Automatic Extraction of Secrets from Malware

Ziming Zhao, Gail-Joon Ahn and Hongxin Hu
Laboratory of Security Engineering for Future Computing (SEFCOM)
Arizona State University, Tempe, AZ 85281, USA
{zmzhao, gahn, hxhu}@asu.edu

Abstract—As promising results have been obtained in defeating code obfuscation techniques, malware authors have adopted protection approaches to hide malware-related data from analysis. Consequently, the discovery of internal ciphertext data in malware is now critical for malware forensics and cyber-crime analysis. In this paper, we present a novel approach to automatically extract secrets from malware. Our approach identifies and extracts binary code relevant to secret hiding behaviors. Then, we relocate and reuse the extracted binary code in a self-contained fashion to reveal hidden information. We demonstrate the feasibility of our approach through a proof-of-concept prototype called ASES (Automatic and Systematic Extraction of Secrets) along with experimental results.

I. INTRODUCTION

Malicious codes have been tremendously evolved with various intents and techniques. With the boom in Internet and electronic commerce, we witnessed the switch of malware authors’ interests from boasts to economic benefits. Along with this paradigm shift, malware authors have been developing sophisticated methodologies to undermine the analysis of captured malware and compromised systems. In the early days of malicious code analysis, it was relatively easy to reveal the malicious behaviors of malware. However, the code analysis is no longer straightforward since we must first defeat heavily obfuscated code and encrypted data.

One way to study the behavior of malware is to comprehend its binary code because corresponding source code is not available in the most cases. Understanding malware starts with disassembly, which recovers human-readable symbolic representation of malware from its binary form. Obfuscation techniques are prevalent in both benign and malicious programs to prevent malware from being disassembled and understood by analysts. Disassembly desynchronization [28] has profound impacts on linear sweep disassembly, while dynamically computed target addresses such as indirect calls or jumps may thwart recursive traversal disassembly. Static approaches [19] with the help of control flow graphs and statistical methods can accurately identify a large fraction of instructions obfuscated by aforementioned techniques.

Recent obfuscation techniques such as packing [17] and emulation [6] technologies have been widely adopted. Packing disguises malicious code as non-executable data in malware files and transforms it back to executable code at runtime. Emulation converts binary code to some bytecode and attaches both bytecode and its corresponding emulator to the malware. Static [14] and dynamic [17] approaches have been proposed to automatically unpack such packed malwares. Also, dynamic data-flow [29] and taint analysis [24] have been presented to generate control flow graphs for emulator-based malware analysis.

As code obfuscation techniques are not sufficient for malware authors to disguise their motives, they attempt to make use of other protection approaches, which are normally used to protect data rather than code [12], to hide their secrets from analysts. Therefore, prior code extraction techniques may work smoothly for packing and emulation, but they are ineffective to accommodate the new trend of malware evolution.

Given the significance of this problem, another research branch in malware analysis is concerned about the extraction of malware-related data [11]. Data extraction plays an important role for malware forensics since investigators could use the wealth of information that can be retrieved from malware to identify suspicious activities. Previous solutions for data extraction from malware were highly manual [4]. A major drawback of these attempts is that human knowledge of the binary code location and behavior is required.

Recently, Caballero et al. [10] and Kolbitsch et al. [18] attempted to automatically extract interesting binary code pieces and reuse them for security analysts. One application of their binary code reuse technique is to reveal some malware-related data. The method presented by Caballero et al. [10] identifies the prototype of binary code fragments that correspond to source code level functions. They used dynamic analysis to extract the actual parameters for code fragments and inferred the formal parameters by multiple runs. Kolbitsch et al. [18] proposed a method, which also uses the combination of static and dynamic analyses, to extract the complete algorithm related to a certain activity that may consist of multiple binary level functions. They relocated and executed the extracted algorithm for domain name generation and botnet protocol infiltration.

In this paper, we classify malware-related data as internal or external plaintext or ciphertext in terms of its origin and form. We observe that existing manual and automatic solutions pay more attention to external ciphertext data, such as botnet command and control (C&C) protocol. In this
In order to automatically extract secrets embedded in malware, we propose a novel technique combining static analysis and code execution together to reveal valuable hidden information in binary. Our approach takes a malicious binary code as input, then automatically discovers the plaintext value of hidden secrets in this binary. Our method does not require prior knowledge of (i) the existence and location of secrets and (ii) protection algorithms used in given malicious code. The key idea behind our solution is that the malicious code itself has to decode its embedded secrets before using them if the hidden secrets exist. Hence, we can identify the binary variables which have to be used as plaintext, infer the existence of secrets and locate the decoding algorithm. Then, we use binary code relocation technique to inject identified code into another process address. By executing this code in a protected process, we can reveal the hidden information in malware.

The differences and advantages of our approach over existing proprietary algorithm extraction solutions are as follows: 1) Caballero et al. [10] used a bottom-up approach to identify the interfaces of all binary functions in malware. Instead, we use a top-down approach to merely extract interesting binary code which is responsible for the behavior we are concerned about; 2) instead of extracting the prototype of binary functions, we recover the actual parameters and runtime context of each invocation of binary function. Therefore, our binary code reuse does not involve human analysts to provide parameters for code execution; and 3) our approach performs static analysis to identify interesting code while Kolbitsch et al. [18] proposed dynamic analysis to pinpoint interesting binary. However, dynamic analysis could not cover all the control paths and is vulnerable to malware defense techniques [16].

In addition, we demonstrate the feasibility and applicability of our solution by implementing a proof-of-concept prototype called ASES (Automatic and Systematic Extraction of Secrets, /əˈsiːz/), which is an IDA Pro [3] plugin to recover external modules and API names loaded by malware, along with our experimental results.

The rest of this paper is organized as follows. Section II presents the problem definition and overviews our approach and system architecture. Section III describes our decryptor identification method. Section IV presents our binary relocation algorithm. The evaluation of our solution is discussed in Section V followed by the related work in Section VI. Section VII discusses the future work and concludes the paper.

II. SYSTEM OVERVIEW

In this section, we first categorize malware-related data based on two different dimensions. We then overview the research issues in extracting internal ciphertext data in binary. Also, we outline our approach with a motivating example.

Figure 1: Malware-related Data

A. Malware-related Data

Figure 1 shows malware-related entities and data. The malware itself, the host victim machine and C&C server are the three major players in a typical cyber-crime setup. In terms of origin, malware-related data could be categorized into external data and internal data. External data, such as local host information, may be retrieved by malware on the fly or obtained from network communication with C&C servers. For example, a malware may query the hard disk serial number of its host machine, transform it into an encrypted form with its proprietary algorithm and send it to its C&C server. Then the C&C server may send back an encrypted spam template so that the malware could decrypt it and send spam in the future. Internal data is embedded in malware in a static way. Although data segment is designed to store global and static variables initialized by programmers, programmers may choose to store data in any other customized segments.

In terms of form, malware-related data may be divided into plaintext and ciphertext data. In Figure 1, the solid rectangle and arrow indicate encrypted data and secured communication while the dashed rectangle and arrow represent plaintext data and unprotected communication. In practice, external data is observed in form of both plaintext and ciphertext. Local host information is normally obtained by operating system APIs or interrupts. This information is not encrypted since confidentiality is not a serious issue between a host machine (OS and CPU) and its applications. On the contrary, C&C messages between the server and malware are usually encrypted to prevent analysts from comprehending them easily.

Some internal data exists as plaintext which may be in the form of text or programmer-defined structure. The internal plaintext data may be introduced in compiling or linking stage by malware programmers as global or static variables. One example of data generated by a compiler is a string. For example, it would be ‘This Program Cannot Be Run in DOS Mode’ in PE header. If a windows executable is invoked in real-mode, a stub will display this message and
make this program exit. The PE header also includes other plaintext data, such as import address table which indicates external libraries used by this program. Internal plaintext data could be easily recovered by tools such as GNU strings which print printable character sequences in any file format. To recover other plaintext data which is not in the form of text, the knowledge of data structure is also needed. External plaintext data could be captured by monitoring the host behavior [33] or network behavior [25] of malware.

Compared with plaintext data extraction, ciphertext data receives much more attention in malware analysis and forensics because valuable information is more carefully protected by malware programmers. In practice, most existing solutions are performed to find, deobfuscate and understand the transformation code chunks for deciphering data by human analysts in a manual way [4]. Recently, Caballero et al. [10] and Kolbitsch et al. [18] attempted to automatically reuse binary in malware to decrypt and rewrite botnet protocols. Their approaches deal with external ciphertext data, but neglect the importance of internal ciphertext data.

B. Problem Definition

In a malware executable, internal sensitive data include module names, API names, URLs, email addresses, and any other meaningful strings and structures which may lead to the disclosure of the malware behaviors or be used for forensic analysis. External module and API names give a static high-level outline of malware behaviors. URLs and email addresses could help forensic analysts trace adversaries behind the scene. Due to the importance of such information residing in an executable, malware authors use data protection mechanisms ranging from simple XORs to sophisticated cryptographic algorithms to hide them via internal ciphertext data from security analysts [26]. Based on our observation, we believe internal ciphertext data is as important as external ciphertext data in cyber-crime analysis. We denote internal ciphertext data, protection mechanisms, and the code used to decrypt ciphers as secret, encryption and decryptor, respectively.

Encrypting secrets has been widely adopted in constructing real world malicious code by malware authors [5], [26]. Although revealing these secrets by manual step-by-step debugging or execution instrumentation is possible, this process is very difficult for extracting large-scale secret due to the following reasons: first, the process is tedious. In most cases, extracting secret is to manually repeat the similar analysis process. Valuable information may be overlooked due to potential lack of consistent attention; second, the process involved with both static and dynamic methods is time-intensive. It usually takes 2 to 5 minutes for sandbox systems, such as Anubis [1], to generate analysis reports; and thirdly, the process may heavily rely on computation and storage if instrumentation techniques are used.

Therefore, extracting secrets in large-scale is not practical by adopting such primitive methods. Furthermore, there exist many dynamic defense techniques [13], [15] to prevent monitoring or instrumentation. Given a malicious binary code, our goal is to recover plaintext data from encrypted data in the code. Furthermore, the process of extraction should not require human involvement and to access source code or symbol information.

C. Motivating Example

Among sensitive data, external dynamically-linked libraries (DLL) and API names are of great importance owing to the fact that they give us an outline of what kind of actions a program performs. For example, a program which loads crypt32.dll tends to make some cryptographic operations, such as encryption and hashing. A program loading ws2_32.dll may imply some low level network actions, as we discussed earlier. API names give a more detailed static profile of the behavior of a program. Invocation of CryptHashData implies some data is hashed during the execution of this program. Also, correct identification of WSAConeet and its parameters could tell analysts which network protocol is being used by the program.

In implicit DLL loading, import lookup table (ILT) and import address table (IAT) work together to provide loader with information to make connection with external API in Windows PE format [23]. ILT stores external API information, such as API names and ordinal numbers in DLL. Then, during binding, the entries in the IAT are overwritten with the actual addresses of the symbols that are being imported. Thereafter, the program could call external API by jumping to its address indicated in IAT.

Because API information is stored in the format of plaintext, existing tools, such as IDA Pro and Dumpbin [2], can display the names of imported APIs by analyzing ILT if they are implicitly loaded. Malware authors usually take control over this loading procedure by decrypting ciphered API information and explicitly loading external API at runtime. This process, named run-time dynamic linking under Microsoft platform, uses the LoadLibrary API to load DLLs. The GetProcAddress API is used to look up exported symbols by name, and FreeLibrary is invoked to unload DLLs. Analogous APIs—such as dlopen, dlsym, and dlclose—also exist in the POSIX standard.
subroutine IATBuilder sub_40C7F3
...
1 lea eax, [ebp-400h] ;
2 push 0041F8DC ;xreary32.qyy
3 push eax ;
4 call sub_40E712 ;Decryptor
5 pop ecx ;
6 push eax ;
7 call GetProcAddress ;
8 mov edi, eax ;
9 lea eax, [ebp-400h] ;
10 push 00421634h ;FrgReebeZbqr
11 push eax ;
12 call sub_40E712h ;
13 pop ecx ;
14 push eax ;
15 push edi ;
16 call GetProcAddress ;
17 mov [4C0604], eax ;
...

Figure 3: Subroutine sub_40C7F3

As a motivating example, Figure 2 shows a simplified call graph and data section of a sample virus. The solid line with arrow indicates the calling operation. The dashed line with single-headed arrow from subroutines to data indicates the read operations. And, the dashed line with double-headed arrow indicates read and write operations from the subroutine to the data location. We use the same naming convention as IDA Pro for subroutine and data. sub_x is the name given to the subroutine at address x. dword_x is a 32-bit data at location x. dd_x denotes a byte string located at x, whose length is unknown.

In this motivating example, before performing any malicious behavior, the virus builds a customized IAT by calling sub_40C7F3, which first decrypts DLL and API names, gets external API addresses by invoking GetProcAddress, LoadLibrary and GetProcAddress, then writes them in the memory ranging from 0x4C0288 to 0x4C09A0. This memory range (0x4C0288 to 0x4C09A0) plays the same role as an import address table. Any future calls to external API will be redirected through this table. To achieve this goal, sub_40C7F3 calls sub_40E712 recursively to decrypt data stored from 0x42140C to 0x42192D, which invokes sub_40E697 to mutate the keys. Therefore, sub_40E712 is the decryptor and we call sub_40C7F3 as IATBuilder.

Figure 3 shows the decryption and customized IAT related code in sub_40C7F3. Instruction 1 loads the effective address of a local variable. EBP-400h indicates this variable is not allocated by sub_40C7F3 but by its ancestor in call graph. Through manual analysis, we notice this address is used to store the decrypted strings. Instruction 2 pushes the address of an encrypted string xreary32.qyy on stack. Instruction 3 pushes the address for decrypted string on stack. Then, Instruction 4 calls subroutine sub_40E712, which is the decryptor. After sub_40E712 returns, the decrypted string, which is kernel32.dll in this case, is stored at address EBP-400h. Instruction 6 pushes this address on stack again for Instruction 7 to get the module handle by calling Windows API GetProcAddress. Instruction 8 stores this handle to EDI. Instructions 9 through 13 do the same actions as Instructions 1 through 5 do. The only difference is the input for sub_40E712 is changed to FrgReebeZbqr that is an encrypted API name. Instruction 14 pushes the name of API on stack, which is GetProcAddress returned from Instruction 12. Instruction 16 gets the actual address of API SetErrorMode in EAX. Instruction 17 stores this address in an entry 0x4C0604 of the customized IAT table.

Figure 4 shows partial code of decryptor sub_40E712. Instruction 3 reserves 38h bytes on stack for local variables of this procedure. Instructions 4 and 6 move the addresses of global variables (421ADCh, 421AC0h) to ESI for the further use. Our analysis on this sample virus identifies that these two global variables are keys for decryption. Instructions 5 and 6 access local variables of this procedure and Instruction 8 calls another subroutine.

D. Approach and System Architecture

In this section, we articulate the system architecture of our proof-of-concept prototype called ASES, which automatically recovers external API information. ASES is a C/C++ program on top of IDA Pro, which includes modules to perform static analysis, binary relocation, and code execution. As an IDA Pro Plugin, ASES leverages the features of IDA Pro including disassembly, control flow graph, cross reference, and activation record analysis.

ASES consists of four different phases: pre-processing, decryptor identification, decryptor relocation and decryptor reuse. Figure 5 shows the four phases of ASES and the interactions between each module. In pre-processing phase, malicious code is disassembled by recursive descent disassembly. Section information and plain import table are read from PE header and stored in local database for further analysis. All functions and function-like code are identified, and function call relationships are stored in a call graph as well. In each function, the stack layout is also recovered: the starting address and size of each input parameter are calculated and any accessed local variables are identified. A control flow graph is also generated for
Encryption detection is to identify decryptor. There are two major components are recorded in database. Furthermore, both code and data cross-references each function. Strings longer than 4 bytes in binary are all recorded. If a CALL instruction is identified, JMP recursively copies callee to its space. If a JMP instruction is identified, AES checks whether the jumped location is out of start and end. If that is the case, AES copies the jumped code to its space. Since it is impossible to predict the number and size of functions to be copied, AES reserves a large amount of space to hold the code. Figure 6 shows an example code to reserve space in .data and .text sections under Windows Visual Studio environment by using pragma directive. Statements 3 and 4 in Figure 6 define two constants to hold 20 regions with 200 bytes for code. Statements 7 and 8 reserve space in .text section and define it as g_Code. After recursive function relocation, AES runs its variable relocation engine to relocate binary variables. Similar to the way to reserve code, AES reserves space in .data section for variables that need to be relocated.

In the final code execution phase, AES jumps to the relocated start by calling g_Code[0] and waits for its return. Since code is recursively copied and redirected by function relocation engine, the execution context, epilogue and decryptor itself are all executed. After code execution, AES outputs the value of all variables in this code.

### III. Decryptor Identification

The goal of decryptor identification is to find the code that is responsible for internal cipher decoding. Note that decryptor does not necessarily correspond to one high-level language function. It may comprises several high-level language functions and most importantly it includes the instructions for passing function parameters. We first define hot, start and end instructions for decryptor identification. In light of these definitions, the problem of identifying decryptor, its context and epilogue is transformed into identifying hot instruction and its corresponding start and end instructions.

**Hot Instruction:** In a program \( P \), there exist some instructions that certain data must be plaintext when these instructions are ready to be executed. For instance, the Windows API GetProcAddress, whose prototype is shown in Figure 7, takes the name of function, which the caller wants to use, as the second parameter. Therefore, when GetProcAddress is called in a program, the second parameter must point to some plaintext API names, otherwise the call will fail. We call an instruction hot instruction, if its parameters are encrypted and decrypted at runtime. Instruction 7 in Figure 3 is an example of hot instruction. When the program counter in CPU reaches this location at runtime, the string located by EAX at Instruction 6 must be plaintext.

**Start Instruction:** We define start instruction as the instruction from where relocation should start. For a given hot instruction, there are many instructions that we could start relocation from and end up with the same outcome. We define supremum start instruction as a start instruction that has the shortest path to the corresponding hot instruction.

![Figure 5: ASES System Architecture](image)

```c
1 const int VAR_SIZE = 100;
2 const int VAR_NUM = 20;
3 const int CODE_SIZE = 200;
4 const int CODE_NUMBER = 20;
5 #pragma data_seg(.data)
6 char g_Var[VAR_NUM][VAR_SIZE] = {{0}};
7 #pragma code_seg(.text)
8 char g_Code[CODE_NUMBER][CODE_SIZE] = {{0}};
```

![Figure 6: Space Reservation](image)

```
2 const int VAR_NUM = 20;
3 const int CODE_SIZE = 200;
4 const int CODE_NUMBER = 20;
5 #pragma data_seg(.data)
6 char g_Var[VAR_NUM][VAR_SIZE] = {{0}};
7 #pragma code_seg(.text)
8 char g_Code[CODE_NUMBER][CODE_SIZE] = {{0}};
```

![Figure 7: Prototype of GetProcAddress](image)
End Instruction: End instruction is the last instruction we need to relocate. An end instruction is not necessarily the immediate precedent instruction of corresponding hot instruction but the instruction that can provide an exact stack match from the start instruction. For example, Instruction 5, instead of Instruction 6 in Figure 3, is the end instruction for a hot instruction 7. Failure to correctly identify end instruction may result in program crash due to stack imbalance.

A. Identifying Hot Instructions

The first step to identify hot instructions is to find the functions whose parameters have to be plaintext. In this work, we focus on identifying functions whose parameters are strings and ignore parameters in other structured forms. In standard C library, string parameters are passed by char *. However, some char * parameters are not strings. It could also be a pointer to memory buffer. Therefore, we use the combination of formal parameter type and name to infer whether this parameter is a plaintext string. For each C library function, we check whether it has any char * or const char * in its prototype. Then, we use regular expression to check whether the name of this parameter has any substring such as name and path in it. If it is true, that means this function has at least one string parameter. It is much easier to identify string parameters for Windows library functions. Because Windows libraries use PVOID to denote the address of memory buffer and LPSTR, LPCSTR, LPWSTR, LPCTSTR, and LPCTSTR to represent string parameters. Windows function prototype also provides information whether this parameter is for input or output by defining __in and __out. This information is also utilized since we are only concerned about input parameters.

After we identify functions that take string parameters with the above approach, we check whether the given malware invokes any of these functions with the help of disassembly and cross-reference information discovered by IDA Pro. Then, we test whether the parameter of this function is encrypted. Although entropy testing [21] is successful in identifying symmetric and asymmetric key encryption, it is incapable of identifying substitution encryption for which the entropy has no significant change. In addition, to perform entropy analysis, a relatively large number of samples should be collected. Besides, the encrypted data is scattered in malware image and its amount is unknown. To address this challenge, we present an effective method to simply call the candidate instruction with its parameters. For Instruction 7 in Figure 3, our solution invokes GetModuleHandle(‘xreary32.qyy’). Because file xreary32.qyy does not exist, GetModuleHandle returns false. We consider the parameter is encrypted and Instruction 7 is a hot instruction.

B. Identifying Supremum Start Instructions

For every hot instruction, the entry point of the malicious code could be viewed as its start instruction. However, if we relocate code from the entry point to end instruction, many instructions which do not belong to decryptor will be also relocated. This leads us to consider two challenges: First, unnecessary computational cost can be brought into since unnecessary code may be executed. Second, the executed unnecessary code may be maliciously performing some network attacks and information theft.

Our goal is to identify all instructions relevant to the hot instruction and to exclude irrelevant instructions at the same time. The core idea is to use backward slicing [8], [34] on hot instruction and its parameters to determine all previous instructions which affect the parameters. The final challenge is that given a hot instruction we need to identify all of its parameters. Because there is no variable in binary, this process is not as obvious as high-level language counterpart. Inspired by Clemens et al. [18], we use the prototype information retrieved from library header files to infer the binary parameters and perform a backward slicing on each parameter. Finally, among the instructions identified by backward slicing the one that precedes all others is recognized as the supremum start instruction.

C. Identifying End Instructions

Different calling conventions in C/C++ result in different responsibility for stack cleanup. A __stdcall function cleans the stack by itself, while a __cdecl function needs the help from its caller to restore the stack. Hence, for binary code that corresponds to a __stdcall function, the end instruction is the same one as the hot instruction. However, the end instruction for a __cdecl function is not so clear to determine. If we simply relocate a __cdecl function binary without relocating its caller’s stack cleanup code, the size of data pushed on stack before the function is called and the size of data popped from stack after the function is executed do not match each other. Such a stack imbalance will cause the crash of the hosting process.

The basic idea to determine an end instruction is to emulate the stack size change from a start instruction. Note that emulation of other behaviors in binary function is not necessary. Because only the size of stack is used for determining the end instruction, the actual value on the stack is irrelevant for the analysis. Our approach initializes a relative stack pointer with 0 and starts analysis from the start instruction to analyze every stack operation instruction. Stack operation instructions include PUSH, POP and arithmetic operations on ESP, such as SUB ESP, 44. If a PUSH or SUB on ESP is encountered, we increase relative stack pointer by the size of data placed on stack accordingly. On the other hand, if a POP or ADD on ESP is encountered, we decrease relative stack pointer accordingly. If control transfer instructions, such as CALL, are faced, we jump
into the callee to record the stack change recursively. The analysis stops only when the hot instruction is passed and relative stack pointer is zero. The instruction where the analysis stops is identified as the end instruction.

IV. DECRYPTOR RELOCATION AND REUSE

In this section, we discuss how to relocate and reuse decryptor identified from previous section. The relocation process makes sure that the relocated code is in a self-contained fashion. Since we do not assume the existence of symbol information and relocation table, the approach we propose is totally based on binary code itself without having any meta-data. Binary relocation without meta-data faces the following challenges: 1) there does not exist information for which references should be relocated; 2) variable type information is not available [30]. It may not be possible to infer high level variable types from binary code; and 3) variable size information is not available. It is not even clear how many bytes of memory the variables hold. We address these challenges by categorizing data reference into different types of variables.

We categorize variables in binary into register variable, stack variable, ancestor stack variable, and global variable. We use a C program and its corresponding x86 assembly to illustrate our binary relocation approach. Figure 8 (a) gives the C program that uses global and local variables. Figures 8 (b) - (c) show the corresponding x86 program of function func1, func2 and main. 1 We do not need to allocate memory space for heap variables, because those variables will be allocated at runtime by relocated code itself.

Register Variable: In a piece of disassembly code, registers are considered as register variables whose values are stored in physical CPU registers. Register variables do not take any memory space in runtime address space (RAS) and therefore do not need relocation. Instruction 1 in Figure 8 (b) is an example of register variable which is stored in EBP. Register variables can be used to access or represent the address of other kinds of variables. Instruction 4 in Figure 8 (b) is an example in this case. [EBP+8] denotes the variable at the address that is 8-byte higher than the stack frame pointer.

Stack Variable: Stack variables are those that are allocated by stack operation instructions at runtime. Instruction 1 SUB ESP, 38h in Figure 4 allocates 38h bytes for local variables. Instruction 3 in Figure 8 (b) is another example to allocate local variable, although it does not look like one at a careless glance. This instruction allocates 4 bytes for the corresponding C program integer b1. If a function is fully relocated, its local variables are stack variables. sub_40E712 shown in Figure 4 is an example of fully relocated binary function, so we do not need to allocate memory space for its local variables.

Ancestor Stack Variable: Ancestor stack variables are those that are stack variables originally, but their residing functions will not be fully relocated. This happens when the start instruction is located after the entry point of this function. We call them ancestor stack variables, because they are originally allocated at runtime as well by ancestor callers of relocated code identified in our static analysis. sub_40C7F3 shown in Figure 3 is an example of partially relocated procedure. The code before Instruction 1 and its ancestor callers are not relocated in new runtime address space. Therefore, those instructions which are responsible for allocating local variables are stripped from the relocated code. Instruction 1 LEA EAX, [EBP-400h] accesses such a variable. We allocate global space in .data section for ancestor stack variables.

We further illustrate the difference between stack variable and ancestor stack variable by an example code shown in Figure 8. In this code, two global variables are defined. main calls func2 that calls func1. As shown in Figure 9, g_int1 and g_int2 reside in global data section. bm, b2 and b1 reside in corresponding activation record (AR) of main, func2 and func1. If Instruction 4 in Figure 8 (d) is recognized as the start instruction, allocation for bm which is a stack variable in original runtime address space (RAS_old) is missed. In this case, bm is an ancestor stack variable. Therefore, in new runtime address space (RAS_new), bm is allocated in global data section instead. If Instruction 4 in Figure 8 (c) is identified as the start instruction, both bm and

Figure 8: A C Program That Uses Global/Local Variables and Its x86 Program in Intel Syntax
b2 are moved to global data section in RAS_new as shown in Figure 9 (b). Figure 9 (c) shows variable relocation, if Instruction 4 in Figure 8 (b) is identified as the start of relocation.

We treat global variables the similar way as does ancestor stack variables. Reserved .data section space is allocated for global variables. In addition, the values of initialized global variables are copied to RAS_new. Instruction 10 PUSH 00421634h in Figure 3 is an example of access global variable. The value FrgRebeeZbqr of variable at 00421634h is copied to its relocated variable in RAS_new.

In order to reuse the relocated code, we wrap the relocated code body in an assembly block and place this block in a C function with a prototype int runCodes(void). After code relocation, ASES calls runCodes() to give the control to the relocated code.

V. EVALUATION

To evaluate the effectiveness of our approach, we tested Virut.d [7] with ASES. Virut.d is a polymorphic, memory-resident Windows 32-bit malware, which has entry point obscuring capabilities. Upon running, Virut.d injects winlogon.exe and infects files on local and shared drives. Virut.d has good camouflage by using process injection, but it does not adopt any rootkit techniques. Virut.d contains an IRC-based backdoor which provides unauthorized access to infected computers. The snapshot of the interface of ASES is given in Figure 10. Analysts run ASES by clicking the specific plugin button in IDA Pro, and ASES yields the results in output window.

We conduct experiments on a machine with Intel Core2 Duo CPU 3.16 GHz 3.25 GB RAM running Windows XP Professional SP3 and IDA Pro 5.6.0.931. We use Windows API GetTickCount to measure the performance of our prototype. Without ASES, IDA Pro identified 135 APIs imported from seven different DLLs by reading Virut.d’s PE header. Table I (a) shows these seven DLLs and one function from each DLL. These DLLs are typical dynamic link library files loaded by Windows applications, which handle memory management, input/output operations, interrupts, windows user interface, process status helper, web pages and network behaviors. Although this DLL information can imply some behaviors of this executable, this revealed information is limited because they are almost loaded by every Windows program nowadays.

Now, we describe the results ASES discovered from this malware:

Hot Instruction Identification. At the very first step of static analysis, ASES detected 94 suspicious function invocations, then narrowed down the number of hot instructions to 82. These invocations scattered from address 406BC3h to 40D688h, which imply that code in this section may be responsible for initialization.

Start/End Instruction Identification. ASES identified start/end instructions for all 82 hot instructions. We noticed that, for most cases, the code distance from the start instruction to the end instruction is the same. We suspect the malware author reused some source code, therefore the compiler generated the same binary code. However, we did observe several cases, in which the code distance from the start instruction to the end instruction is larger than normal. We manually checked these cases for the evaluation purpose and verified that ASES identified the correct instruction.

Binary Relocation. ASES relocated five variables including ancestor stack variables and global variables between
each start and end instruction pair. We manually checked the correctness of this step for the evaluation purpose, and we found the smallest variable is 4 bytes, while the largest is around 30 bytes. ASES relocated five functions as well. Note that if a function is called more than once, ASES relocates it for each invocation. This redundancy can be removed, if the information of relocated functions are stored. ASES gives control to relocated code after static analysis and binary relocation. In our experiments, this step was very effective and fast.

**Experimental Results**: ASES identified 82 API names which were encrypted in this malware. These APIs can be categorized into two types: 1) the DLL is loaded implicitly, but the API was not found in the import table. The first four APIs recovered in Table I (b) are in this case. The implicit load of kernel32.dll, wininet.dll, and shell32.dll were identified by IDA Pro. However, invocations to CreateMutexA, InternetCrackUrlA, HttpOpenRequestA, and SHChangeNotify were not disclosed. The disclosure of these APIs made clear that this malware has some http operations, which is useful to profile the behavior of this malware; 2) neither the DLL is loaded explicitly, nor the API. The rest of four APIs in Table I (b) are in this latter case. urlmon.dll contains functions used by Microsoft OLE, which allows an operation for embedding and linking to documents and other objects. advapi32.dll is a part of an advanced API service libraries supporting numerous APIs including many registry calls. dnsapi.dll is a module that contains functions used by the DNS Client API. mpr.dll contains functions used to handle communication between the Windows operating system and the installed network providers. Not only these DLLs look unfamiliar to security layman, we also identified some APIs which are undocumented, such as DnsFlushResolverCache, used to flush the DNS cache. Disclosure of these unfamiliar and undocumented DLLs and API helps outline the malware even further in a fine-grained manner.

**Performance**: In normal cases, IDA Pro takes less than one minute to process executables in our pre-processing phase. To reveal 82 external API names from Virut.d, ASES only took 672 milliseconds after pre-processing, supporting reasonable real-time responsiveness. Static analysis phase and binary relocation phase take around 45% of the process, respectively, and code execution takes around 10% of the entire process.

**VI. Related Work**

Binary code reuse is the most related research effort to our work. Lin et al. [20] proposed reuse-oriented trojan that extracts interfaces in benign programs and adds malicious functionalities on top of them. The idea is to reuse binary code and transform it into code with malicious purpose. Caballero et al. [10] performed the first systematic study of automatic binary code reuse and implemented BCR, which can extract binary functions and wrap it with a C interface. Kolbitsch et al. [18] developed INSPECTOR almost at the same time as BCR was introduced. Their approach was able to extract an entire functionality from binary.

Another work related to ours is dependency checking for binary code. Weiser first proposed program slicing [34] to check statement dependency of source code in high-level language. Most programming slicing techniques [32] focus on slicing high-level language programs where variables information and transfer of control is clear. Akgul et al. [1] presented how to perform dynamic slicing on assembly [8]. Sharif et al. presented abstract variable binding to reverse engineer emulators [29]. They used absolute memory addresses as variables, analyzed data flows among entire trace to determine variable binding, and used forward and backward slicing to identify dependent abstract variables.

There also exist some attempts to recover variable value for binary in a static way. Value-set analysis [9] recovers variable-like entities statically from executables and infers information about the content of these variables at every program point. However, it cannot infer the actual value of any variable but gives a possible value set. Symbolic execution [27] allows analysts to reason program behaviors by building a logic formula which represents a program execution. It may help reveal some valuable information in binary, but cannot directly infer the actual value neither.

**VII. Conclusion and Remarks**

In this paper, we classified malware-related data in terms of origin and form, and addressed the significance of internal ciphertext data for malware forensics. We have presented
a novel approach to automatically extract secrets from malware executables without any human involvement. The proposed approach consists of three major tasks to identify secret-related code in binary, relocate and reuse it without symbol information. We also developed a prototype system, ASES, to extract external API information from malware binaries. Our evaluation results on real world malware showed that ASES could identify sensitive data effectively and recover plaintext from executable systematically.

Although packing is beyond the scope of our approach, the techniques we proposed in this paper are general and can also be realized on top of dynamic malware analysis platforms, such as BitBlaze [30] and Anubis [1], which have the ability to unpack malware. Even though our techniques to identify and relocate binary code ensures that only decryptor is extracted from malware and executed in another address space, we could enhance the security of the host process with software-based fault isolation (SFI) [22], one-way isolation [31] and other similar techniques.

For our future work, we plan to address the challenges of other models of secret protection in malware along with rigorous testing of our approach with other types of malware.

ACKNOWLEDGMENT

This work was partially supported by the grants from National Science Foundation (NSF-IIS-0900970 and NSF-CNS-0831360) and Department of Energy (DE-SC0004308).

REFERENCES