HookFinder: Identifying and Understanding Malware Hooking Behaviors

Heng Yin Zhenkai Liang Dawn Song

October 17, 2007
CMU-CyLab-07-015

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

HookFinder: Identifying and Understanding Malware Hooking Behaviors

Heng Yin, Zhenkai Liang
Carnegie Mellon University
{*hyin, zliang*}*@ece.cmu.edu*

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

Abstract

Installing various hooks into the victim system is an important attacking strategy used by malware, including spyware, rootkits, stealth backdoors, and others. In order to evade detection, malware writers are exploring new hooking mechanisms. For example, a stealth kernel backdoor, *deepdoor*, has been demonstrated to successfully evade all existing hook detectors. Unfortunately, the state of the art of malware analysis is painstaking, mostly manual and error-prone. In this paper, we propose the first systematic approach to automatically identifying hooks and extracting the hook implanting mechanisms. We propose *fine-grained impact analysis*, as a unified approach to identify hooking behaviors of malicious code. Since it does not rely on any prior knowledge of hooking mechanisms, it can identify novel hooks. Moreover, we devise a *semantics-aware impact dependency analysis* method to provide a succinct and intuitive graph representation to illustrate the hooking mechanisms. We have developed a prototype, HookFinder, and conducted extensive experiments using representative malware samples from various categories. The experimental results demonstrated that HookFinder correctly identified the hooking behaviors for all the samples, and provided accurate insights about their hooking mechanisms.

1 Introduction

The arms race between malware writers and malware defenders has become evident. In order to evade malware defense techniques, malware writers are always striving to explore novel attacking techniques. In response, on the outbreak of new malware, it is critical for malware defenders to have an accurate and responsive understanding of the attacking mechanisms in order to win the battle.

One important malware attacking vector that needs to be understood is its hooking mechanism. Malicious programs implant hooks for many different purposes. Spyware may implant hooks to get notified of the arrival of new sensitive data. In particular, keyloggers may install hooks to intercept users' keystrokes, password thieves may install hooks to get notified of the input of users' passwords, network sniffers may install hooks to eavesdrop on incoming network traffic, and BHO-based adware may also install hooks to capture URLs and other sensitive information from incoming web pages. In addition, rootkit may implant hooks to intercept and tamper with critical system information to conceal its presence in the system. Furthermore, stealth backdoors may place hooks on the network stack to establish a stealthy communication channel with remote attackers.

With prior knowledge of how existing malware implants hooks, several tools [4, 13, 20] check known memory regions for suspicious entries. However, they are completely ineffective when malware makes use of new approaches to install hooks. This concern is not hypothetical. Recently, a stealthy kernel backdoor, *deepdoor* [21], was developed using a novel method to hook the network stack, and has been demonstrated to be able to successfully evade all the existing detection methods.

In response to rapidly innovated malware techniques, the anti-malware society needs an effective mechanism to discover new hooks and understand their hooking mechanisms in a timely manner. Unfortunately, the existing malware analysis procedure is painstaking, mostly manual and error-prone. Various code obfuscation techniques employed by malware writers make this process even more difficult. In this paper, we propose the first systematic approach to this research problem. In particular, given an unknown malicious binary, we aim to identify if this code installs any hooks into the system, and if so, provide detailed information about how it

installs the hooks.

The intuition of our approach is that a hook implanted by the malicious code is one of the impacts (in terms of memory and registers) that the malicious code has made to the whole system, and this impact eventually affects the execution flow of the system to jump into the malicious code. In order to capture this distinct behavior, we propose a novel approach, fine-grained impact analysis. It works by identifying all the impacts made by the malicious code, and keeping track of the impacts flowing across the whole system. If the control flow is affected by one of these impacts to jump into the malicious code, then we determine that this transition is caused by a hook, which is installed by the malicious code. To understand how this hook is implanted, we perform dependency analysis on the history of impact propagation, leveraged with OS-level semantics.

To explore the feasibility of our approach, we have designed and developed a prototype, called HookFinder. Our experiments have demonstrated that for each malware sample in our test set, HookFinder is able to identify the hooks created by the sample and give valuable insights about their hooking mechanisms within minutes. We also believe that HookFinder can be automated to categorize the large volume of malware samples that anti-virus companies receive everyday with respect to their hooking behaviors, and instantly realize and respond to novel hooking mechanisms.

In summary, this paper makes the following contributions:

- We propose fine-grained impact analysis as a unified approach to identify the hooking behavior of malicious code. Since it does not rely on any prior knowledge of hooking mechanisms, our approach is well fitted in identifying novel hooking mechanisms.
- In order to provide valuable insights about how malware implants hooks, we devise a semantics-aware impact dependency analysis method, which provides a succinct and intuitive graphical representation to help malware analysts understand the hooking mechanism employed by this malware.
- We have designed and developed HookFinder to demonstrate the feasibility of our approach. We have conducted extensive experiments with representative malware samples from various categories, and demonstrated that HookFinder could correctly identify their hooking behaviors, and provide accurate insights about their hooking mechanisms.

The paper is structured as follows. The next section gives an overview of our approach. Section 3 describes details on the design and implementation of HookFinder. Section 4 presents the experimental results. Section 5 discusses some related issues. Section 6 surveys related work and Section 7 concludes the paper.

2 Problem Statement and Our Approach

In this section, we formally define the problem of hooking behavior detection and analysis, and give a brief overview of our approach.

2.1 Problem Statement

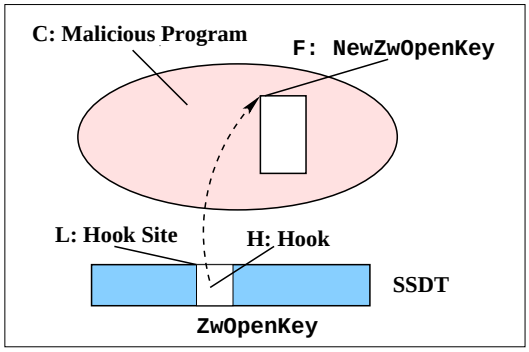
Given a malware sample, we aim to determine whether it contains hooking behaviors. If so, we want to reveal details about its hooking mechanism. A hooking behavior can be formalized as follows. A malicious program C contains a local function F, and attempts to implant a hook H into a memory location L of the system. When a certain event happens, the system will load the hook H, and then the execution is redirected to F. We refer to the address of F as *hook entry*, and L as *hook site*. Figure 1(a) shows a piece of pseudo code that hooks an entry in the System Service Descriptor Table (SSDT) of Windows system. This hooking mechanism is used in many kernel-mode malware samples, such as the Sony Rootkit [22]. In this example, `NewZwOpenKey` is the hook entry F, the hook site L is the actual entry for `ZwOpenKey` in the service descriptor table, and the hook H is the address of `NewZwOpenKey` in that entry, as illustrated in Figure 1(b).

Data Hook vs. Code Hook A hook H can be either data or code. If H is interpreted as data by the system, and is used as the destination address of some control transfer instruction to jump into the hook entry F, we term it a *data hook*. For example, the hook in Figure 1 is a data hook, because it is the address of the hook entry, and is interpreted as the jump target. H can also be interpreted as code. In this case, we call it a *code hook*. A code hook contains a jump-like instruction (such as `jmp` and `call`), and is injected to overwrite some system code

```
#define SYSTEMSERVICE(_function) \
  KeServiceDescriptorTable.ServiceTableBase \
  [*(PULONG)((PUCHAR)_function+1)]

void HookSyscalls() {
  ...
  OldZwOpenKey = SYSTEMSERVICE(ZwOpenKey);
  SYSTEMSERVICE(ZwOpenKey) = NewZwOpenKey;
  ...
}
```

(a)



(b)

Figure 1: **An SSDT Hooking Example**. This code attempts to hook `ZwOpenKey`, by writing the address of its own function `NewZwOpenKey` into the corresponding entry of the SSDT `KeServiceDescirptorTable`.

(such as kernel modules and common DLLs). When the overwritten system code is executed, the execution will be redirected into the malicious code. We need to identify both cases, and we should be able to tell what kind of hook it is, when we identify one. As we will see later, the policies used to detect hooking behaviors are different between these two categories due to their different nature.

Direct Modification vs. Function Call Malware has two choices to install H into L. First, it may directly write H into L using its own code. Second, it may call a function to achieve it on its behalf. Windows system provides several APIs for applications to register various event handlers (i.e., hooks). For example, `SetWindowsHookEx` allows an application to register a hook for certain Windows event, such as keystroke events. Whenever a keystroke is entered into the system, Windows will call the hook function provided by this application. In addition, functions like `memcpy` and `WriteProcessMemory` can overwrite a memory region on behalf of their callers. Thus, once we identify a hook, we need to determine which method the malware used to register the hook.

If the malware directly modifies L to install H, we need to understand where L is, and how the malware sample obtains L. Since L is usually not located in a fixed place, malware has to find it from some static point. This static point can be a global system symbol, or the result of a function call. After obtaining this static point, malware may walk through the data structures referenced by it to eventually locate L. The example in Figure 1 makes use of this method, and the hook site L is calculated from a global symbol `KeServiceDescriptorTable`. Therefore, if the malware directly overwrites L, we need to answer the following questions:

- Where is the static point?
- How does the malware obtain the static point?
- How does it infer the final location L from the static point?

If the malware invokes an external function to register H, we need to identify the function's address and name. In addition, we need to know the actual arguments that are used to call this function. The function call and its argument list can give semantic information about how the hook and what kind of hook is registered. For example, if we identify that a malicious program calls `SetWindowsHookEx` to register a hook, we are able to tell from the first argument what type of hook is registered. Therefore, if the malware invokes an external function to register F, we need to answer the following questions:

- What is the external function, including its entry address and its name?
- What arguments does the malware use to invoke this function?

2.2 Our Approach

Since most of malware programs include various code obfuscation techniques to foil static analysis, our approach is based on dynamic analysis. That is, we actually run malware in a special environment, and observe

how it implants the hook, and how the hook is activated by the operating system. Our approach is divided into two steps: hook detection and hooking mechanism analysis.

Hook detection: fine-grained impact analysis Our approach is based on the following intuition. Malicious code makes changes, including memory and the other machine state changes, to the execution environment as it runs. We call these changes as *impacts*. Obviously, a hook H is one of the impacts made by the malicious code, and this impact finally redirects CPU's control flow into the malicious code. Hence, if we are able to identify all the impacts of the malicious code, and observe one of the impacts being used to cause the execution to be redirected into the malicious code, we can determine a hook installed by the malicious code. Furthermore, we are also interested in how an impact is formulated since initially, for the purpose of understanding the hooking mechanism. Therefore, we identify the *initial impacts*, the newly introduced impacts by the malicious code, keep track of the impacts propagating over the system.

Based on this intuition, we propose *fine-grained impact analysis*. We mark all the initial impact made by the malicious code at the byte level. The initial impacts include the data written directly by the malicious code, and the data written by the external code on its behalf. Then we keep track of the marked impacts propagating through the whole system. During the execution, if we observe that the instruction pointer (i.e., `EIP` in x86 CPUs) is loaded with a marked impact, and the execution jumps immediately into the malicious code, then we identify a hook. Furthermore, we have determined that the jump target is the hook entry F, the memory location that the instruction pointer is loaded from is the hook site L, and the content within L is the hook H.

Hooking mechanism analysis: semantics-aware impact dependency analysis Once identifying a hook H, we want to understand the hooking mechanism. During the impact propagation, we record into a trace the details about how the impacts are propagated in the system. Therefore, from the trace entry corresponding to the detected hook H, we can perform backward dependency analysis on the trace. The result gives how the hook H is formulated and installed into the hook site L. However, such a result is difficult to understand, because it only provides hardware-level information and sometimes can be big. We combine OS-level semantics information with the result, and perform several optimizations to hide unnecessary details. The final output is a succinct and intuitive graphical representation, which is straightforward for malware analysts to understand its hooking mechanism.

Note that our approach would catch "normal" hooking behaviors. Windows provides a number of APIs, such as `CreateThread` and `CreateWindow`, for applications to register their callback functions. Windows will invoke these callbacks on certain events. These normal hooking mechanisms can be compiled into a white-list. Then when normal looks will be captured by our detection approach, we can classify them as normal hooks, by extracting their hooking mechanisms and comparing with the white-list.

3 System Design and Implementation

To demonstrate the feasibility of our approach, we design and implement a system, HookFinder, to identify the hooking behavior and understand the hooking mechanism. In this section, we give an overview of HookFinder and describe its components.

3.1 System Overview

The overview of HookFinder in illustrated in Figure 2. HookFinder is based on a *whole-system emulator*. It emulates an x86 computer and runs a Windows guest system on top of it. The malware to be analyzed is executed in the Windows guest. There are two reasons why we employ a whole-system emulator. First, it facilitates instrumenting CPU instructions in a fine-grained manner. In particular, we are able to instrument every CPU instruction executed in the Windows guest system. Second, it provides an excellent isolation between the analysis environment and the malware. Therefore, it is extremely difficult for malicious code to interfere with our detection and analysis procedure and affect the analysis results. In the implementation, we use QEMU [2] as our emulator, due to its efficiency and its open source code.

Within the emulator, we build three components: *impact analysis engine*, *semantics extractor*, and *hook detector*. The impact analysis engine is a central component, which performs fine-grained impact analysis. It marks the impacts made by the malware, and keeps track of impacts propagating over the whole system. A

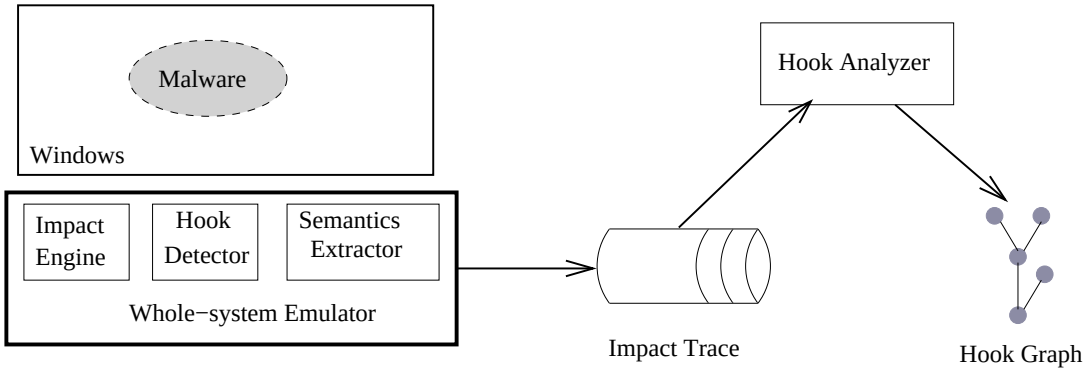


Figure 2: System Overview

whole-system emulator only provides a hardware-level view of the system, such as the states of CPU registers, physical memory, and I/O devices. However, malware analysts need to understand the malware and system behaviors at the operating-system level. The semantics extractor implements the functionality of extracting OS-level semantics information from the emulated environment. For example, it provides process and module information of the current instruction executed. It can also provides information of external function calls. The hook detector behaves like a controller, cooperating with the impact analysis engine and the semantics extractor to identify hooks.

To analyze hooking mechanisms, the impact propagation events, as well as necessary OS-level semantics information, are recorded into a trace, called the *impact trace*. The *hook analyzer* analyzes the impact trace and generates a succinct and intuitive graphical representation, *hook graph*. The hook graph conveys essential information for malware analysts to easily understand the hooking mechanism.

3.2 Impact Analysis Engine

The impact analysis engine performs fine-grained impact analysis, and is composed of two sub-components: *impact marker* and *impact tracker*. The impact marker is responsible for marking the initial impacts made by the malicious code, and the impact tracker keeps track of the impacts propagation.

Impact Marker In the impact marker, we aim to identify all the initial impacts that can be used to install the hooks. This is important, because if we fail to mark some initial impacts, malware writers may exploit this fact to evade our detection.

First, we consider the case that an instruction from malicious code directly make an impact. In this case, we mark the destination operand, either memory location or a CPU register, if it is not marked already. Note that the impact marker needs the information from the semantics extractor to determine whether an instruction is from the malicious code.

In addition, we consider the case that malicious code may make an initial impact by calling an external function. For example, it may call `ReadFile` to obtain the address of the hook entry F from a configuration file, and then install it as the hook H into the hook site L by calling `memcpy`, without H being marked. Therefore, we need to mark the output of that external function too. Again, the semantics extractor can determine when an instruction is executed under the context of an external function call.

To identify the outputs of an external function, we developed two different methods, for the registers and memory, respectively. For register outputs, we take a snapshot on the entry of external function call, and when the external function returns, we compare the register states with the snapshot, and mark the registers with different values, with the exception of `ESP`, `EBP` and `EIP`. For memory outputs, we mark a memory location if it is written under the context of the external function call, and it is not a local variable on the stack. To determine a local variable, we obtain the stack range for the current thread from the semantics extractor, and compare the memory location with the value of `ESP` on the entry of the external function call: if the memory location is smaller than the value of `ESP` and within the stack range, then it is a local variable.

Furthermore, malware may dynamically generate new code. The self-generated code is also part of impacts made by the malicious code, and therefore must be marked. Thus, we can determine if an instruction is generated from the original malicious code by simply checking if the memory region occupied by that instruction is marked. If so, we also treat that code region as malicious code, and mark the inputs taken by the self-generated code too.

Impact Tracker The impact tracker keeps track of the impacts propagating over the system. It tracks all the data dependencies between source and destination operands. That is, if any byte of any source operand is marked, the destination operand is also marked. In addition, for a memory source operand, if its address becomes marked, we also mark the destination operand. This policy enables us to track how the malicious code walks walk through a data structure, starting from a marked pointer to the data structure. These two policies are similar to those in the dynamic taint analysis systems [7, 10, 11, 18, 28].

What makes impact tracker really different is the way it checks immediate operands. That is, if an instruction has an immediate operand, the impact tracker checks if the memory region occupied by this immediate is marked and propagates impact accordingly. In contrast, the dynamic taint analysis systems treat immediate operands as clean. In our scenario, instructions including immediate operands may be generated by the malicious code, and therefore need to be checked. For example, in the code hook case, the malicious code may inject into the system code a jump instruction with a hard-coded target address, to redirect the execution to the malicious code. This immediate operand is deliberately injected by the malicious code to set up a hook.

To enable dependency analysis, the impact tracker performs an extra operation during the impact propagation. That is, we assign a unique identifier to each marked byte of the destination operand. We refer to this identifier as *dependency ID*. Then for each instruction that creates or propagates the marked data, we write a record into the impact trace. The record contains the relationships between the dependency IDs of marked source and the destination operand, associated with other detailed information about that instruction.

3.3 Semantics Extractor

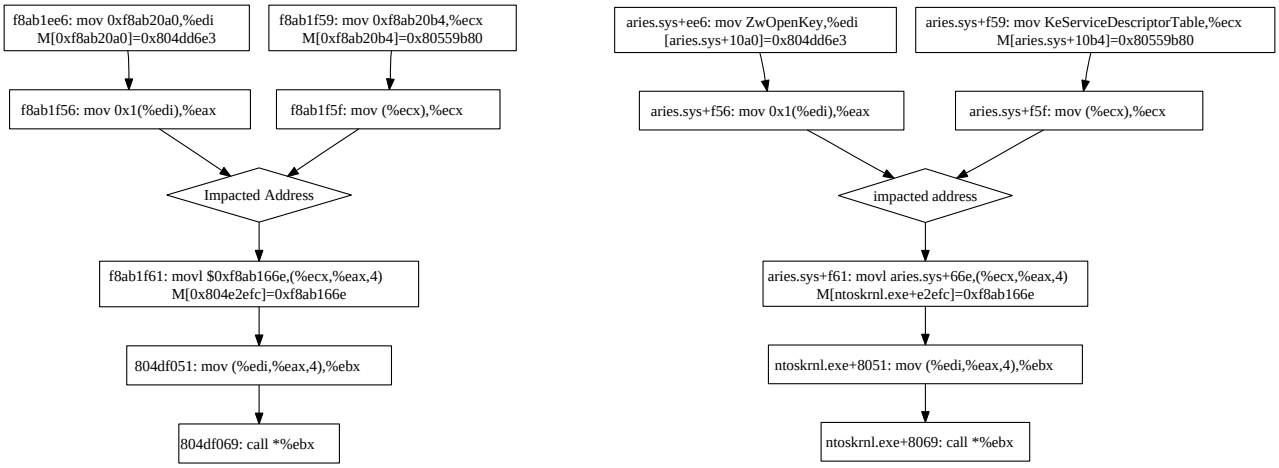
The semantics extractor bridges the semantic gap between the hardware-level view and the software-level view. Specifically, the purposes of the semantics extractor are three-fold: (1) determine the process, thread, and module information of the current executed instruction; (2) determine if an instruction is executed in the context of an external function call, and if so, resolve its function name and arguments; and (3) determine the symbol name if a memory read is to a symbol.

Process, Thread, and Module Information Several previous systems [10, 14, 28] have discussed extracting OS-level semantics from a virtual machine monitor or a whole-system emulator. Theoretically, the emulator is able to extract information about process, thread and module, by examining the emulated system states. However, for the simplicity of implementation, we employ the technique proposed in [28]. That is, we have developed a kernel module and inserted into the emulated operating system to collect the process, thread, and module information.

External Function Call Previous systems [10, 28] have also discussed how to determine external functions called by the malicious code, by comparing the stack pointers. The intuition is that the malicious code has to push the arguments and the return address onto the stack to call an external function. Thus by comparing the stack pointer when the execution enters the malicious code, and the one when the execution leaves, we can determine if the execution jumping out of the malicious code is because of an external function call. We realize this idea in our implementation of HookFinder.

Then given the entry address of an external function, we want to resolve its function name. We achieve this by parsing the PE header of a module whenever it is loaded into the system. Each binary in the PE format contains a table (Export Table) that for each of its exported functions maps its name with its offset within the binary. Combining the offset with the base address that the module is actually loaded in, we can infer the actual address of an external function.

Symbol Name When an instruction reads a memory location, we want to determine if it is reading a symbol, and if so, resolve the symbol name. This is useful in generating an OS-level hook graph. Similarly to resolving



(a) Hardware-level hook graph (b) OS-level hook graph

Figure 3: Hardware-level and OS-level hook graphs for a hook in Sony Rootkit.

external function name, we parse the PE header of a module whenever it is loaded into the system. We extract symbol names with their offsets in both Export Table and Import Table, and infer the actual address of a symbol using the module base address and its offset.

3.4 Hook Detector

The hook detector works by checking if the control flow is affected by some marked value, which redirects the execution into the malicious code. More precisely, we observe whether the instruction pointer `EIP` is marked, and the execution jumps immediately from the system code into the malicious code region, or the code region generated from the malicious code. If the conditions are satisfied, we identify a hook: the jump target is the hook entry F, the memory location that EIP is loaded from is L, and the content in L is H.

The above policy functions properly for identifying data hooks, but is problematic for code hooks. This is because a code hook is a piece of code generated by the malicious code, and thus is treated as malicious code by the above policy. Therefore when the code hook redirects the execution to the malicious code, the above policy will not raise an alarm because it sees the execution being transferring from malicious code to malicious code. To solve this problem, we extend the above policy, such that the execution transitions from a code hook region into malicious code will raise an alert.

Then the question is how to distinguish code hook regions with the other self-generated code regions. Self-generated code usually remains in the module space of the malicious code, or stays on a region that is not occupied by any module (such as in heap), whereas a code hook region is a piece of code that overwrites a code region in a different module. Therefore, during execution, if the currently executed basic block is marked and from a different module, and EIP is marked and jumps into the malicious code, we identify a code hook.

3.5 Hook Analyzer

Once a suspicious hook is identified, the hook analyzer is able to extract essential information about its hooking mechanism by performing *Semantics-aware dependency analysis* on the impact trace. The procedure consists of the following three steps: (1) from the hook H, perform backward dependency analysis on the impact trace, and generate hardware-level hook graph; (2) with the OS-level semantics information, transform the hardware-level hook graph into an OS-level hook graph; and (3) if necessary, simplify the hook graph by hiding unnecessary details and merging the similar nodes. We detail these steps respectively.

Hardware-level Hook Graph Remember that each record in the impact trace has the dependency information. Once identify a hook H, we obtain its dependency ID ID_h. Since the impact trace records the relationships between dependency IDs, we first search forward the impact trace for the record that defines ID_h. From that

record, we obtain the source dependency IDs that ID_h depends on. Then for each of the source dependency IDs, we search backward for the record that defines it. If there are any source dependency IDs in that record, we also perform the backward search for those IDs. We perform this backward search recursively until all records have been located. Then we connect these records according to their dependency relationships. Besides the dependency information, each record contains detailed information about an instruction, such as its address and the values of its operands. If the instruction is executed under the context of an external function, the record also contains the entry address of that external function, and the value of ESP on the entry of call. We put this detailed information into the node corresponding to the record. The resultant graph is the hardware-level hook graph. Figure 3(a) shows a real hardware-level hook graph for a hook in Sony Rootkit [22], which employs the same hooking mechanism as the sample shown in Figure 1. A rectangle node denotes an instruction propagating malware's impacts. A diamond node denotes that its successor's destination address affected by the malware. Note that to save the space, we only display really important information for each node, such as the instruction address and the dissembled instruction. For each memory operand, we show its address and value. If the instruction is executed under the context of an external function call, we can also show the entry of the function call and the ESP value on the entry.

OS-level Hook Graph With the OS-level semantics information provided by the semantics extractor, we can transform a hardware-level hook graph into an OS-level hook graph. Given the address of the instruction, we can show which module it belongs to and its offset to the module base. Similarly for memory access, we can determine if it falls into any module space. If the memory access is to a symbol, we can even resolve the symbol name. Given the entry address of an external function, we can resolve its function name. Then, the resultant graph is an OS-level hook graph. Figure 3(b) illustrates the OS-level hook graph that is transformed from Figure 3(a). We can see that Figure 3(b) correctly reflects the hook registration procedure shown in Figure 1. That is, symbols `ZwOpenKey` and `KeServiceDescriptorTable` are used to calculate the hook site L (shown in the diamond-shaped node), and an address (aries.sys+66e) is written into L. This is H, the address of the hook entry F.

In addition to resolving function name, HookFinder can also identify the function arguments from the impact trace. Because pushing arguments is a type of impacts, those operations is recorded in the trace. For each function activation in the trace, HookFinder locates the first record of the activation, denoted as i. The records precede record i contains function arguments. Normally, function arguments is pushed in reverse order, i.e., the first argument is pushed onto the stack last.

Graph Simplification Sometimes, the resultant hook graph can be very complex. For better readability and clarity, we simplify it using the following criteria. (1) if the adjacent two nodes belong to the same external function call, merge them together. (2) if the adjacent two nodes are move-like instructions, such as mov, push, and pop, merge them together, and if those instructions propagate the same value without modification, we merge those move-like instruction into a single node. We apply these two policies repeatedly until no nodes can be merged.

4 Evaluation

In this section, we present details on the experimental results of HookFinder, by evaluating it with real-world malware samples. We first give a summary of the experimental results over these samples, and then present details on three of them. In all our experiments, we run HookFinder on a Linux machine with a dual-core 3.2 GHz Pentium CPU and 2GB RAM. On top of HookFinder, we install Windows XP Professional SP2 with 512M of allocated RAM as the guest operating system.

4.1 Overview

Our sample set consists of eight malware samples, which are obtained from public resources (such as [17, 19]) and collaborative researchers. In Table 1, we characterize these samples according to whether they are packed, whether they are kernel or user threats, and which categories they belong to. Since deepdoor is not released by its author, we use a similar kernel backdoor, *Uay Backdoor*, which resembles deepdoor's hooking mechanism. We include Uay backdoor to verify the capability of HookFinder in identifying novel hooks.

Sample	Size	Packed?	Kernel/User	Category
Troj/Keylogg-LF	64K	Y	User	Keylogger
Troj/Thief	334K	N	User	Password Thief
AFXRootkit [1]	24K	Y	User	Rootkit
CFSD [6]	28K	N	Kernel	Rootkit
Sony Rootkit [22]	5.6K	N	Kernel	Rootkit
Vanquish [25]	110K	N	User	Rootkit
Hacker Defender [12]	110K	N	User&Kernel	Rootkit
Uay Backdoor [24]	212K	N	Kernel	Backdoor

Table 1: Malware Samples in Our Experiment

In the experiment, HookFinder has successfully identified hooks for all the samples. We summarize the results in Table 2. In the second column of Table 2, we list the elapsed time for each sample. It breaks down into two parts: the runtime for running the sample in the emulated environment (shown as the first number), and the runtime for generating hook graphs (as the second number). After executing a sample, we wait for 2-3 minutes to make sure it has fully started. In order to trigger potential hook behavior, we then perform a series of simple interactions with the emulated system, including listing a directory, and pinging a remote host, which may cost another 2 or 3 minutes. The runtime for generating hook graphs varies from 2 seconds to 33 minutes, depending on the trace size, the number of hooks, and other factors. In total, HookFinder spends up to 39 minutes on a sample during the evaluation, which is efficient compared to manual malware analysis that can last hours or days.

The third column lists the size of the impact trace for each sample. As we can see, the maximum size in the table is 14G, which is acceptable for a complex program executing millions of instructions.

The fourth and fifth column shows the number of suspicious hooks and the total number of identified hooks, for each sample. We found some normal hooks registered by the following functions: *EVENT_SINK_AddRef*, *FltDoCompleteProcessingWhenSafe*, *StartServiceDispatcherA*, *CreateThread*, *CreateRemoteThread*, and *PsCreateSystemThread*. Note that our approach does not distinguish the intent of a hooking behavior. Thus, we will identify all hooks in the first place; then we may maintain a white-list for normal hooking mechanisms.

The last column gives essential information about the hooking mechanism. We found that three samples installed code hooks. All three samples derive the hook sites by calling *GetProcAddress*. Vanquish directly writes the hooks into the hook sites, whereas AFXRootkit and Hacker Defender call *WriteProcessMemory* and *NtWriteVirtualMemory* respectively to achieve it. The other six samples installed data hooks, four of which call external functions to install the hooks. In particular, CFSD calls *FltRegisterFilter*, and Trojan/Keylogg-LF and Troj/Thief call *SetWindowsHookEx*. We also extracted arguments for these function calls, and we found that Trojan/Keylogg-LF installed a WH_KEYBOARD_LL hook, and Trojan/Thief installed a WH_CALLWINDPROC hook. The remaining two samples directly write hooks into hook sites. The static points are *KeServiceDescriptorTable* and *NdisRegisterProtocol* for Sony Rootkit and Uay Backdoor, respectively.

4.2 Detailed Analysis

Here we present detailed results for two malware samples: Uay Backdoor and Vanquish. The hook graph of each sample is shown in Figure 4.

Uay backdoor HookFinder identified five data hooks in total for this sample. We reviewed the generated hook graphs, and we found that three of them were installed by *PsCreateSystemThread*. This kernel function creates a system thread with the thread entry provided by the caller. Thus, these three hooks are normal hooks. The other two are suspicious, and their hook graphs are similar. We show one graph in Figure 4(a). We also show the corresponding unsimplified hardware-level graph in Figure 5 in the Appendix.

Sample	Runtime	Trace	Hooks		Hooking Mechanism
			Total	Malicious	
Troj/Keylogg-LF	6m+9m	3.7G	2	1	Data, Call:*SetWindowsHookEx(WH_KEYBOARD_LL,...)*
Troj/Thief	4m+3s	143M	1	1	Data, Call:*SetWindowsHookEx(WH_CALLWINDPROC,...)*
AFXRootkit	6m+33m	14G	4	3	Code, Call:*WriteProcessMemory*
CFSD	4m+2m	2.8G	5	4	Data, Call:*FltRegisterFilter*
Sony Rootkit	4m+2s	25M	4	4	Data, Direct, Static Point:*KeServiceDescriptorTable*
Vanquish	6m+12m	4.4G	11	11	Code, Direct, Static Point:*GetProcAddress*
Hacker Defender	5m+27m	7.4G	4	1	Code, Call:*NtWriteVirtualMemory*
Uay backdoor	4m+25s	117M	5	2	Data, Direct, Static Point:*NdisRegisterProtocol*

Table 2: Summarized experimental results

As we can see in Figure 4(a), there are two branches in the bottom. The left branch describes how the hook site L was inferred, and the right branch presents how the hook H was formulated. From the top of the right branch, we can see that H originated from the output of a function call *NdisAllocateMemoryWithTag*. This kernel function is used to allocate a memory region in the kernel space. According to the function's semantics, this output has to be the address of the allocated memory region. This address is finally implanted into the hook site L.

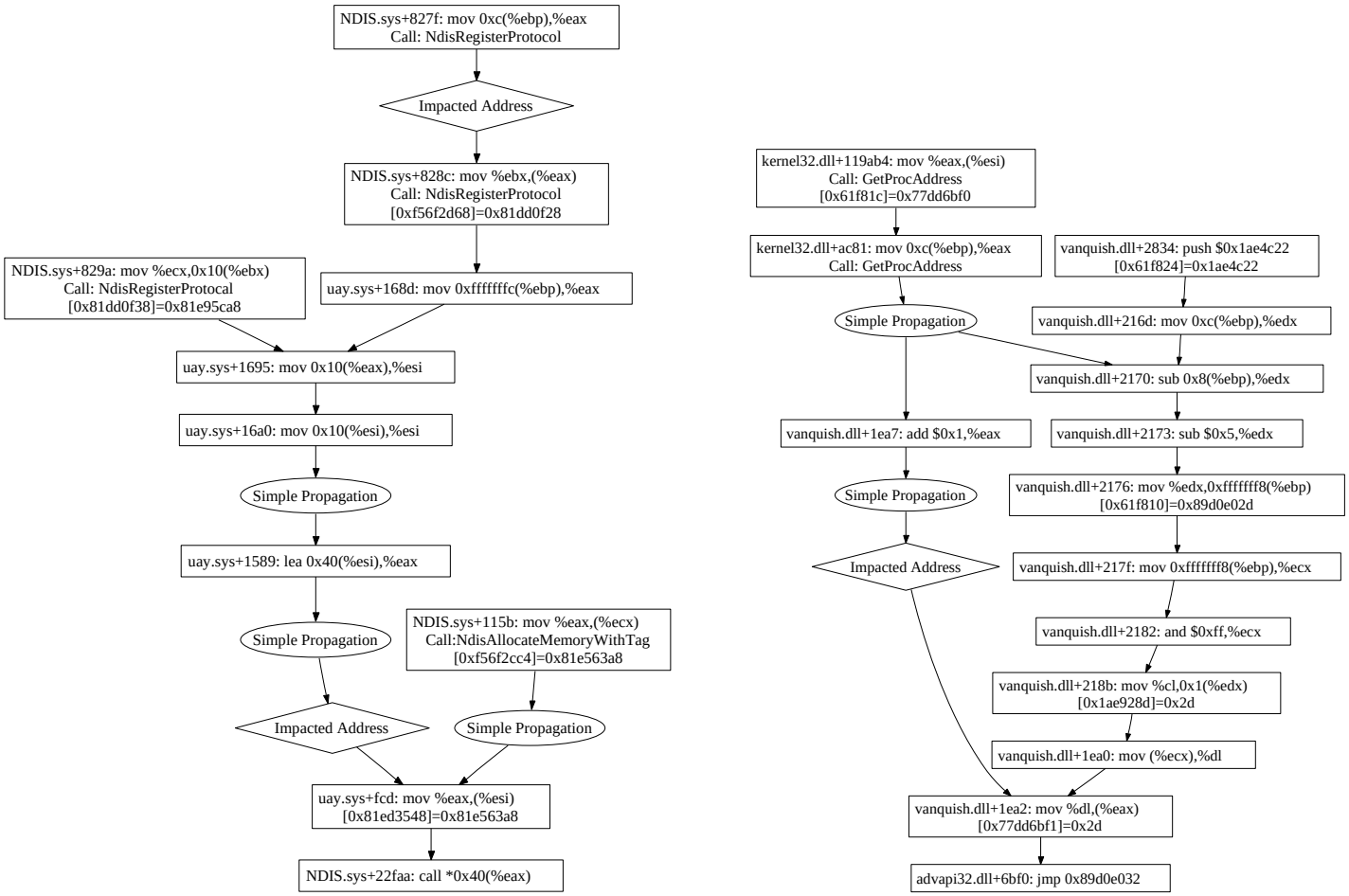
From the top of the left branch, we observe that L is derived from the output of a function call *NdisRegisterProtocol*. This kernel function registers a network protocol. According to the function semantics, we believe this output is the protocol handle in the second argument. This handler points to an internal data structure maintained by the Windows kernel. Then we can see the instruction (at uay.sys+1695) read a field with the offset 0x10 in this data structure. The obtained value (v_1) is then used as a pointer to read another value (v_2) from the offset 0x10 in the data structure pointed by v_1, in the subsequent instruction (at uay.sys+16a0). Then, the instruction (at uay.sys+1589) adds v_2 with 0x40, and the resulted value is eventually used as the hook site L. We believe that this sample actually walks into this internal data structure that it obtains from *NdisRegisterProtocol*, and locates the designated hook site L. Interestingly, the definition of the data structure for the protocol handle created from *NdisRegisterProtocol* is not released in any documentation from Microsoft, but this malware sample seems to be able to understand this data structure, and knows how to locate the desired hook site from it.

The hook graph for another suspicious hook is very similar to this one, except that it adds v_2 with 0x10. With the knowledge of how this internal structure is defined, we would be able to tell which two functions this malware sample actually hooked.

Vanquish HookFinder identified 11 code hooks in total for Vanquish. After reviewing the hook graphs, we found that Vanquish hooked four unique APIs: *RegCloseKey*, *LoadLibraryExW*, *RegEnumKeyW* and *RegEnumKeyExW*. Thus, multiple hooks may correspond to one API hooking, because Vanquish installs one hook per process for that API.

We show a hook graph for hooking *RegCloseKey* in Figure 4(b). The other hook graphs are similar. First, we can see the bottom node. This is the actual instruction Vanquish injected into the system code to set up the hook. It is a `jmp` instruction, and its address is the entry point of *RegCloseKey*. The rest of the graph shows how the jump target of this instruction is formulated. Here the address of this jump target (i.e., 0x77dd6bf1) is the hook site L, and the content in L is H (i.e., 0x89d0e032). Again, the left branch represents how L was inferred, and the right branch indicates how H was formulated.

The left branch starts with the output of function call *GetProcAddress*. This function returns the actual function address, given an function name. Therefore, the source of the left branch is the address of a function call, and the actual value is 0x77dd6bf0, which is the address for *RegCloseKey*. As we follow the links down, we can see this address is added by 1 and used as L. Obviously, the offset 1 is for the opcode of `jmp`. Now



(a) Hook graph of Uay backdoor

(b) Hook graph of Vanquish

Figure 4: Analysis results. A rectangle node denotes an instruction. A diamond node denotes that its successor's destination address is derived from this node. An ellipse node labeled "simple propagation" denotes a sequence of data moves.

for the right branch, we can see that it originates from an immediate (0x1ae4c22) pushed onto the stack. This value is first subtracted by the address for *RegCloseKey*, and then subtracted by 5. Then the value is "and" with 0xff to get the lowest byte, and this byte is written to the hook site L directly. Obviously, these steps are used to calculate the relative address for the `jmp` instruction.

5 Discussion

In this section, we discuss the resilience of our system to various evasion techniques that malware writers may exploit.

Exploiting Control Dependency The essential of our approach is to identify all the impacts made by the malicious code, and keep track of the impact propagation via data dependency. It is natural for malware writers to think of exploiting control dependency, to evade our detection. For example, the malicious code may embed a complex switch statement like below to cut the data dependency between a and b.

```
switch(a)
{ case 1: b=1; break; case 2: b=2; break; ... }
```

This evasion is not viable. This is because that in the impact marker, we thoroughly mark all the initial impacts (i.e., memory and register writes) made by the malicious code. Thus, the output b will be marked

anyway.

Not Exhibiting Hooking Behaviors When Tested Malware may not exhibiting hooking behavior during the dynamic analysis. It may stay inactive until certain conditions are satisfied. Malware may also detect the presence of the emulated environment and stay dormant. In those cases, HookFinder cannot detect hooking behavior. This is a common shortcoming of dynamic analysis. Some complementary work has been done to address this problem. Vasudevan et al. proposed several stealthy techniques, such that malware cannot easily detect the analysis environment [26]. Moser et al. [15] and Brumley et al. [3] also used QEMU to build malware analysis systems, which are able to uncover hidden behaviors of malware by exploring multiple execution paths. We will leave incorporating these techniques into HookFinder as future work.

6 Related Work

Hook detection. Researchers have developed several tools, such as VICE [4], System Virginity Verifier [20], and IceSword [13], to detect the existence of hooks in the system. With prior knowledge how malicious code usually set hooks, these tools examine known memory regions for suspicious entries. The common examined places are system service descriptor table (i.e., SSDT) exported by the OS kernel, interrupt descriptor table (i.e., IDT) that stores interrupt handlers, import address tables (i.e., IAT) and export address tables (i.e., EAT) of important system modules. Assuming that important system modules do not modify their code (with a few exceptions), System Virginity Verifier checks if code sections of important system DLLs and drivers remain the same in memory as those in the corresponding binaries on disk. In nature, these tools fall into misuse detection, and thus cannot detect hooks in previously unknown memory regions. In comparison, our approach captures the intrinsic characteristics of hooking behaviors: one of the malware's impacts has to be used to redirect the system execution into the malicious code. Therefore, it can identify unknown hooking behaviors. Moreover, it also provides insights about the hooking mechanisms.

Dynamic taint analysis. The fine-grained impact analysis resembles the dynamic taint analysis technique, which is proposed to solve and analyze many other security related problems. Many systems [8, 9, 16, 18, 23] detect exploits by tracking the data from untrusted sources such as the network being misused to alter the control flow. Other systems [7, 10, 28] make use of this technique to analyze how sensitive information is processed by the system. Chow et al. applies dynamic taint analysis to understand the lifetime of sensitive information (such as password) in operating systems and large programs [7]. Egele et al. utilize this technique to analyze BHO-based spyware behavior [10]. Yin et al. also make use of dynamic taint analysis to detect and analyze privacy-breaching malware [28]. Moreover, dynamic taint analysis is used for other applications, such as automatically extracting protocol message formats [5], and preventing cross-site scripting attacks [27].

7 Conclusion

In this paper, we presented a novel dynamic analysis approach, *fine-grained impact analysis*, to identify malware hooking behaviors. This approach characterizes malware's impacts on its system environment, and observes if one of the impacts is used to redirect the system execution into the malicious code. Since it captures the intrinsic characteristics of hooking behavior, this technique is able to identify novel hooks. Moreover, we devised a *semantics-aware impact dependency analysis* method to extract the essential information about the hooking mechanisms, which is represented as hook graphs. We developed a prototype, HookFinder, and conducted extensive experiments using representative malware samples from various categories. The experimental results demonstrated that HookFinder can correctly identify the hooking behaviors for all the samples, and generated hook graphs provide accurate insights about their hooking mechanisms.

References

[1] Afxrootkit. http://www.rootkit.com/project.php?id=23.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[3] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University School of Computer Science, January 2007.

[4] J. Butler and G. Hoglund. VICE–catch the hookers! In *Black Hat USA*, July 2004. http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf.

[5] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[6] Clandestine file system driver. http://www.rootkit.com/vault/merlvingian/cfsd.zip.

[7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (Security'04)*, August 2004.

[8] M. Costa. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, October 2005.

[9] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*, December 2004.

[10] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proceedings of the 2007 Usenix Annual Conference (Usenix'07)*, June 2007.

[11] A. Ho, M. Fetterman, C. Clark, A. Watfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys 2006*, April 2006.

[12] Hacker defender. http://www.rootkit.com/project.php?id=5.

[13] IceSword. http://www.antirootkit.com/software/IceSword.htm.

[14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference, General Track*, 2006.

[15] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy(Oakland'07)*, May 2007.

[16] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[17] Offensive computing. http://www.offensivecomputing.net/.

[18] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys 2006*, April 2006.

[19] rootkit.com. http://www.rootkit.com/.

[20] J. Rutkowska. System virginity verifier: Defining the roadmap for malware detection on windows systems. In *Hack In The Box Security Conference*, September 2005. http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt.

[21] J. Rutkowska. Rootkit hunting vs. compromise detection. In *Black Hat Federal*, January 2006. http://www.invisiblethings.org/papers/rutkowska_bhfederal2006.ppt.

[22] Sony's DRM Rootkit: The Real Story. http://www.schneier.com/blog/archives/2005/11/sonys_drm_rootk.html.

[23] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.

[24] UAY kernel-mode backdoor. http://uty.512j.com/uay.rar.

[25] Vanquish. https://www.rootkit.com/vault/xshadow/vanquish-0.2.1.zip.

[26] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized-Executions. In *Proceedings of 2006 IEEE Symposium on Security and Privacy (Oakland'06))*, may 2006.

[27] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*, February 2007.

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

Appendix: Hardware-level Hook Graphs

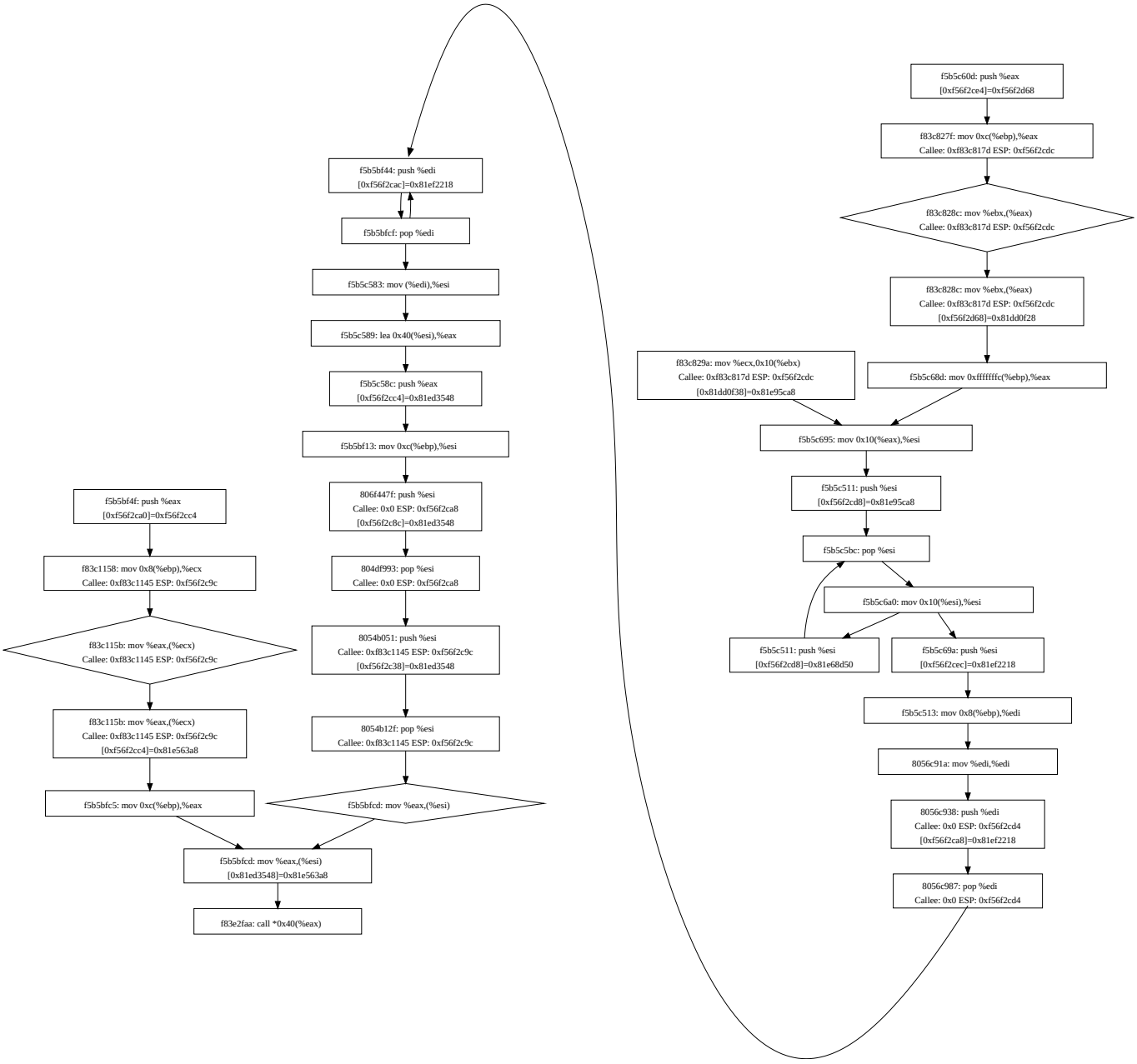


Figure 5: Hardware-level hook graph for Uay backdoor

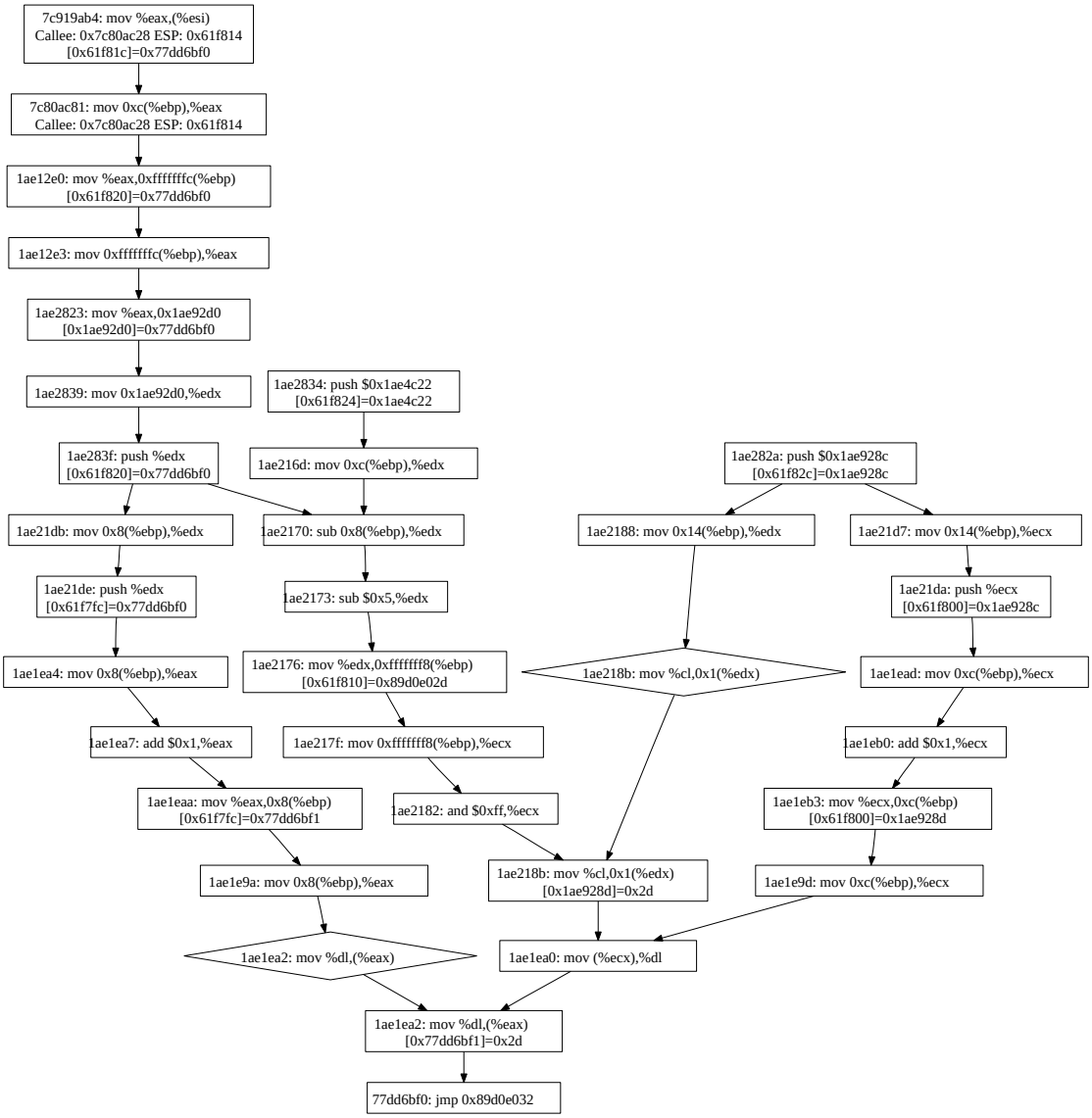


Figure 6: Hardware-level hook graph for Vanquish