions shown in "All"), IM: InfoMap, WT: WalkTrap, FG: FastGreedy, ML: MultiLevel, SG: SpinGlass, *: a false negative case (the decryption function was not contained).

Samples		All	RAF (%)					
		(#)	Graph	IM	WT	FG	ML	SG
Beecrypt	AES	1260	19.37	0.79	3.81	6.75	7.62	8.81
	Bfish	1137	8.71	0.53	6.24	8.53	8.53	8.27
B-GLAD	AES	1575	20.95	0.70	0.25	4.00	2.54	3.24
OpenSSL	AES	1155	23.81	0.26	0.43	0.52	5.45	8.74
	Bfish	1433	8.37	0.35	0.28	7.82	4.54	3.07
	DES	1308	22.55	0.23	0.23	0.46	0.46	12.54
	RC4	1549	22.72	0.39	0.32	0.39	0.65	0.65
Crypto++	RC4	1102	29.49	0.27	0.27	0.27	0.27	0.27
Alina	XOR	2275	18.33	0.66	8.75	6.64	7.03	*4.44
Grum	XOR	2038	8.78	0.74	2.55	3.43	2.70	2.99
Pony	RC4	1171	68.06	*0.26	*0.26	3.25	5.89	26.39
Zeus	RC4	4830	31.76	0.87	1.80	6.77	6.67	10.31
Average		1736	23.84	0.59	2.32	4.63	4.73	7.42

4.2 Evaluation Metric and Purposes

To evaluate effectiveness of FCReducer over 12 samples, we measure how small candidate sets FCReducer provided. To this end, we define the following ratio:

$$RAF = \frac{\text{\# of provided candidates}}{\text{\# of all called functions}} \times 100 \quad (1)$$

RAF (a Ratio to All called Functions) represents a ratio of the number of candidates provided by FCReducer to the number of all *unique* functions called by a given sample. The *unique* indicates that we increment *# of all called functions* by one only once even if an identical function is called more than once. In the right-hand side of Eq. (1), the ratio is multiplied by 100 to notate RAF as a percentage. RAF means that the smaller it is, the fewer candidates an analyst should check, and so we can tell that obtained results are better when RAF values are smaller.

In addition, we check if false negative cases caused or not. False negative cases means the cases in which a candidate set does not contain the beginning address of the decryption function.

The main purposes of the experiments in Sects. 4.3, 4.4, and 4.5 are to check RAF values to see how many function candidates were reduced with the data dependency graph, community detection methods, and the I/O-size heuristic, respectively. Basically, we can see more and more candidates were reduced overall as we progress from one section to the next. In addition, Sect. 4.4 presents a countermeasure against false negative cases using multiple community detection methods (`Proc_CD` for short), and Sect. 4.5 also presents a countermeasure taken with the I/O-

size heuristic (`Proc_I/O` for short).

The main purpose of Sect. 4.6 is to answer the questions: "*How many candidates should be reduced for an analyst to find out the decryption function in a practical amount of time?*" and "*Can FCReducer achieve it?*" To this end, the section first introduces a simple method focusing on the order of called functions as a benchmark, and then it aims to answer the questions by conducting a comparison mainly between `Proc_I/O` and that simple method.

4.3 Size of Data Dependency Graph

Of the 12 samples, FCReducer executed each and memorized the addresses of all unique functions called by the sample. We took those unique functions as an initial candidate set of the decryption function. The size of the initial candidate set is shown in column "*All*" of Table 2. In other words, each value in column "All" means the number of the addresses of all unique functions extracted from each sample.

Among "All", the testing program of Crypto++ called fewest unique functions, the number of which is 1102. Even this testing program should assign a very time-consuming task to an analyst when she checks if each called function is the decryption function. Even worse, Zeus, a real world malware, called most unique functions, the number of which is 4830. We clearly confirmed that it was necessary to reduce the size of the initial candidate set.

While executing each sample, FCReducer generated graph G according to the *GenerateGraph* procedure in Fig. 5. We took those functions as the second candidate set, which access the data propagated from an encrypted message. In fact, we confirmed that the decryption function was contained in the second candidate set of every sample.

The RAF values of the second candidate sets are shown in column "*Graph*" of Table 2. In column "Graph", the RAF value of Zeus is 31.76%. This means that the candidates were reduced to 31.76% of "All", the number of which equaled 1534. The best case among malware samples was the result against "Grum", in which the candidates were reduced to 8.78% of "All". Even in the best case, 179 candidates still remained, and we sensed that the candidates should be more reduced.

4.4 Results of Community Detection Methods

After detecting communities hidden in the graph for each sample, we took the functions in the root community as the *third candidate set*. The RAF values of the third candidate sets are shown in the right five columns of Table 2, where columns "*IM*", "*WT*", "*FG*", "*ML*", and "*SG*" correspond to InfoMap, WalkTrap, FastGreedy, MultiLevel, and SpinGlass.

Regarding Table 2, InfoMap, for example, obtained a RAF value of 0.87% against Zeus. This means that the candidates were reduced from 4830 to 42. If an analyst can check each candidate within a few minutes, this manual check can be achieved in a practical amount of time. InfoMap also obtained a RAF value of 0.59% on average. It was the best among all community detection methods. However, InfoMap caused a false negative case against Pony. This is because the root community of Pony contained just three functions, and it was so small that the decryption function could be excluded from the root community. InfoMap detected 3494 communities in the graph of Pony.

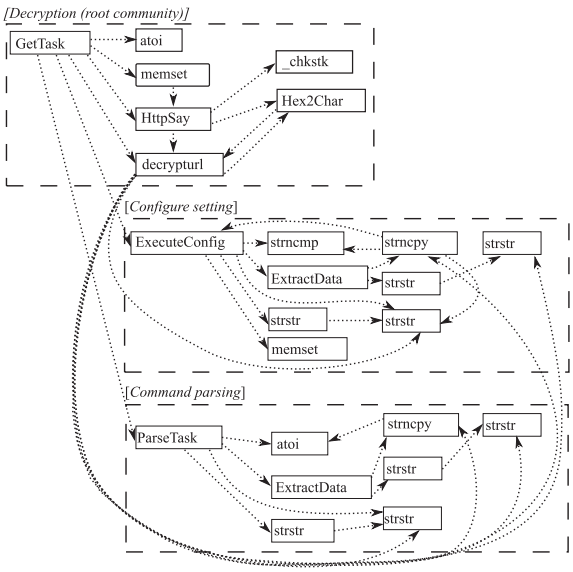

Fig. 7 Part of the graph of Grum obtained with FastGreedy.

The average number of communities detected by IM, WT, FG, ML, and SG were 1214, 407, 51, 41, and 30, respectively. Typically, the probability of false negative cases occurring tends to increase as detected communities increase, due to the decreasing of the candidates in the root community. Based on this, the false negative cases of InfoMap and WalkTrap for Pony could make sense; but reasoning about the false negative of SpinGlass for Alina is out of scope in this paper.

We introduce a countermeasure[†] against false negative cases as follows. An analyst checks the candidate sets obtained with InfoMap, WalkTrap, FastGreedy, MultiLevel, and SpinGlass in order. This order is determined by the average RAF values in Table 2. If an analyst takes this countermeasure against Pony, she checks the first three sets in total since the set of FastGreedy contained the decryption function. She eventually checks 3.77% candidates of "All" (i.e., 0.26% + 0.26% + 3.25%) including duplicate candidates between those sets. Manually checking 3.77% candidates (# of candidates: 44) can be done in a practical amount of time. For the other malware samples, this countermeasure produced the same results as InfoMap because InfoMap did not cause no false negative cases for them. That is, the results are 0.66% of "All" for Alina, 0.74% for Grum, and 0.87% for Zeus.

[†]`Proc_CD` (Community Detection) is short for the procedure of this countermeasure, which is cited in Sects. 4.5 and 4.6.

Table 3 Results of community detection methods with the I/O-size heuristic.

Samples		All	RAF (%)					
		(#)	Graph	IM	WT	FG	ML	SG
Beecrypt	AES	1260	15.71	0.48	2.86	4.92	5.56	6.75
	Bfish	1137	6.33	0.18	4.40	6.24	6.24	5.98
B-GLAD	AES	1575	8.57	0.06	0.06	0.83	0.19	1.08
OpenSSL	AES	1155	8.83	0.09	0.09	0.09	1.90	2.77
	Bfish	1433	2.44	0.28	0.21	2.16	1.26	1.12
	DES	1308	7.72	0.23	0.15	0.23	0.23	4.43
	RC4	1549	7.23	0.19	0.19	0.26	0.32	0.32
Crypto++	RC4	1102	19.69	0.27	0.27	0.27	0.27	0.27
Alina	XOR	2275	5.27	0.26	2.99	2.51	2.73	*1.76
Grum	XOR	2038	*3.48	*0.34	*0.98	*1.32	*1.08	*1.13
Pony	RC4	1171	19.13	*0.00	*0.00	1.02	1.79	8.54
Zeus	RC4	4830	9.59	0.27	0.46	2.36	2.28	3.29
Average		1736	8.88	0.24	1.00	1.91	1.97	2.91

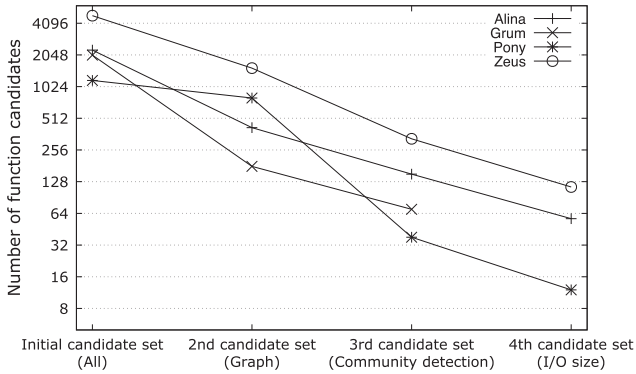

Fig. 8 A result summary of FastGreedy against the four malware samples. The vertical axis is scaled logarithmically. The symbol of Grum for the fourth candidate set is not plotted because this set did not contain the decryption function. It contained 27 function candidates.

In Fig. 7, we give an example of graph G to show how communities were organized. This example is generated based on the graph of Grum obtained with FastGreedy in the following manner. We first remove vertices corresponding to windows API functions [35] such as *CreateFile* and *FindFirstFile* from the original graph to make the graph easy to see. We then pick up three communities that contain functions whose IDs are assigned earlier among all functions. After that, we manually check the roles of those communities shown in the figure: decryption, configure setting, and command parsing. In this example, we can see the communities roughly corresponding to their roles.

Regarding the results shown in Table 2 and the countermeasure against false negative cases introduced in this section, we can tell that FCReducer, in particular community detection, can be effective to reduce function candidates of decryption functions.

4.5 Results of I/O-Size Heuristic

InfoMap provided very good RAF values for the 12 samples; however, it caused a false negative case against Pony. WalkTrap also caused it against Pony. In contrast, FastGreedy did not cause any false negative cases for the 12 samples, but it was inferior to InfoMap and WalkTrap, in terms of the average RAF value. In this section, we first focus on how well the heuristic based on the input/output (I/O) size of functions improved RAF values of FastGreedy. We then introduce a countermeasure taken with the I/O-size heuristic against false negative cases.

The I/O-size heuristic prioritized the candidates in the third candidate sets, which had been obtained with the community methods. We took those prioritized candidates as *the forth candidate set*. The RAF values of this set are shown in the right five columns of Table 3. Overall, the RAF values were clearly reduced. In particular, the RAF values of FastGreedy for Alina and Zeus were reduced from 6.64% (# of candidates: 160) to 2.51% (#: 62) and from 6.77% (#: 322) to 2.36% (#: 110), respectively.

Against only Grum, the heuristic caused a false negative case. The reason of this was a hex encoding scheme of Grum. That is, when Grum received an encrypted message, it had been encoded into a sequence of hex characters. Then the decryption function of Grum first decoded an encoding of the encrypted message to binary data, and then decrypted the decoded data. At this time, the size of the binary data was half that of the encoding. Consequently, the I/O size varied. For the rest 11 samples, we confirmed that the I/O size of their decryption functions was the same.

Figure 8 shows a result summary of FastGreedy. The set for Zeus was largest among the fourth candidate sets. Although it should be a tough task to check each candidate, this task could be still done in a practical amount of time. For Grum, although a false negative case happened, an analyst could reach the decryption function by checking the not-prioritized candidates (i.e., the third candidate set) after checking those prioritized.

In more detail, we describe a countermeasure against false negative cases caused with the I/O-size heuristic as follows. FCReducer applies IM to obtain the third candidate set, and then it applies the I/O-size heuristic to obtain the fourth candidate set. After that, an analyst checks the fourth candidate set before the third. At this time, if the decryption function is not found, the analyst then checks the third candidate set. Again, if the decryption function is not found, FCReducer applies WT and the I/O-size heuristic to obtain the third and fourth candidate sets, respectively. After that, the analyst checks this fourth candidate set. Like this, FCReducer conducts this procedure for IM, WT, FG, ML, and SG in turn until the decryption function is found. In practice, FCReducer always conducts this procedure because in advance it does not know if false negative cases occur or not.

Conducting the procedure of this countermeasure (`Proc_I/O` for short), an analyst eventually checks 0.26% candidates of "All" for Alina, 1.08% (i.e., 0.34% + 0.74%) including duplicate candidates for Grum, 1.54% (i.e., 0.00% + 0.26% + 0.00% + 0.26% + 1.02%) including duplicate candidates for Pony, and 0.27% for Zeus. The results for Alina, Pony, and Zeus are better than the corresponding re-

sults[†] that were obtained according to the countermeasure taken without the I/O-size heuristic (Proc_CD for short), which is described in the third paragraph of Sect. 4.4. The result of Proc_I/O for Grum is worse than Proc_CD, due to the false negative case. For the eight testing samples, the results of Proc_I/O are better than Proc_CD because of no false positive cases occurring.

From the perspective of reducing the candidates, we can tell that the I/O-size heuristic can be effective overall. In addition, regarding the result for Grum, even if the I/O-size heuristic causes false negative cases, an analyst can reach the decryption function, thanks to Proc_I/O. On the other hand, however, we also confirm that false negative cases caused with the I/O-size heuristic can degrade RAF values. Thus this heuristic should be replaced if malware samples to be analyzed often use binary-to-text encoding schemes like the hex encoding of Grum. Other heuristics can be applied to FCReducer without depending on community detection.

4.6 Benchmarking

As a benchmark for Proc_CD and Proc_I/O, we introduce a simple procedure for locating decryption functions. After a malware sample receives an encrypted message, an analyst starts to check functions by hand in the order in which functions are called by the malware until the decryption function is found. During this, the analyst skips a function if it has been already called and checked (i.e., a function is checked only once even if it is called more than once).

To evaluate this simple procedure (Proc_HAND for short), we followed Proc_HAND, and eventually checked 306 unique functions for Alina, 253 unique functions for Grum, 184 unique functions for Pony, and 811 unique functions for Zeus. We then divided the number of checked functions by each value in "All" of Table 2 (i.e., the number of all unique, called functions) for each malware to make the number of checked functions approximate to a RAF value. Table 4 shows the RAF values of Proc_CD and Proc_I/O, which are shown in the previous section, and the approximate RAF values of Proc_HAND. This table confirms that both Proc_CD and Proc_I/O outperformed Proc_HAND, and it also ensures that focusing on community structure in a data dependency graph and the I/O-size of functions is much more effective than the order of called functions for locating decryption functions.

Finally, we consider whether or not Proc_I/O provided small candidate sets that an analyst could check within a practical amount of time. For Proc_I/O, the number of checked candidates is calculated by '*All*' × '*RAF*' / 100 in Table 4 under the assumption that the decryption function was the finally-checked candidate in any sets. As the best case, it was 6 for Alina, and for Pony it was 22 as the worst case.

The elapsed time for checking one candidate heavily depends on know-how of an analyst, but we assume it is from one minute to five minutes from our experiences of this evaluation. In this case, the total elapsed time for Alina is from 6 to 30 minutes, and that for Pony is from 22 to 110 minutes. We accept around *two hours* as a practical amount of time because even in the best case of Proc_HAND, which is the case of Pony, the total elapsed time is from 183 to 915 minutes. At least in this experiment, we conclude that Proc_I/O (i.e., FCReducer) was able to provide small candidate sets.

[†]The corresponding results are 0.66% for Alina, 3.77% (i.e., 0.26% + 0.26% + 3.25%) for Pony, and 0.87% for Zeus. The result for Grum is 0.74%. The results are summarized in Table 4.

Table 4 Comparison between Proc_CD, Proc_I/O, and Proc_HAND. RAF: a ratio of # of provided candidates to "All" (RAF is defined in Eq. (1)), All: # of all functions called by a sample.

	All (#)	RAF (%)		
		Proc_CD	Proc_I/O	Proc_HAND
Alina	2275	0.66	0.26	13.45
Grum	2038	0.74	1.08	12.41
Pony	1171	3.77	1.54	15.71
Zeus	4830	0.87	0.27	16.79
Average	2579	1.51	0.79	14.59

5. Related Work

In this section, we introduce existing systems that can locate cryptographic functions on the memory.

5.1 Systems Based on Dynamic Taint Analysis

Wang et al. [4] propose a system named *ReFormat* to identify the format of messages transmitted between a given program and the server on an encrypted channel. It can also locate cryptographic functions on the memory as follows. With dynamic taint analysis, ReFormat records how an encrypted message is being processed by instructions, tracking its propagation. If a ratio of bitwise and arithmetic instructions reaches a predefined threshold value, ReFormat determines this memory address is the beginning of the cryptographic function. This is based on the fact that cryptographic algorithms are usually implemented with those types of instructions (e.g., xor and add). A drawback of ReFormat is that it requires a fine-grained taint analysis to track the propagation of an encrypted message. This makes ReFormat weak against the code that causes under-tainting like Figure 3.

Li et al. [5] propose a system named *CipherXRay* to locate cryptographic functions and secret keys embedded in a given program even if the program is obfuscated with software packers such as ASProtect [36]. To this end, with dynamic taint analysis, CipherXRay tracks the propagation of an encrypted message to observe avalanche effect [7], [8], which appears during the cryptographic processing. The avalanche effect is an essential property of cryptographic algorithms in which slight change in the input of cryptographic functions causes significant change in the output. If the avalanche effect is detected in a function, CipherXRay determines that it is the cryptographic function. CipherXRay is robust against software packers under the

assumption that the avalanche effect is still occurred under obfuscation. However, it requires fine-grained dynamic taint analysis to observe avalanche effect at bit level granularity. Thus CipherXRay is also weak against under-tainting.

5.2 Systems Based on Prior Knowledge

Gröbert et al. [37] extracts signatures shared between standard cryptographic libraries such as OpenSSL and Beecrypt. Based on extracted signatures, their system can identify cryptographic algorithms of a given program such as AES and RC4. It can also locate the cryptographic function on the memory. However, their system heavily relies on the previously extracted signatures, and this means that it is weak against modified cryptographic functions, as is described in their paper itself [37]. In contrast, FCReducer does not rely on such signatures.

Calvet et al. [38] propose a system named *Aligot* to identify and locate cryptographic functions. Likewise to CipherXRay, Aligot is designed to be robust against software packers as follows. Aligot finds a candidate of the cryptographic function, focusing on cryptographic characteristics like *loops* (i.e., executing a sequence of instructions multiple times). It then extracts its input and output parameters, and pass the input to known cryptographic functions. If the output of a known cryptographic function matches the output of the candidate, Aligot determines the candidate is the cryptographic function. Aligot is robust against software packers under the assumption that the input-output relationship is still maintained under obfuscation. In terms of this robustness and the ability to uniquely locate the cryptographic function, Aligot is superior to FCReducer. However, Aligot relies on known cryptographic functions.

5.3 Dynamic Taint Analysis against Under-tainting

There are existing schemes for dynamic taint analysis (DTA for short) against under-tainting including DTA++ [39] and Newsome's scheme [40]. Those, however, limit their scope to applying DTA to benign applications that should be protected (e.g., from exploit code targeting applications' vulnerabilities) [39], [40]. The under-tainting intentionally caused by malware will be very complicated [9], [39]. In this section, we briefly introduce DTA++, considering that its key idea could be still extended for under-tainting caused by malware.

Simply propagating taint tags along all *implicit flows* can avoid under-tainting itself; however, it will cause the explosion of tainting (i.e., *over-tainting*), instead. That is, the matter is the trade-off between under-tainting and over-tainting. To only propagate taint tags along significant implicit flows, the key idea behind DTA++ is to focus on the implicit flows within *information-preserving transformations* (IPT for short). For example, an implicit flow "`if (x==0) y=0 else if (x==1) y=1···else if (x==0xff) y=0xff`" is within an IPT, and an implicit flow "`if (x==0) y=0 else y=1`" is without an IPT, where `x` and `y` are one-byte variables. In the former, each output value (`y`'s value) is determined by one input value (`x`'s value), which means the information of input is completely preserved within transformation. This means that this kind of implicit flows can completely transform a malicious input to the output with information-preserving, and so it should be significant. In contrast, in the latter implicit flow, the output value "`y=1`" can be caused by many different input values, which means that transforming a malicious input will be restricted or hampered. This is the reason why the latter implicit flow will not be significant. DTA++ can effectively detect implicit flows within IPTs from execution traces of (benign) applications, and it additionally propagate taint tags along them, in which under-tainting would occur if DTA++ did not additionally propagate the taint.

6. Conclusion

This paper presents FCReducer, a system for providing a small candidate set of symmetric cryptographic functions. Given this candidate set, an analyst checks if each candidate is the decryption function or not. Then she obtains the output of the decryption function, which corresponds to the plain-text message. This can be done in a practical amount of time, which is confirmed by the evaluation of Sect. 4. In addition, FCReducer is stronger against under-tainting than existing systems because it just links functions based on their data dependency with coarse-grained dynamic taint analysis. This means that FCReducer covers a kind of programs that cause under-tainting, which existing systems cannot easily deal with.

In our future work, we should conduct parameter tuning of community detection methods for obtaining smaller candidate sets and avoiding false negative cases, although there must be a trade-off between them. In addition, dealing with malware specimens using *multiple encryption* is also in our future work, and a consideration against it is given in Appendix B.

References

[1] FBI (Federal Bureau of Investigation), "Cyber Banking Fraud." https://archives.fbi.gov/archives/news/stories/2010/october/cyber-banking-fraud, 2010.

[2] D. Sullivan, "Beyond the Hype: Advanced Persistent Threats." http://la.trendmicro.com/media/misc/ebook-advanced-persistant-threats-and-real-time-threat-management.pdf, 2011.

[3] R. Puri, "Bots & botnet: An overview," SANS Institute, vol.3, p.58, 2003.

[4] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," Proc. 14th European Conference on Research in Computer Security, ESORICS'09, vol.5789, pp.200–215, Springer-Verlag, 2009.

[5] X. Li, X. Wang, and W. Chang, "CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution," IEEE Trans. Dependable Secur. Comput., vol.11, no.2, pp.101–114, March 2014.

[6] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity

software," 2005.
[7] H. Feistel, "Cryptography and computer privacy," Sci. Am., vol.228, pp.15–23, 1973.
[8] A.F. Webster and S.E. Tavares, "On the design of s-boxes," Advances in Cryptology — CRYPTO '85 Proceedings, ed. H.C. Williams, pp.523–534, Springer, 1986.
[9] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," Proc. 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '08, pp.143–163, Springer-Verlag, 2008.
[10] E.J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," 2010 IEEE Symposium on Security and Privacy, pp.317–331, May 2010.
[11] S. Fortunato, "Community detection in graphs," Physics reports, vol.486, no.3-5, pp.75–174, 2010.
[12] M. Rosvall and C.T. Bergstrom, "Maps of random walks on complex networks reveal community structure," Proc. National Academy of Sciences, vol.105, no.4, pp.1118–1123, 2008.
[13] A. Clauset, M.E.J. Newman, and C. Moore, "Finding community structure in very large networks," Phys. Rev. E., vol.70, no.6, p.066111, 2004.
[14] F. Bellard, "QEMU." http://www.qemu.org/.
[15] J. Zeng, Y. Fu, and Z. Lin, "PEMU: A pin highly compatible out-of-vm dynamic binary instrumentation framework," Proc. 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE'15, New York, NY, USA, pp.147–160, ACM, 2015.
[16] Oracle, "Oracle vm virtualbox." https://www.virtualbox.org.
[17] Xen Project, "The Xen Project." http://www.xenproject.org/.
[18] Intel Corporation, "Pin - a dynamic binary instrumentation tool." http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[19] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," SIGPLAN Not., vol.42, no.6, pp.89–100, June 2007.
[20] L. Cavallaro, P. Saxena, and R. Sekar, "Anti-taint-analysis: Practical evasion techniques against information flow based malware defense," Secure Systems Lab at Stony Brook University, Tech. Rep., pp.1–18, 2007.
[21] D.J. Watts, "The "New" science of networks," Annu. Rev. Sociol., vol.30, no.1, pp.243–270, 2004.
[22] P. Pons and M. Latapy, Computing Communities in Large Networks Using Random Walks, pp.284–293, Springer, 2005.
[23] G. Csardi and T. Nepusz, "The igraph software package for complex network research," InterJournal, Complex Systems, vol.1695, no.5, pp.1–9, 2006.
[24] T. igraph core team, "igraph library api documentation." http://igraph.org/python/doc/python-igraph.pdf.
[25] J. Daemen and V. Rijmen, The design of Rijndael: AES-the advanced encryption standard, Springer Science & Business Media, 2013.
[26] G. Paul and S. Maitra, RC4 Stream Cipher and Its Variants, 1st ed., CRC Press, Inc., Boca Raton, FL, USA, 2011.
[27] The igraph core team, "Python-igraph manual." http://igraph.org/python/doc/igraph.Graph-class.html.
[28] R. Rugina and M.C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," ACM Trans. Program. Lang. Syst. (TOPLAS), vol.27, no.2, pp.185–235, 2005.
[29] "Beecrypt." http://beecrypt.sourceforge.net/.
[30] "Brian Gladman's Home Page." http://www.gladman.me.uk/.
[31] "Crypto++ Library 5.6.2." http://www.cryptopp.com/.
[32] O.S. Foundation, "OpenSSL: The Open Source toolkit for SSL/TLS." https://www.openssl.org/.
[33] V. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of community hierarchies in large network," J. Stat. Mech. P, vol.2008, no.10, 2008.
[34] J. Reichardt and S. Bornholdt, "Statistical mechanics of community detection," Phys. Rev. E., vol.74, no.1, p.016110, 2006.
[35] Microsoft, "Windows 7 api list." https://msdn.microsoft.com/en-us/library/windows/desktop/hh920509(v=vs.85).aspx.
[36] StarForce Technologies Ltd., "Aspack software." http://www.aspack.com/asprotect.html.
[37] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," Proc. 14th International Conference on Recent Advances in Intrusion Detection, RAID'11, vol.6961, pp.41–60, Springer-Verlag, 2011.
[38] J. Calvet, J.M. Fernandez, and J.-Y. Marion, "Aligot: Cryptographic function identification in obfuscated binary programs," Proc. 2012 ACM Conference on Computer and Communications Security, CCS '12, New York, NY, USA, pp.169–182, ACM, 2012.
[39] M.G. Kang, S. Mccamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation," Proc. Network and Distributed System Security Symposium, NDSS'11, 2011.
[40] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS '09, pp.73–85, New York, NY, USA, 2009.
[41] Y. Dodis and J. Katz, "Chosen-ciphertext security of multiple encryption," Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, vol.3378, pp.188–209, Springer, 2005.
[42] M. Green, "Multiple encryption – A Few Thoughts on Cryptographic Engineering." https://blog.cryptographyengineering.com/2012/02/02/multiple-encryption/, 2012.

Appendix A: Approach to Encryption Function

For encryption functions, FCReducer stores the executed instructions of a given program until an encrypted message is sent to the server. At this time, FCReducer marks the sent message with `ID 0`, and the function that writes the encrypt message is considered as the root function. It then conducts procedure *GenerateGraphForEnc* shown in Fig. A· 1 to generate the graph, taking as input each of the stored instructions in descending order of time. After that, FCReducer applies community detection and the I/O-size heuristic to the generated graph.

We summarize the results of InfoMap. Without the I/O-size heuristic, the RAF values against Alina, Pony, and Zeus were 1.10, 1.20, and 0.35 with no false negative cases.

```
1:  procedure GenerateGraphForEnc(I_i)
2:      if I_i = RET then
3:          push id on stack
4:          global_id ← global_id + 1
5:          id ← global_id
6:          add V_id to G
7:      else if I_i = CALL then
8:          pop id from stack
9:      else if I_i = READ addr val then
10:         M_addr ← id
11:     else if (I_i = WRITE addr val) and (M_addr is tainted) then
12:         k ← M_addr
13:         add E_{k,id} to G
14:     end if
15: end procedure
```

Fig. A· 1 A procedure for generating a graph for encryption functions.

On the other hand, Grum did not encrypt a message sent to our C&C server. Against only Crypto++ and OpenSSL (AES) out of the 12 samples, InfoMap caused false negative cases. The average RAF value for all the samples except for Grum was 0.53. With the I/O-size heuristic, the RAF values against Alina, Pony, and Zeus were 0.31, 0.34, and 0.35 with no false negative cases. The average RAF value for all the samples except for Grum was 0.23. The results for encryption functions did not greatly differ from those for decryption function, and thus we could tell that FCReducer could be also applicable to encryption functions.

Appendix B: Consideration of Multiple Encryption

There is *multiple encryption*, which encrypts a plain-text message using multiple instantiations (e.g., functions and tools) of a cryptographic algorithm (or algorithms) [41], [42]. In this appendix, we consider whether or not FCReducer can be effective for multiple encryption in case malware specimens using it confront analysts.

Suppose that a function "*multiple_decrypt(key1, key2, cipher_text, outdata)*" is given and that *success* means to locate this function. In this case, FCReducer will be effective because `multiple_decrypt` is equivalent to (single) decryption functions like RC4(key, cipher_text, outdata), in terms of the usage of functions. As another example, given *decrypt1(key1, cipher_text, outdata1)* and *decrypt2(key2, outdata1, outdata2)*, FCReducer will be effective to locate `decrypt1`. However, locating `decrypt2` could be difficult for FCReducer, in particular in cases where functions are called between `decrypt1` and `decrypt2`. Those called functions could work to raise a possibility that `decrypt1` and `decrypt2` are contained in different communities (i.e., `decrypt2` is excluded from the root community). Including this case, we plan to check whether or not FCReducer is effective for multiple encryption in our future work.

Ryoya Furukawa received his B.E. and M.E. degrees from Kobe University, Japan, in 2015 and 2017, respectively. He currently works at PwC Cyber Services LLC and is also a Ph.D. student in Kobe University. His current research interests include malware analysis.

Ryoichi Isawa received his B.E. and M.E. degrees from the University of Tokushima, Japan, in 2004 and 2006, respectively. He received his Ph.D. degree from Kobe University, Japan, in 2012. He is currently a senior researcher at the National Institute of Information and Communications Technology (NICT), Japan. His current research interests include malware analysis, network security, and information security. He is a member of the IEEE.

Masakatu Morii received his B.E. degree in electrical engineering and his M.E. degree in electronics engineering from Saga University, Saga, Japan, and his D.E. degree in communication engineering from Osaka University, Osaka, Japan in 1983, 1985, and 1989, respectively. From 1989 to 1990, he was an Instructor in the Department of Electronics and Information Science, Kyoto Institute of Technology, Japan. From 1990 to 1995, he was an Associate Professor in the Department of Computer Science, Faculty of Engineering at Ehime University, Japan. From 1995 to 2005, he was a Professor in the Department of Intelligent Systems and Information Science, Faculty of Engineering, at the University of Tokushima, Japan. Since 2005, he has been a Professor in the Department of Electrical and Electronics Engineering, Faculty of Engineering, at Kobe University, Japan. His research interests include error correcting codes, cryptography, discrete mathematics, computer networks, and information security. He is a member of the IEEE.

Daisuke Inoue received his B.E. and M.E. degrees in electrical and computer engineering and Ph.D. in engineering from Yokohama National University in 1998, 2000 and 2003, respectively. He joined Communications Research Laboratory (CRL), Japan, in 2003. CRL was relaunched as National Institute of Information and Communications Technology (NICT) in 2004, where he is currently Director of the Cybersecurity Laboratory. He has received a number of awards including the commendation for science and technology by the minister of MEXT, Japan, in 2009, the Good Design Award 2013, the Asia-Pacific Information Security Leadership Achievements 2014, and the award for contribution to Industry-Academia-Government Collaboration by the minister of MIC, Japan, in 2016.

Koji Nakao manages research activities for network security technologies in NICT (National Institute of Information and Communications Technology). He received his B.E. degree of Mathematics from Waseda University (Japan) in 1979. Since joining KDDI in 1979, he has been engaged in the research on communication protocol, and information security technology for telecommunications in KDDI laboratory. After 2003, he has moved to KDDI head office to construct and manage its security systems. In 2004, he has started to additionally work for NICT. He received the IPSJ Research Award in 1992, METI Ministry Award and KPMG Security Award in 2006, and Contribution Award (Japan ITU), NICT Research Award, Best Paper Award (JWIS) and MIC Bureau Award in 2007 and The Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology (Prizes for Science and Technology: Research Category) in 2009. He is a fellow of IEICE and a member of IPSJ. He has also been a part-time instructor in Waseda University and Nagoya University. In April 2017 he was appointed as an official security adviser of NISC under the cabinet of the Japanese government.