J Hardw Syst Secur (2017) 1:114–136
DOI 10.1007/s41635-017-0013-2

A Survey on Function and System Call Hooking Approaches

Juan Lopez[1] · **Leonardo Babun**[1] · **Hidayet Aksu**[1] · **A. Selcuk Uluagac**[1]

Received: 26 January 2017 / Accepted: 18 August 2017 / Published online: 21 September 2017

Abstract Functions and system calls are effective indicators of the behavior of a process. These subroutines are useful for identifying unauthorized behavior caused by malware or for developing a better understanding of the lower-level operations of an application. Code obfuscation, however, often prevents user monitoring and modification of subroutine calls. Subroutine hooking offers a solution to this limitation. Function and system call hooking approaches allow for subroutine instrumentation, making hooking a valuable and versatile skill across industry and academia. In this survey, we present several criteria for the classification and selection of hooking tools and techniques as well as an examination of the major hooking approaches used on Windows, Linux, macOS, iOS, and Android operating systems. We also evaluate and compare the performance of different subroutine hooking tools and techniques based on computing resource utilization such as CPU time, memory, and wall-clock time. To the best of our knowledge, this is the first paper that encompasses both system call and function hooking techniques and tools across the major desktop and mobile operating systems.

✉ Leonardo Babun
lbabu002@fiu.edu

Juan Lopez
jlope518@fiu.edu

Hidayet Aksu
haksu@fiu.edu

A. Selcuk Uluagac
suluagac@fiu.edu

[1] Florida International University-Engineering Center, Miami, FL 33174, USA

Keywords System calls · API calls · Hooking techniques · Function interposition · Function interception · Binary instrumentation · Function instrumentation

1 Introduction

Function and system call hooking approaches are useful techniques used in both research and software development to monitor or modify a program's execution. There are a variety of situations where hooking is advantageous including malware identification [27, 33, 40, 60, 68, 70], software testing [43, 55, 61, 64], and software distribution [38]. The main reason behind function and system call hooking is the fact that the source code of most application are unavailable, making hooking one of the most utilized techniques for binary instrumentation [3, 54, 59] as well as malware reverse engineering [52, 69]. In essence, hooking a subroutine (function or system call) involves (1) identifying a subroutine in a process that needs to be instrumented (the target subroutine) and (2) performing modifications to the operating system, run-time environment, program libraries, or the executable program itself. These changes may be done before program execution or at run-time, and they will guarantee that the target subroutine will be instrumented whenever it is called.

This work presents a detailed catalog of the existing techniques and tools available to implement function and system call hooking on the most commonly used operating systems: Windows, Linux, macOS, iOS, and Android. At the time of writing this article, there was not any previous work that covered both function and system call hooking tools and techniques across the major operating systems. Moreover, improvements in technology have led to the development of new hooking tools and techniques, some of which have

not been examined in any previous work. We solve this limitation by (1) presenting a comprehensive explanation of hooking, covering the most important theoretical aspects of the technique, supported by other relevant work that has been done on the topic; (2) defining several different criteria used for the classification and selection of hooking techniques and tools; (3) examining and describing some of the most popular tools and techniques used for both function and system call hooking; and (4) presenting the results of our performance analysis on the overhead introduced by several of the hooking tools and techniques analyzed.

Organization The remainder of the paper is organized as follows. Section 2 presents background information about hooking. In Section 3, we examine some related work. In Section 4 , we propose different criteria for the classification and selection of hooking tools and techniques. Sections 5 and 6, examine specific function and system call hooking techniques and tools, respectively. In Section 7, the overhead introduced by several of these hooking techniques and tools is evaluated on the most commonly used operating systems. Lastly, Section 8 concludes the paper.

2 Background

In this section, we present some operating system concepts, including user-space, kernel-space, and shared libraries, which are fundamental in understanding the hooking process. In addition, we provide an overview of subroutine hooking.

2.1 Operating System Concepts Relevant to Hooking

Before presenting specific hooking concepts, we need to define the two segregated layers used in all operating systems and explain the use of shared libraries across various operating systems.

2.1.1 User-Space vs. Kernel-Space

Operating systems provide two modes of operation, a low-privileged *user-mode* to execute user-defined programs that are not part of the operating system, and a high-privileged *kernel-mode* to execute operating system code such as system drivers and services [56]. Because kernel-mode has direct access to the system hardware, it is important to restrict access to this mode. Typically, this is accomplished through the use of protection or privilege rings which are enforced by the hardware. Nearly all user programs require services provided by kernel such as a user interface, I/O operations, file management, and communication with other processes [58]. In this environment, system calls provide an interface to the operating system services, allowing user-mode programs to request services from the kernel. System calls may be called either directly from a user program or through the use of an *application programming interface* (API), which provides users with a set of abstracted functions that initiate the system calls necessary to link the higher-level requests with the operating system services. As a result of this operating system organization, *we need to examine hooking techniques for both functions (particularly system call APIs) in user-space and system calls in kernel-space*. Most system calls have user-space wrapper functions [31], so we can hook either the function or the actual system call. For example, to create a new process in Windows, a user can utilize the Windows API function, *CreateProcess*, which ultimately calls the system call *NTCreateProcess* in the kernel. Consequently, we could hook the Windows API using a function hooking technique, or we could directly hook *NTCreateProcess* using a system call hooking technique.

2.1.2 Shared Libraries

Most operating systems including Windows, macOS, Linux, iOS, and Android allow processes to use *static* and *shared libraries*. Programs that use static libraries are considered self-contained because these libraries are linked at compile-time. However, there are numerous libraries that may be used by more than one program such as those defined in the Windows API [23] or POSIX API [15]. Therefore, it would be wasteful to have every program contain their own copy of the same library. Shared libraries, as their name implies, allow multiple programs to access the same library file, reducing both memory and disk usage [51]. Furthermore, because shared libraries are linked at either program load-time or run-time, it is possible to make modifications to a shared library file without the need to modify or recompile any program that is dependent on this library. The flexibility provided by shared libraries can be exploited with a variety of techniques that allow for function hooking. On Windows, a shared library is known as a *dynamic-link library* (DLL). In Linux and Android, these libraries are identified with the *.so* extension, while in macOS and iOS, the *.dylib* extension is used. The differences in the implementation of shared libraries across different operating systems, in part, determine the available hooking techniques for each platform.

2.2 An Overview of Subroutine Hooking

We define *hooking* as *the interception of specific functions or system calls to monitor and/or alter the execution of the specified call*. Typically, this is accomplished through the use of *detour functions* or *detour system calls* [46]. For

simplicity, throughout this paper, we use the term *subroutine* to refer to both functions and system calls. Furthermore, we refer to the specific subroutine that will be hooked as the *target subroutine* and the process containing this subroutine as the *target process*. Moreover, we define *instrumentation* as code in a detour subroutine that monitors and/or modifies the target subroutine. Finally, we refer to the subroutine in the program code that calls the target subroutine as the *caller*. All hooking operations will eventually need to return control to the target subroutine caller.

Because most source code is unavailable, subroutine hooking is sometimes the only way to instrument a process. Figure 1 provides an overview of the typical steps involved in hooking a subroutine. As this figure shows, the hooking process first identifies the subroutine in a process that needs to be instrumented (the target subroutine). Then, it is necessary to select an existing detour subroutine or define a new one. This subroutine includes the instrumentation to trace and/or alter the target subroutine [39]. Changes to the target subroutine may include modifications to subroutine parameters and/or return values. Next, certain modifications must be made to the operating system, run-time environment, libraries, or executable file to install the hook. These necessary alterations will depend on a number of factors including the operating system, the subroutine type, either a function or system call, and the moment when the hook will be inserted (before program execution or at run-time). After the hook is installed, whenever a subroutine in the process calls the target subroutine, the code execution will jump to the detour subroutine. Therefore, the caller subroutine will effectively be calling the detour subroutine instead of the target subroutine. Finally, instrumentation of the target subroutine in the detour subroutine can occur before the target subroutine executes (*pre-processing*), after the target subroutine has executed and returned (*post-processing*), or in both cases [39]. For instance, with pre-processing instrumentation, the detour subroutine can change the input parameters to the target subroutine and call it with these modified parameters [12]. After the target subroutine executes completely, it once again jumps to the detour subroutine to execute post-processing instrumentation code such recording the subroutine return value. Consequently, a hook implementation that includes both pre- and post-processing will redirect execution to the detour subroutine multiple times. Ultimately, the detour subroutine will return control to the target subroutine caller [36].

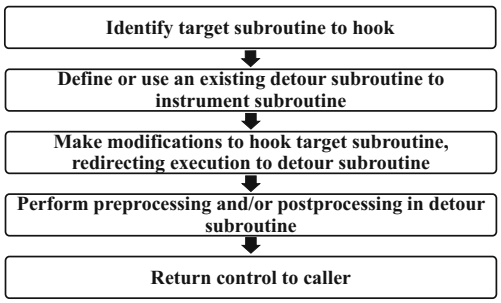


Fig. 1 The typical steps involved in subroutine hooking. The user must always select the target subroutine(s), but the rest of the steps may be automated by a hooking tool or utility

3 Related Work

There has been significant research on different mechanisms for subroutine hooking. A number of articles examine user-space techniques used for function hooking. Kim [42] discusses two techniques for intercepting functions, inline hooking (target function modification) using Microsoft Detours [13] and Import Address Table (IAT) hooking using Syringe, developed by the OK Thinking Software. Hunt and Brubacher provide a thorough explanation of inline hooking used by the library Microsoft Detours to intercept Win32 functions [39]. These authors also discuss Detours' mechanism to modify the IAT of Win32 binaries, allowing for function interception. Furthermore, Willems et al. [64] provide an overview of Windows API hooking, focusing on inline code overwriting and DLL code injection. Richter and Nasarre [53] also discuss Windows hooking, including an entire chapter on numerous Windows DLL injection techniques and different API hooking techniques including inline hooking and IAT modification. Mohd and Maarof [57] present three Windows API hooking techniques: inline hooking, IAT hooking, and debugger hooking. Crucially, the authors analyze the ease with which malware can detect the use of these techniques.

Other publications further discuss the use of function hooking in other operating systems. Myers and Bazinet [51] present several function hooking techniques across several operating systems, including the Detours library for Windows and the run-time environment variables for macOS and Unix systems. Zdziarski [71] examines the tool Cyscript, which can hook Objective-C methods in iOS applications. The author also describes how iOS dynamic libraries can be injected with the GNU debugger to intercept functions. For system call hooking, Wampler [63] examines three rootkit attacks on both Linux and Windows: system call modification, system call target modification, and system call table redirection, which can be used to hook system calls. Moreover, Tan et al. [45] investigate system call tracing tools on Android and iOS. The authors compare the utility *strace* to *utrace* on Android. They also discuss *Dtrace* and Apple Instruments tools for Mac OS and iOS. Because of several disadvantages with existing tools on iOS, the authors developed their own system call tracing tool named

iTrace. Andersson et al [28] examine code injection techniques on Windows and Unix, describing both function and system call hooking.

A major component of this paper is the evaluation of the performance overhead introduced while utilizing various hooking techniques and tools. Hunt and Brubacher [39] describe alternative function hooking approaches including DLL redirection and breakpoint trapping and compare the performance of these techniques to the Detours library using function invocation time. Marhusin et al. [48] analyze the performance of the Deviare API hooking engine on Windows, comparing it against a plain system (no hook) and a system running an antivirus. They instrument several functions in kernel32.dll related to file operations and examine differences in startup time across ten Windows programs. Tan et al. [45] examine the performance overhead of the system call tracing tools, strace on Android and iTrace on iOS.

Subroutine hooking is a major process in security applications widely used in malware detection, application behavior analysis, and compromised software monitoring. In [49], Mehdi et al. present a novel variable-length representation of system calls intended for malware detection. In [29], hooking techniques are utilized to create system and function call signatures of deterministic Cyber-Physical Systems (CPS) processes in order to detect compromised smart grid devices. Additionally, Xu et al. propose a novel malware detection mechanism based on API/system call tracing and malicious behavior of malware in [66]. Moreover, researchers in [47] utilize subroutine hooking techniques to trace system calls and detect malware instances and their corresponding malware families. Finally, authors in [44] examine system call lists to improve malware detection rates, and they use MapReduce to reduce the overhead of this analysis.

Differences from the Existing Works While various publications have investigated either user-space (function calls) or kernel-space (system calls) hooking, we have not found a single article that provides a thorough examination of both types of subroutine hooking. To the best of our knowledge, this is the first work that focuses on surveying the currently available hooking tools and techniques in both user-space and kernel-space across the five major operating systems: Windows, Linux, Mac OS, iOS, and Android.

4 The Classification and Selection of Hooking Approaches

In this section, we present several characteristics useful for the classification and selection of hooking tools and techniques based on specific user applications. These properties are summarized in Fig. 2.

4.1 Subroutine Type

The operating system's separation between the user and kernel processes allows to define two types of subroutine hooking: function hooking and system call hooking. Function hooking generally occurs in user-space and typically does not require root or administrator privileges. Function hooking is primarily concerned with functions from system call APIs such as the POSIX [15] or Windows API [24]. Some examples of function hooking tools include Microsoft Detours, EasyHook [8], WinAPI Override32 [22], and Frida [9].

In contrast, system call hooking occurs in kernel-space, and consequently most techniques require higher privileges. Dtrace and strace are two operating system tools that allow for this type of hooking.

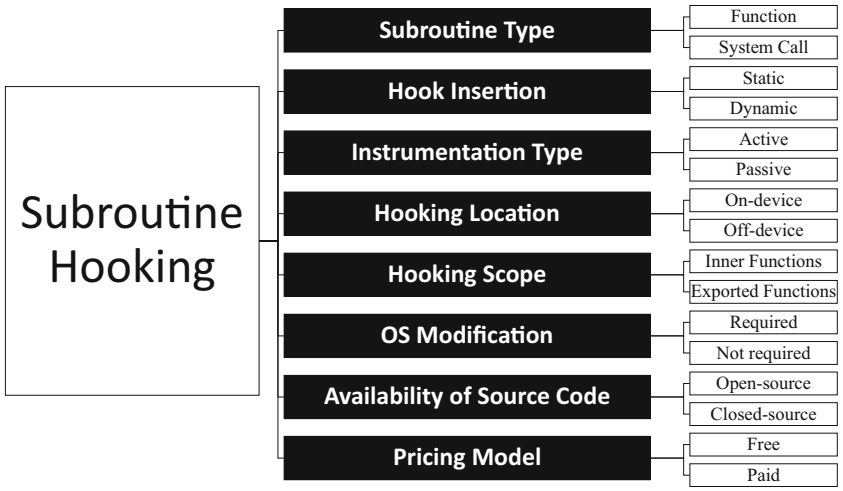


Fig. 2 A summary of the major attributes that should be considered when classifying or selecting a hooking tool or technique

4.2 Hook Insertion

Hooking techniques can also be organized based on the moment when the hooking code is inserted. With *static hooking* [65], certain adjustments must be made prior to the target program's execution. In this paper, we consider any hooking technique or tool that requires the modification of the binary executable, operating system files (including libraries), or run-time environment to be static. These changes are generally permanent. However, changes to the run-time environment can be temporary, persisting only during the execution of the process. There has been considerable research on static *caller-side rewriting* techniques which perform physical modifications (on disk) to a binary executable by dissembling it, inserting instrumentation code, and recompiling the program [30, 32, 34, 67]. However, because of the disadvantages of caller-side rewriting [60, 65], we only examine the static hooking techniques that do not modify the executable file. A common static hooking approach is to use the Linux environment variable LD_PRELOAD [12] or its macOS counterpart DYLD_INSERT_LIBRARIES [7] to alter the run-time environment. Contrary to static hooking, the term *dynamic hooking* [65] refers to the insertion of the hooking code at run-time by modifications made in memory. Dynamic hooking techniques do not require operating system modification. One of the common dynamic hooking techniques is *callee-side rewriting*, where the hook is inserted by modifying the target subroutine in memory [65]. We explain this approach in more detail in Section 5. Examples of dynamic hooking tools include Microsoft Detours, EasyHook, and Frida.

4.3 Instrumentation Type

The functionality included in the detour subroutine allows one to distinguish between two types of instrumentation: passive and active. In *passive instrumentation*, the user intercepts a subroutine, examining tracing information such as the number occurrences of a subroutine, arguments, and return value, but in general, the original functionality of the target subroutine is not altered. Typically, the detour function will include code that writes tracing information to standard output or to a file, before or after jumping to the target subroutine. For instance, Dtrace allows users to trace but not modify system calls.

In contrast, *active instrumentation* involves modifications to the functionality of the target subroutine such as changes to the input parameters or the return value of the target subroutine. Alternatively, the user may choose not to call the target subroutine, entirely overriding the behavior of the target subroutine. In this group, we can include library interposition by using LD_PRELOAD. Many hooking approaches allow for both types of instrumentation. For instance, Microsoft Detours can be used for function tracing (passive), or alternatively, it can be used to alter functions (active).

4.4 Hooking Location

Most hooking techniques are considered *on-device* or *local*, where instrumentation is accomplished on the target device and no additional devices are required. For instance, Introspy allows users to monitor selected functions on an Android device, and the results of this trace can be examined on the same device. On the other hand, *off-device* or *remote* techniques require one or more auxiliary devices in addition to the target device to perform hooking. One of the best examples of off-device hooking is Frida (see Section 5), which operates on a client-server model. With Frida, hooking a subroutine on a mobile device requires two devices, the target mobile device which acts as the server and a client, normally a desktop system.

4.5 Hooking Scope

A crucial characteristic of function hooking techniques is their hooking scope, which defines the possible functions in a program that can be instrumented. Programs may call *exported functions*, which are imported from shared libraries such as libc or the Windows API, or they can call *inner functions*, which are defined in the program itself. While nearly all techniques can hook exported functions, few techniques can instrument inner functions. Programs such as WinAPIOverride [22] can hook any function, including Windows APIs (exported functions) and inner functions.

4.6 Operating System Modification Requirement

Some hooking techniques, largely techniques which hook system calls, require operating system modification. For example, most mobile operating system hooking tools such as Cydia-Substrate [4] require changes to operating system. In most operating systems, however, users cannot modify critical operating system structures. Therefore, these types of techniques can be used on a limited number of devices and require administrative or root privileges. In contrast, hooking approaches that do not require operating system modifications will be more accessible to a greater number of devices and users. Most function hooking tools on desktop operating systems such as Microsoft Detours and WinAPI Override32 do not require operating system changes. Moreover, most desktop operating systems provide

hooking utilities such as strace and environment variable modification mechanism, both of which allow the user to perform hooking without altering the operating system.

4.7 Availability of Source Code

This criterion indicates whether the source code of the hooking tool is open source or closed source. An open-source application allows the user to further understand the mechanism behind the hooking process and adjust the code to implement desired functionality that may not be included or fully developed in the existing implementation. For instance, Easyhook and Frida are both open source. The closed-source tools such as Microsoft Detours and Nektra Spystudio [14], however, must be utilized in their current format and do not allow for individual adjustments.

4.8 Pricing Model

The price criterion defines if the tool under consideration is sold for profit or is available for free. Often, this metric can be coupled with the previous criterion of source code availability as most "for profit" software is also closed source, though, this is not always the case. All the tools and techniques discussed in this paper are free. However, two tools, Microsoft Detours and Nektra Spystudio, offer both a free version and a paid commercial edition, which offers additional features over the former.

5 Function Call Hooking Tools and Techniques

In this section, we provide an overview of various *function hooking techniques and tools* used across Windows, macOS, Linux, iOS, and Android.

5.1 Dynamic Function Hooking Approaches

5.1.1 Inline Hooking

Inline hooking is a dynamic callee-side rewriting technique that typically requires three functions: (1) a *target function*, (2) a *detour function*, and (3) a *trampoline function*. To initiate the process, one must first identify the target function. Next, a section of the target function must be overwritten with new instructions that redirect execution to a new detour function. The way a target function can be efficiently overwritten mostly depends on the specifics of the operating system and the processor architecture. Typically, on x86 systems, the first five bytes are overwritten with a five byte jump instruction (JMP) to a user-defined detour function. More efficient ways of hooking also use NOP, CALL, and PUSH/RETN instructions to overwrite original function code and provoke execution redirection [57]. While the implementation of inline hooking on the x86 architecture, especially with Windows, is straightforward, it is more difficult on ARM devices [60].

Because part of the target function's code has been overwritten, the overwritten instructions must be saved in a new function, the *trampoline*. A trampoline function must include the instructions that were overwritten and a jump to the instructions following the jump in the target function. If the trampoline is correctly implemented, calling the trampoline should work exactly the same as calling the unhooked target function. With the hook in place, execution will be redirected to the detour function, which may already be defined by a hooking tool or may need to be created by the user. This detour function allows for the implementation of passive and/or active instrumentation. In Fig. 3, we present the typical steps followed by an inline hooking implementation with both pre- and post-processing in the detour function.

Inline Hooking on Windows: Detour functions are normally encapsulated in a shared library attached to the address space of the process. In Windows, one can place the detour function(s) in a DLL file, which then needs to be injected into the process. Richter and Nasarre [53] explain several techniques for DLL injection. Inline hooking is simplified on the Windows operating system because of Hotpatching [56]. This Microsoft-developed mechanism essentially leaves space at the beginning of functions by prepending seven bytes (a mov edi, edi instruction, followed by five nop instructions) before the actual function code. This space is used to insert hooking code using JMP instructions. The primary advantage of hooking hotpatched

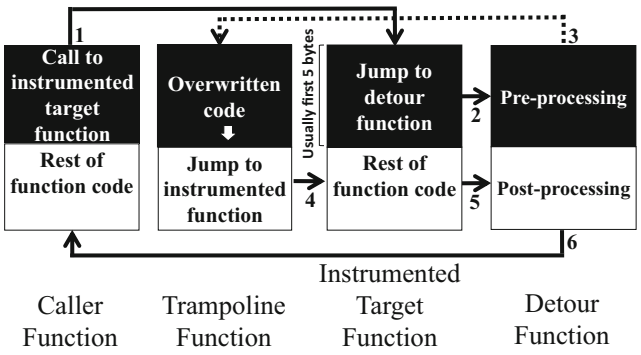


Fig. 3 An example of the normal procedure followed by an inline hooking implementation that includes both pre- and post-processing. It consists of six steps: (1) a caller function calls the target function; (2) a jump instruction in the target function redirects execution to the detour function; (3) pre-processing occurs followed by a call to the trampoline, and the trampoline executes and jumps to the remaining instructions in the target function; (4) after the target function executes, it returns to its caller, the detour function; (5) the detour function performs post-processing; and (6) the detour function returns control to the original caller function

functions is that there is no need for overwriting actual function code, so overwritten instructions are not copied to a trampoline function.

5.1.2 Import Address Table Modification

All Windows executables and DLLs are encapsulated in a data structure known as a Portable Executable (PE) [50]. A PE file contains several sections including text (program code), data, debug symbols, and two crucial tables: the Import Address Table (IAT) and the Export Address Table (EAT) [50]. Typically, Windows applications import DLLs defined by other programs or in the operating system. All imported functions along with function addresses are stored in the IAT. Similarly, all functions that the program itself wants to export are placed in the EAT. For example, a DLL would include all defined functions in the EAT so that other programs can reuse these functions.

As the name suggests, the IAT modification focuses on altering a Windows program's IAT. The critical section of the IAT is the function address, which is used by an application when a function is called. To hook a function using this technique, one needs to change the address of the target function in the IAT to the address of the detour function [35]. Whenever the target function is called, execution will be redirected to the new address in the IAT (the detour function). Before modifying the IAT, a DLL needs to be injected into the instrumented process. The DLL must contain the detour function, which must have the same function prototype (name, parameters, and return value) as the original function. There are numerous techniques for DLL injection [53].

IAT hooking has four important drawbacks. First, if the target function is an inner function, then it is defined in the program and it does not need to be imported, so there will be no entry in the IAT. Therefore, *inner functions cannot be hooked using IAT*. Second, IAT hooks are easily detected. This is an important shortcoming when performing malware identification [57]. Third, instead of loading all DLLs at load-time, some programs will load a DLL only when it is actually called inside the program. This is known as delay import and will prevent the use of IAT hooking since DLLs cannot be modified before the application execution. Lastly, some programs utilize manual loading of DLLs using Windows APIs such as *LoadLibrary* and *GetProcAddress*. When DLLs are loaded manually, there are no entries for the DLL in IAT. Consequently, there is no address that one can modify in the IAT to achieve hooking.

5.1.3 Sample Tools for Dynamic Function Hooking

The following dynamic hooking tools are grouped by operating system. We first present platform-specific tools for Windows, followed by Frida, a cross-platform tool that can be used in several OSes.

Microsoft Detours—Windows

Overview: Microsoft Detours [39] is a hooking library that uses inline hooking to intercept Win32 functions. There are two versions of Detours (Version 3.0, Build 338), a free Express edition released under the Microsoft Research Shared Source Licence Agreement (MRSSLA) for non-commercial use and a paid professional version. The Express edition can instrument 32-bit processes on x86 processors while the Professional version can instrument both 32-bit and 64-bit processes on x86, x64, IA64, and ARM systems. Detours allows for both active and passive instrumentations.

Usage: The user must package the detour function(s) in a DLL. User-defined detour functions must have the exact same call signature as the target function. Furthermore, the DLL file must include the detours.h header file, which provides access to the Detours API. Inside the DLLMain, the user must execute the *DetourAttach()* API call, which takes two arguments: the address of the target Windows API function and the detour function. This call will alter the target function allowing for API hooking. At this point, the user needs to insert the DLL with detour functions into the application. Normally, this can be accomplished by using the *withdll.exe* program, included in the Detours installation package. If the user does not want to perform API hooking, *DetourDetach()* can be used inside the DLL to restore the original program flow.

Disadvantages: Detours does not support system-wide hooks. Each process must be manually configured by injecting the DLL file that contains the detour functions. Therefore, it is difficult to hook a specific API on several processes. In addition, Detours cannot hook inner functions.

EasyHook—Windows

Overview: EasyHook (v2.7.6035.0) [8] is a free open-source hooking library for 32-bit and 64-bit Windows processes released under the MIT license. Similar to Microsoft Detours, EasyHook uses inline hooking and allows for both types of instrumentation, but it has a few advantages. Notably, EasyHook can hook unmanaged code with either unmanaged or managed functions. Managed code hooks provide important benefits such as automatic hook removal and memory management to prevent memory leaks. In addition, EasyHook guarantees thread safety by using a thread deadlock barrier.

Usage: The process to hook an API using EasyHook is very similar to the one from Microsoft Detours. The user must first define the detour function inside a DLL with

the same call signature as the API. The main method of the DLL must call the function *LocalHook.Create()*, defined in the EasyHook API. This will create the hook using the addresses of both the target and the detour functions. Subsequently, one must create an injector process, which calls *RemoteHooking.Inject()*. This function then uses the process id to insert the DLL file into the specified process, allowing the function hook to work.

Disadvantages: Like other hooking libraries, EasyHook does not support system-wide hooks and hook installation requires more effort than tools with a GUI. Moreover, EasyHook cannot inner functions.

Nekra SpyStudio API Monitor—Windows

Overview: Nektra SpyStudio API Monitor (v2.9.2 x64) [17] is a free closed-source (released under a proprietary license) hooking tool for Win32 API calls based on the Deviare open-source hooking engine. There is, however, a paid version with the same functionality as the free version, but it includes priority support from the developer. This tool can trace Windows APIs on 32-bit and 64-bit processes, only allowing for passive instrumentation.

Usage: This application presents a simple GUI with a list of running processes and a menu for target Windows API selection, organized by function category. The user can hook an already running process or a new process at the start of its execution. Moreover, the user can simultaneously instrument several APIs on multiple processes.

Disadvantages: As noted by the developer of Spystudio [17], it cannot intercept Internet Explorer (version 8 or 11) nor Chrome. During our testing, trying to hook a Chrome process resulted in Chrome unexpectedly crashing. In addition, this tool cannot hook inner functions and cannot perform active instrumentation. Lastly, did not include this program in our performance test because our test program would repeatedly terminate when we attempted to hook it using this tool.

WinAPIOverride—Windows

Overview: WinAPIOverride [22] is a free Windows hooking tool released under the GNU GPL license that allows for both active and passive instrumentations of 32-bit and 64-bit processes. As of this writing, the current version of WinAPIOverride (v6.5.2) is closed source. However, previous versions are open source, and they are still available from the developer's website. Some of the major features of WinAPIOverride include the ability to (1) hook both Windows API calls and inner function calls, allowing for the analysis or override of any function; (2) hook assembly functions; (3) hook hardware and software exceptions; and (4) view and modify register values used in function calls. In general, this tool provides two types of hooks: a monitoring hook and several overriding hooks. The monitoring hook simply logs function calls and can be modified by defining custom filters so that the user only sees selected function calls. Overriding hooks make actual modifications to the functionality of target functions, allowing for active instrumentation.

Usage: This tool can be used with the GUI interface or from the command-line terminal. Initially, the user has three different options: (1) hook an existing process, (2) initiate a process so WinAPIOverride immediately hooks it, or (3) automatically hook all starting processes. Once a process is hooked, a user must decide if function calls will be monitored or overridden. To monitor function calls, the user must specify the APIs that wished to be logged from the monitoring configuration. If the function calls need to be altered, the overriding configuration must be specified.

Disadvantages: During our testing, we experienced some sluggish application performance when tracing too many Windows APIs on various programs. In addition, when we performed our hooking performance test (Section 7), our test program became unresponsive several times.

Rohitab API Monitor—Windows

Overview: Rohitab API Monitor (v2 Alpha-r13 x64) [1] is a closed-source hooking tool released under a proprietary license. It allows for passive and/or active instrumentation of Windows API calls in 32-bit and 64-bit processes. This tool can hook target processes with three different methods: *Context Switch*, *Internal Debugger*, or *Remote Thread*. Moreover, the program displays the trace results in summary tables and call trees, which show the hierarchy of API calls.

Usage: First, the user must select the target function(s) from a database of over 13,000 Windows API definitions. Then, users must select the target process from a list of running processes to hook, or they can choose to execute a new process. Users can examine API input parameters and return values, and by setting pre-call or post-call breakpoints, these values can be modified.

Disadvantages: This program can only hook Windows API functions. Furthermore, as of this writing, this tool has not been updated in several years. Nevertheless, during our performance testing, it was very reliable and successfully instrumented our test program.

Frida—Windows, macOS, Linux, iOS, Android

Overview: Frida (v6.5.2) [9] is a multi-platform dynamic instrumentation tool for Windows, macOS, Linux,

Android, and iOS. This tool is a free open-source framework that allows for both active and passive instrumentations of exported and inner functions. It hooks functions by injecting a shared library into the target process and uses inline hooking to redirect execution. Additionally, Frida operates on a client-server model. The client must be a system running Windows, macOS, or Linux, with the Frida Python package installed on it. The Python package is in charge of executing a Javascript file containing the instrumentation code and also managing the communication with the server. The server will be the target process running on any of the supported platforms.

Usage: Note that we only focus on Android and iOS hooking using Frida since we provided several tools for hooking on Windows in previous subsections. In the case of Android and iOS, Frida requires two devices, the client device (desktop operating system) and a server device (iOS or Android) executing the target process. Typical installation of Frida on Android requires a rooted device. The user must simply download the Frida-server binary onto the device and execute it to start the server. This can be accomplished with a terminal emulator or through *Android Debug Bridge* (ADB). On the other hand, the iOS version of Frida supports two modes depending on the user's device. If the target device is jailbroken, Frida will work exactly the same as its Android counterpart, giving the user complete liberty to instrument any application. To use Frida on a jailbroken device, users must install the Frida package through Cydia [4]. Frida's other mode of operation, however, is more limited as it is designed to work with unjailbroken iOS devices. Because of iOS restrictions, users can only instrument functions using Xcode [25]. To instrument an application in Xcode, users need to download the FridaGadget.dylib and import it into an existing Xcode project. This shared library will now be loaded whenever the application is executed, allowing for function hooking.

After connecting the mobile device (iOS or Android) via USB to the client system, the user can execute scripts using a combination of Python and Javascript. In the scripts, the users must specify the target process and the target function(s). With Frida, users can hook running processes, or they can launch a new process, immediately suspending the main thread, injecting the Frida shared library into the process, and resuming execution. Passive and/or active instrumentation code must be included in the script.

Disadvantages: Using Frida with mobile operating systems requires rooted Android devices or jailbroken iOS devices. While Frida for unjailbroken for iOS devices is possible, it is far too limited because the user can only hook functions in applications for which the source code is available (Xcode project). Furthermore, Frida usage yields considerable performance overhead on all platforms, especially on desktop operating systems as our performance evaluation indicated. We provide more details on Frida's performance in Section 7.

5.2 Static Function Hooking Approaches

5.2.1 Injecting Proxy Libraries

Another approach to function hooking is the use of detour shared libraries, which contain new definitions for one or more target functions. These detour libraries are known as *proxy libraries* [28]. In this case, the user must make modifications to run-time environment, usually by changing environment variables, to force the dynamic linker to load the proxy libraries. Consequently, when a program calls a particular function, the program will call the function defined by the proxy library instead of one defined in original library. In general, the user is free to implement active and/or passive instrumentation in the detour functions.

To load shared libraries, Linux, macOS, and iOS provide environment variable directives. In the case of Windows, the operating system does not provide an explicit command to load new libraries at program run-time, but it is possible to modify the AppInit_DLLs entry in the Windows Registry to force the loading of detour DLLs [27]. This workaround, however, is restricted on Windows 8 and later versions [2].

There are two major disadvantages of using proxy libraries. First, only exported functions from shared libraries such as libc can be hooked (inner functions cannot be hooked). Second, this technique can only instrument new processes which are launched with the modified run-time environment. Already running processes cannot be hooked using proxy libraries.

5.2.2 Modifying the Operating System

Some function hooking techniques require the modification of certain operating system files or structures. This sometimes constitutes a limitation since techniques that alter the operating systems normally require high user privileges, restricting the number of user that can utilize them. For instance, operating system changes to iOS require a jailbroken iOS device, and similarly, Android requires the device to be rooted. Unfortunately, obtaining root privileges is not possible on all devices.

5.2.3 Sample Tools for Static Function Hooking

In this subsection, we describe some sample function hooking techniques that make use of run-time environment and operating system modifications.

Sysinternals Process Monitor—Windows

Overview: Sysinternals Process Monitor or ProcMon (v3.31) [56] is a free closed-source application that can monitor specific Windows APIs related to the file system such as *CreateFile*, *ReadFile*, and *LockFile*. It utilizes a Windows driver to implement its hooking mechanism and is one of the few hooking tools with a user-space front-end and kernel-mode hooking mechanism in the back-end.

Usage: Executing the program immediately presents the user with a real-time list of system calls and Registry changes from all processes. This means that the user does not select a specific process to hook. Instead, all processes are monitored by default. The user can search for specific APIs, or examine all APIs executed by a specific process. The resulting trace can be saved to a text file.

Disadvantages: Procmon is limited to Windows systems, and it only supports very few Windows APIs (file system or networking), making it unsuitable for users interested in other types of APIs. In addition, users cannot hook inner functions.

LD_PRELOAD—Linux

Overview: The environment variable LD_PRELOAD [12] allows users to force the dynamic linker to load one or more specified proxy libraries. Passive and/or active instrumentation code can be included in the proxy library.

Usage: To hook an API call, the user needs to create a proxy library. First, the user must define a detour function in a C source file with the same function prototype as the target function. This source file must be compiled as a shared library with the (.so) extension. Assuming that the shared library is in the same directory as the target executable, the proxy library can be injected into the target process using the command:

```
$ LD_PRELOAD=[library name] ./[target executable]
```

When the target process calls the target function, it will use the detour function defined in the proxy library instead of original function defined in a system library such as libc. Users, however, can use the *dlsym* function [5] inside the proxy library to call the original target function defined in a system library.

Disadvantages: The main limitations of this technique are that only new processes can be hooked and only exported functions (defined in system libraries) can be intercepted. In addition, Myers and Bazinet [51] noted that using C++ detour functions with C binaries is unreliable.

DYLD_INSERT_LIBRARIES—macOS and iOS

Overview: The DYLD_INSERT_LIBRARIES directive [7] forces the dynamic linker on macOS and iOS to load user-specified libraries first. It is the equivalent to the Linux environment variable LD_PRELOAD. Similar to its Linux cousin, it allows the user to hook any exported function, and it is up to the user to implement the detour function with the instrumentation code.

Usage: Similar to LD_PRELOAD, the user must first create a proxy library with detour function(s) that have the same function prototype as the target function. This library will need to be compiled as .dylib file, which is similar to the Linux .so shared library file. The user can force the dynamic linker to load the proxy library with the following command:

```
$ DYLD_FORCE_FLAT_NAMESPACE=1
DYLD_INSERT_LIBRARIES=[library name] ./[executable]
```

Known Issues or Disadvantages: Using shared libraries in macOS has the same disadvantages as in Linux, that is, the capacity to only hook new processes and the inability to hook inner functions. Furthermore, recent versions of macOS allow binary files to be flagged as restricted, preventing the dynamic linker from loading new shared libraries. Any program with this flag cannot be hooked with this technique.

Cydia Substrate—iOS and Android

Overview: Cydia Substrate [4] is a free closed-source instrumentation framework that allows users to install Substrate extensions, which can be used to hook inner and exported functions in jailbroken iOS devices and rooted Android devices [4]. Root privileges are required because this framework makes changes to the operating system. On iOS, Cydia Substrate will change the *launchd* manifest for *SpringBoard* so that when the device reboots, *SpringBoard* loads the Cydia Substrate dynamic library [71]. Consequently, this library will be injected into every new process, and Cydia will be able to load the user's extensions [71]. These extensions can contain detour functions for instrumentation and are compiled as shared libraries [4].

Usage: We are focusing on Cydia usage on iOS. To hook C functions, users can use Theos [21], a macOS toolkit that allows users to develop and deploy iOS applications. Users can then create C source files with Theos that contain the detour function definition(s) along with original function prototype(s), in case the user is interested in calling the original function definition. After including the substrate.h header into the C file, the user will have

access to Cydia Substrate API. The most commonly used APIs are *MSHookFunction* and *MSFindSymbol*. The former method scans the target application binary for a user-specified function and returns the symbol pointer of the function. The user must pass this symbol pointer, the address of the detour function, and a pointer to the original function to the function *MSHookFunction*, which performs the actual hooking. The user can then use Theos to create dynamic library (.dylib) from the source file and install it on the target device. By using a terminal on the device or by using SSH, the user can utilize the DYLD_INSERT_LIBRARIES environment variable to load the .dylib library.

Disadvantages: Cydia Substrate only works on jailbroken iOS. As of this writing, the latest version of iOS cannot be jailbroken. Jailbreaking requires a device running at most iOS 9.3.3. Because of all iOS installations must by signed by Apple, it is usually not possible to downgrade to jailbreak-compatible iOS versions. Furthermore, this tool is only supported on rooted Android devices with version 2.4 through 4.3.

Introspy—Android

Overview: Introspy-Android [11] is a free open-source tool primarily designed to monitor APIs related to Android security components. This tool records function calls, arguments, return values, and sends this data to both a database and the Android log. A GUI application allows the user to select which processes and APIs will be monitored. By default, Introspy monitors function calls used in hashing, keys, SSL, and inter-process communication. However, the user can add custom hooks to trace other Android APIs.

Usage: Installation requires a rooted Android device running versions 4.1 to 4.3 (Android Jellybean). The user must first install Cydia Substrate, and then, the precompiled Introspy-Android Core.apk and IntroSpy-Android Config.apk, which provides the GUI interface mentioned above. Additionally, if custom hooks are required, the user must create a new call handler class and modify the CustomHookList.java file. After this process is completed, the program must be recompiled and reinstalled on the target device. As API calls are hooked, data will be recorded in a database file, which is located in the same directory of the monitored application. The user can use the auxiliary program, Introspy Analyzer (preferably on Windows, macOS, or Linux) to convert a Introspy-Android database file to an easily digestible HTML report.

Disadvantages: As mentioned above, compatibility with Introspy-Android is very limited. Using a device with a version of Android newer than 4.3 may not be compatible. In our testing, this tool did not function correctly on a device running Android 5.1. We attribute this incompatibility to Substrate, which is only supported on Android 2.3 to 4.3. Introspy cannot work correctly without employing functionality provided by Substrate. An alternative approach to use IntroSpy-Android is to employ a rooted emulator running Android 4.3. In this case, the user can install Substrate, Introspy, and the applications that will be monitored on the emulator. This option, however, assumes that the target applications are compatible with this Android version.

Xposed Framework—Android

Overview: The Xposed Framework [26] is a free open-source tool that allows users to install modules on Android devices to hook inner and exported functions. Because Xposed requires the user to alter operation system, it will require root privileges. Xposed modifies the *app_process* executable used when initializing the *Zygote* process by adding a new library to the Java classpath to load both the framework and all necessary modules. Modules allow users to specify the pre-processing and post-processing code, which can include both passive and/or active instrumentation

Usage: To use the Xposed Framework, the user must first install the XposedInstaller.apk on the target Android device. Currently, Xposed is compatible with Android 4.0.3 to Android 4.4. There is, however, a Xposed version available for Android 5.0 or higher, but there exist incompatibility issues, and devices from certain manufacturers are not fully supported. To implement custom hooks, the users need to create a module, which overrides the methods, *beforeHookedMethod* and *afterHookedMethod*. As the names suggest, the code in the *beforeHookedMethod* will be executed before the original function is called and the function *afterHookedMethod* will be called after the original function. Alternatively, users can override *replaceHookedMethod*, which executes the specified code without ever calling the original function. The user must then compile the module and transfer it to the target device. The Xposed GUI application must be used to enable the module. Lastly, the user must restart the device, so that the module is loaded at boot time.

Disadvantages: Due to incomplete compatibility with all Android devices, the Xposed framework may not be ideal for all users. The most stable version of the Xposed Framework is designed for a rooted device with relatively old versions of Android. There are workarounds that allow running Xposed on newer Android versions, but compatibility is not guaranteed.

6 System Call Hooking Tools and Techniques

In this section, we explore two general approaches to kernel-space hooking. First, we describe several operating system utilities which can be used for hooking. Then, we examine how certain operating system modifications can allow for system call interposition.

6.1 Dynamic System Call Hooking Techniques

6.1.1 Operating System Utilities and System Calls

Operating systems provide a number of system programs (or utilities) and system calls that provide important services including file management, system status information, and program compilation, execution, and debugging [58]. Among these, there are particular tools and system calls that facilitate system call instrumentation. In this subsection, we present utilities or system calls that either directly allow for system call interception or are used by other programs to facilitate system call instrumentation. Many of these system utilities such as *strace* are cross-platform and can be found in different OSes. Other hooking tools, however, are platform-specific such as Apple Instruments.

6.1.2 Dynamic System Call Hooking Tools

For this specific topic, we begin our analysis and survey with system call hooking tools only compatible with macOS and iOS. Then, we examine the cross-platform system call, *ptrace*, and the utility *strace*, which uses ptrace to hook other system calls.

Dtrace—macOS and iOS Simulator

Overview: Dtrace [37] is a powerful tracing framework that was originally part of the Solaris operating system. Beginning with OS X Leopard, Apple ported Dtrace to macOS. Dtrace can also be used with the iOS simulator in Xcode. In general, Dtrace allows users to execute D language scripts, which are used to set either system-wide or process-specific hooking probes. When a particular probe is triggered, certain actions can be programmed. For example, a user can set a probe on the system call *open*, which will print the function arguments and return values for any process that implements this system call. Because Dtrace executes in kernel-space, no context switching is needed, significantly reducing performance overhead [37].

Usage: Creating a Dtrace script requires at least two components: the *probe description* and the *action section* [37]. The former specifies the conditions under which the actions will take place. For this, the user can specify the following conditions or parameters: (1) whether the probe will be system-wide or process-specific, (2) the module or library name if the user wants to trace functions inside a specific library, (3) the name of the system call that will be hooked, and (4) whether actions should occur before or after the system call is called [37]. A script with multiple probes can trace several system calls. In the action section, the user can specify a number of instructions or actions to be executed on the target system call such as printing the input arguments or the return value.

Disadvantages: To use Dtrace effectively, users must learn the D scripting language. If a user is not careful, certain scripts may cause significant performance degradation. In addition, only users with root privileges can use Dtrace.

dtruss—macOS

Overview: dtruss [6] is essentially a Dtrace script that makes tracing system calls easier than creating a new D language script. This tool will display system call names, arguments, and return values for a specified process, and it can monitor running processes or execute a new process with a trace.

Usage: The most basic dtruss script will print all system calls for a specific process. It can be specified as follows:

```
$ sudo dtruss -p [process id]
```

There are various options available to customize the trace results [6]. For instance, to monitor the execution of a specific system call in a process, the following command can be used:

```
$ sudo dtruss -t [system call name]
-p [process id]
```

Disadvantages: With dtruss, the user can either monitor all system calls or one specific system call. However, dtruss is not appropriate if the user needs to specifically monitor several specific system calls at the same time. In this case, a custom Dtrace script is required. Furthermore, because dtruss uses Dtrace, it requires root privileges.

Apple System Calls Instrument—macOS and iOS

Overview: The Apple System Calls Instrument [10] is specific tool within Instruments, a Xcode tool set, that traces system calls using Dtrace, allowing users to instrument a single process or all running processes on a

macOS or iOS device. With this tool, users can examine key information for each system call including input parameters, return value, source path, source library, and time-stamp.

Usage: To use Apple System Calls Instrument, users need to have Xcode installed on a computer running macOS. This tool can be executed from Xcode or through the command-line terminal. Users must select the target device, which can be either a macOS or iOS device (connected via USB), and the target process(es). After data is collected, users can filter by specific system calls or libraries. Because this tool is developed by Apple, the iOS device does not need to be jailbroken. This is an important advantage because, as we mentioned before, other system call hooking techniques do require a jailbroken iOS device.

Disadvantages: There are two main disadvantages of using the System Calls Instrument. First, as noted by Tan et al [45], this tool can only display the address of string arguments. The actual string argument is not shown. Second, because this tool relies on a USB connection with the iOS device, there are performance issues when instrumenting heavily used system calls.

Ptrace—Linux, macOS, Android, and iOS

Overview: Ptrace [16] is a system call that can be used to both monitor and/or change other system calls . It is available on several Unix-like and Unix-based operating systems including Linux, macOS, Android, and iOS. This system call allows for a parent process to observe and/or modify the execution of a child process known as the *tracee*. The ptrace user has complete flexibility to examine or modify the tracee's memory and registers.

Usage: To use ptrace, the user can create a parent process, which will hook another process. In this case, the parent process will call *fork()*, which creates a child process. This child process executes the command *PTRACE_TRACEME* command, which allows the parent to hook it. Immediately after, the child executes *execve()*, where one of the arguments is the target process. The user can also use the command *PTRACE_ATTACH* to hook an already running process. Once the hooking parent-child process is done, the user can take advantage of a variety of commands including the following:

- PTRACE_PEEKUSER: allows the parent process to examine the arguments from system calls executed in the child process.
- PTRACE_POKEUSER: notifies the parent when a system call is being executed, and it allows the parent to change system call arguments.
- PTRACE_DETACH: stops the parent process from hooking the tracee.
- PTRACE_KILL: kills the tracee.
- PTRACE_SINGLESTEP: stops tracee after every instruction.

Disadvantages: The ptrace system call has significant performance overhead performance [41]. Furthermore, if the user is simply interested in monitoring system calls, the existing command-line utility strace may be more suitable.

Strace—Linux and Android

Overview: Strace [20] is Linux utility derived from the system call ptrace that monitors all system calls and signals for a particular process. Because Android is based on the Linux kernel, it also possible to use strace on Android. With strace, users can choose to monitor a new process or trace an already running process. Strace will return system call names, arguments, and return values. Additionally, there are several options available to customize strace such as sending trace output to a file and monitoring fork processes. On Linux, strace is already installed and can be executed from the terminal. There are several versions of strace available for Android. In this work, we tested a ported version of the Linux utility, developed by Alireza Frozandeh Nezhad [18].

Usage: To install strace on Android, the user must download the binary file and copy it to */system/bin*. Installation will require a rooted device. Once installed, the user can use a terminal emulator directly on the device to execute the strace utility. Alternatively, the user can use ADB to access the ADB shell, giving the user a similar environment to the Linux shell. The user will need to use the *su* command to obtain root privilege in the shell.

The following commands apply to strace on both Linux and Android [20]. Users can use the following basic command to print all system call names and arguments for a particular process:

```
$ strace [-options] [program executable]
```

Some the most useful options include the following:

- e: trace specific system call(s)
- o: save the trace to a file
- p: monitor a process that is already running using process id
- c: count system calls and display summary

Disadvantages: Because strace uses ptrace, one of its main disadvantages is its performance overhead. Depending on the hooked process, the performance degradation can be significant, something that is particularly concerning on resource-limited Android devices.

Furthermore, strace cannot trace multiple processes at once. In addition, a rooted Android device is needed to use the ported version of strace.

6.2 Static System Call Hooking Techniques

6.2.1 Operating System Kernel Modification

An alternative technique for system call hooking is the modification of certain structures in an operating system's kernel. Similar to function hooking, the general strategy is to intercept system calls, redirecting them to detour system calls, which contain passive and/or active instrumentation code. The most common kernel modification hooking technique is to make changes related to the system call table. Wampler [63] identifies three mechanisms by which kernel-level rootkits modify system call execution. While these approaches are discussed only in the context of malicious usage by malware, we can also use these techniques in legitimate ways by hooking system calls. First, addresses in the system call table can be replaced with the addresses of detour system calls. This is known as *system call table modification*. The second strategy is *system call target modification*, where the target system calls are overwritten with a jump instruction to a detour system call. This technique is very similar to inline hooking (discussed in Section 5), but in this particular case, the system call modifications are done on the disk, instead of in memory. Lastly, we can use *system call table redirection*, where we modify the kernel to use a detour system call table instead of the original one. This user-defined table includes entries for target system calls that point to detour system calls.

In general, one can modify the kernel directly by modifying operating system files. This approach is not ideal, however, because changes are permanent and require kernel recompilation to take effect. In addition, most operating systems including Windows, macOS, iOS, and Android restrict users from making any direct modification to the kernel. In contrast, most x86-64 operating systems including Windows, macOS, and Linux support *loadable kernel modules* (LKMs), which allow users to modify kernel operations. Crucially, LKMs can be used to implement kernel-space hooking. For instance, Edger et al. [33] installed an LKM on an Android device to hook system calls. In addition, unlike direct kernel modifications, LKMs can be implemented without kernel recompilation and can be easily loaded or unloaded. In macOS and Windows, a LKM is referred as a kernel extension (kext) and kernel-mode driver, respectively.

Operating systems place different restriction for LKM privileges which influences the kind of kernel-mode hooking techniques that are available on each operating system. Even with LKMs, changes to critical structures such as the system call table are not possible on most operating systems. For instance, on all 64-bit versions of Windows, Microsoft has implemented *Kernel Patching Protection* (KPP), also known as PatchGuard, which guarantees the integrity of critical operating system structures such as the *System Service Descriptor Table* (SSDT) and the *Interrupt Descriptor Table* (IDT) by preventing modification to these structures [62]. Even Linux, whose open-source nature has traditionally given users the greatest flexibility to modify the kernel, restricts changes to the system call table on its latest versions. In the following subsection, we look at hooking techniques on earlier versions of the Linux kernel, which allowed modifications to kernel structures.

6.2.2 Specific Techniques and Tools for Implementing Hooks in the Kernel

In this subsection, we discuss how to manually modify the Linux kernel to implement system call hooking and examine Strace for NT, a static system call hooking tool for Windows.

Linux System Call Modification

Overview: One system call hooking technique on Linux is the modification of the system call table by adding one or more detour system calls to the table. Then, one can swap system calls numbers between the target and detour system calls so that when the target system call is called, execution will be redirected to the detour function. The user must implement passive and/or active instrumentation in the new system call.

Usage: The system call table can be modified directly or with an LKM. Here, we focus on the former approach. To implement the new system call, the user must first create a new C source file that will define a system call with the same prototype as the target system call. This detour system call must include the instrumentation code. Next, the new system call must be added to several kernel files including syscall_table.32.s, unistd_32.h, syscalls.h, and the kernel Makefile. Users will need root privileges to modify these files. Crucially, users need to add the new system call to the file syscall_table.32.s and switch the position of the target system call with the detour system to ensure redirection. Finally, the kernel needs to be recompiled so that the changes can take effect. Note that these instructions were successfully used on Linux kernel version 2.6.36. For newer versions of the Linux kernel, the procedure may differ.

Disadvantages: The primary disadvantage, as mentioned before, is the fact that the system call table modification

is restricted on most operating systems. Even with the operating system support, there are still a variety of pitfalls with the use of this technique. First, the user can only hook one system call with each detour system call. Each system call that is hooked must have a detour system call. Therefore, this technique is not very practical if the user needs to instrument several system calls. Another issue is that kernel modifications are permanent and affect all processes. Depending on the user's needs, this may be an advantage or disadvantage. Furthermore, making modifications to the kernel is delicate, and certain changes can make the system unstable or even unusable.

Strace for NT

Overview: Strace for NT (v0.3) [19] is system call hooking tool for the Windows NT4, 2000, and XP operating systems. This tool offers similar functionality to the Linux strace, allowing users to trace system calls. It uses a kernel driver to modify function pointers in the system call table, redirecting execution to a detour system call [28].

Usage: This tool can be executed from the command-line on compatible operating systems. Similar to strace for Linux and Android, the following options are available:

- -o: To output the trace to a file.
- -e: To trace specific system calls.
- -p: To monitor the specified process id.

Disadvantages: Because this tool requires kernel modification, it is not compatible with current versions of Windows. As of this writing, this program has not been updated recently and only older versions of Windows (NT4, 2000, and XP) are supported.

Finally, Table 1 summarizes and compares all the system and function call hooking tools and techniques (19 in total) based on the different categories defined in Section 4 and also, based on the details about each different tool or technique provided in this section and in Section 5. Again, the selection of the right tool or technique will greatly depend on the user's specific application and the environment characteristics on which the hooking/instrumentation will take place.

7 Hooking Performance

In this section, we analyze the performance of different hooking tools and techniques across all major operating systems.

7.1 Methodology

For our performance test, we created a similar test program for each operating system that performed the following operations:

- Creates 10,000 text files.
- Opens each file, writes the message "Hooking Testing," and closes the file.
- Opens each file, reads the contents into a buffer, and closes the file.
- Deletes all 10,000 files.

Each operation was performed inside a loop, for a total of four loops, each of which iterated 10,000 times. After the test program finished executing all the operations, it reported the following statistics:

- Physical memory size: memory usage in the actual hardware (RAM).
- Virtual memory size: memory allocation in a virtual address space.
- CPU user time: time spent executing in user-space.
- CPU system time (kernel time): time spent in the kernel, servicing system calls.
- Wall-clock time: real time measured from the moment the process begins execution until all file operations are completed.

The test program was first executed without any instrumentation. Then, we used different hooking tools and techniques to hook the test program and quantified the overhead by measuring the difference in performance between the cases where the hooking tools were utilized and the cases were no instrumentation was performed. The test program was launched by the hooking tool, so that it could be instrumented from the start of its execution. In the subsequent subsections, we explain the methodology followed for each operating system.

7.1.1 Windows

To measure memory utilization in Windows, we used *GetProcessMemoryInfo()* in the Process Status API (PSAPI) to obtain the peak working set and peak page file usage (commit charge). For CPU utilization, we used *GetProcessTimes()* to obtain kernel and user CPU time. Finally, to measure wall-clock time, we used *QueryPerformanceCounter()* to obtain a start and end time.

Using three different hooking tools, we instrumented the following Windows API functions: *CreateFile*, *WriteFile*, *ReadFileEx*, and *DeleteFile*. The tools were configured as follows:

- *Rohitab API Monitor (v2 Alpha-r13)*: We first selected the four APIs from the API filter, and with option to

Table 1 Summary of system and function call hooking tools and techniques based on the proposed classification features

	Microsoft Detours	EasyHook	Nektra SpyStudio	WinAPI Override32	Rohitab API Monitor	Frida	Sysinternals Process Monitor	LD_PRELOAD	DYLD_INSERT_LIBRARIES	Cydia-Substrate	Introspy	Xposed Framework	Dtrace	dtruss	Apple System Calls Instrument	ptrace	strace	Linux System Call Table Modification	Strace for NT
Subroutine Type / Hooking Scope																			
Inner Functions	○	○	○	●	○	●	○	○	○	●	○	●	○	○	○	○	○	○	○
Exported Functions	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	○	○	○
System Calls	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●	●	●	●	●
Hook Insertion																			
Dynamic	●	●	●	●	●	●	○	○	○	○	○	○	●	●	●	●	●	○	○
Static	○	○	○	○	○	○	●	●	●	●	●	●	○	○	○	○	○	●	●
Instrumentation Type																			
Passive	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Active	●	●	○	●	●	●	○	●	●	●	○	●	○	○	○	●	○	●	○
Hooking Location																			
On-device	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Off-device	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○
User Interface																			
GUI	○	○	●	●	●	○	●	○	○	●	●	●	○	○	●	○	○	○	●
Command Line Interface	○	○	○	●	○	●	○	●	●	○	●	○	●	●	●	●	●	●	○
Hooking Library	●	●	○	○	○	●	○	○	○	●	○	●	○	○	○	○	○	○	○
OS																			
Requires OS Modification	○	○	○	○	○	○	○	○	○	●	●	●	○	○	○	○	○	●	●
Windows	●	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	●
Linux	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	●	●	●	○
macOS	○	○	○	○	○	●	○	○	●	○	○	○	●	●	●	●	○	○	○
iOS	○	○	○	○	○	●	○	○	●	●	○	○	●	○	●	●	○	○	○
Android	○	○	○	○	○	●	○	○	○	●	●	●	○	○	○	●	●	○	○
Source-Code																			
Closed-Source	●	○	●	○	●	○	●	○	○	●	○	○	○	○	●	○	○	○	○
Open-Source	○	●	○	●	○	●	○	●	●	○	●	●	●	●	○	●	●	●	●
License																			
ASL	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○
GNU GPL	○	○	○	●	○	●	○	○	○	○	●	○	○	○	○	○	○	○	○
MIT	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Proprietary	●	○	●	○	●	○	●	○	○	●	○	○	○	○	●	○	○	○	●
Part of OS	○	○	○	○	○	○	○	●	●	○	○	○	●	●	○	●	●	●	○
Pricing Model																			
Free	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Paid	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

monitor a new process (Ctrl + M), we selected our test program executable. We attached to the executable using the context switch option.

– *WinAPIOverride64 (v6.5.2)*: We selected the option *Attach at application startup* and added our executable to the application path. Once our program was in a paused state, we loaded our API monitoring configuration, which included the four Windows APIs. Then, we allowed our executable to continue execution.
– *Frida (v8.1.10)*: We created a Python file that first launched our executable using *frida.spawn()* and before our executable continued execution, we loaded a Javascript script that was used to hook the four Windows APIs. This script used *Module.findExportByName()* to obtain the address of each API. This address is used by *Interceptor.attach()* to intercept each API whenever it is called. Inside *Interceptor.attach()*, we defined *onEnter* and *Onleave*, to record API arguments and return value, respectively. After the script was loaded, the function *frida.resume()* allowed our executable to resume execution. This Python file was executed from the Windows Command Prompt.

These experiments were conducted on a Windows 10 machine with Intel Core i5-4590 3.30-GHz CPU and 16 GB of RAM.

7.1.2 Linux

To examine memory usage on Linux, we accessed the file proc/[pid]/status in the proc file system. In this file, we read the value of VmPeak and VmHWM for the virtual and physical memory usage, respectively. For CPU time, we used the function *times()* defined in *sys/times.h*, which returns a tms structure with CPU utilization and system time. Finally, to measure wall-clock time, we used *clock()* defined in time.h to obtain a start time (before file operations) and end time (after file operations).

In the case of Linux, we analyzed the performance of two function hooking techniques, the environment variable LD_PRELOAD and Frida. Our evaluation is the result of instrumenting the following functions defined in libc: *fopen()*, *fprintf()*, *fscanf()*, *fclose()*, and *remove()*. In addition, we tested the system call hooking utility strace by examining the following system calls: *open()*, *write()*, *read()*, *close()*, and *unlink()*. The tools were configured as follows:

- *Environment Variable LD_PRELOAD*: We created a proxy library that included definitions for five different detour functions for each target function. Each detour function printed the function arguments, calling the target function with its corresponding arguments and printing the return value of the target function. The library file was a C file that was compiled with gcc as a shared library. By using LD_PRELOAD, we guaranteed that this shared library would be loaded at run-time every time we executed our test program.
- *Frida (v8.1.10)*: Similar to our Windows test, we executed a Python file from the Linux terminal. This file launched our executable using *frida.spawn()*, and loaded a script, which instrumented all five libc functions, printed function arguments, and return values to the console. The script also used *Module.findExportByName()* to obtain the address of each target function and *Interceptor.attach()* to hook each function. After the script was loaded, *frida.resume()* allowed our test program to continue its execution.
- *strace*: We used strace with the command:

```
strace -e trace=open,write,read,
close,unlink [executable]
```

  With this command, strace utility instrumented the five specified system calls, displaying input parameters and return value for each system call.

Linux's experiments were conducted on an Ubuntu Linux 14.04 system with two processor cores Intel i5-5257U 2.7 GHz CPU and 1 GB of RAM.

7.1.3 macOS

For macOS, we followed a similar approach to Linux. However, because macOS lacks the proc file system, we had to obtain our measurements differently. We used the function *getrusage()*, which fills a structure containing the CPU time and *task_info()*, which returns the resident and virtual memory size. Like Linux, wall-clock time is measured using *clock()*. We tested function hooking performance using Frida and with a shared library using the environment variable DYLD_INSERT_LIBRARIES, which is similar to Linux LD_PRELOAD. We instrumented the following functions: *fopen()*, *fprintf()*, *fscanf()*, *fclose()*, and *remove()*. Furthermore, we evaluated system call hooking performance with the utility Dtrace, tracing the following system calls: *open_nocancel()*, *write_nocancel()*, *read_nocancel()*, *close_nocancel()*, and *unlink()*. The tools were configured as follows:

- *Frida (v8.1.10)*: Because of Frida's cross-platform compatibility, we reused the same Python file used in the Linux testing. However, to locate the five target functions, we had to modify *Module.findExportByName()* so that Frida searches for the address of the target functions in the library *libc++.1.dylib*. Similar to the Linux version, the Python file spawned the test program in a suspended state, loaded the instrumentation script, and resumed the program.
- *Environment Variable DYLD_INSERT_LIBRARIES*: We used the same library file from the Linux version for tracing the five functions described above. Each function's arguments and return values were printed to the console. We compiled this library as a macOS dynamic library (.dylib).
- *Dtrace*: We created a Dtrace script using the D programming language, which instruments the five target system calls on entry and exit, logging function arguments and the return value. As opposed to using DYLD_INSERT_LIBRARIES, executing dtrace scripts requires higher administrative privileges. This script was executed from the terminal.

The experiments were conducted a macOS Sierra (Version 10.12.1) system with an Intel Core i5-5257U 2.7-GHz CPU and 8 GB of RAM.

7.1.4 iOS

Our iOS test program used the function *task_info()* to obtain physical memory and virtual memory size. Furthermore, we utilized *getrusage()* to measure CPU user time. In this case, we tested the function hooking tool Frida, instrumenting

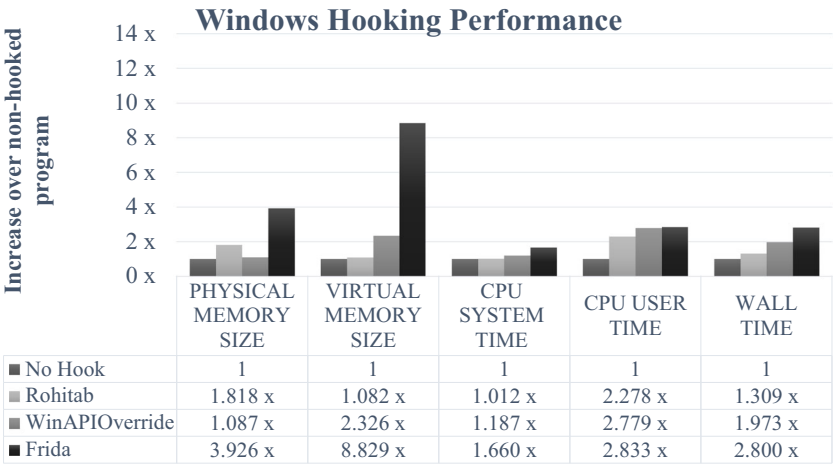


Fig. 4 Performance overhead for different hooking tools and techniques on Windows

the following functions: *read()*, *close()*, *write()*, *open()*, and *removefile()*. The tool was configured as follows:

– *Frida (v8.1.10)*: Similar to the previous OSes, we created a Python that instrumented the five target functions. These functions were located in *libsystem_kernel.dylib*, and it was necessary to specify this library in *Module.findExportBy- Name()* to locate the target functions.

These experiments were conducted on a jailbroken iPhone 6S Plus with 1.85-GHz A9 CPU with 2 GB of RAM, running iOS version 9.3.3.

7.2 Android

Like Linux, Android allowed us to use the proc file system to examine performance metrics. In *proc/[pid]/- status*, we read the values of VmPeak and VmHWM to obtain the virtual and physical memory usage, respectively. We obtained CPU time by examining the 14th and 15th values in *proc/[pid]/status*, which gave us user and system time, respectively. Lastly, we measured wall-clock time using *System.currentTimeMillis()*.

We tested function hooking using Frida, instrumenting the following libc functions: *fopen()*, *fclose()*, *read()*, *write()*, and *remove()*. In addition, we traced the following system calls using the utility strace: *openat()*, *unlinkat()*, *close()*, *write()*, and *read()*. The tools were configured as follows:

– *Frida (v8.1.10)*: We created a Python script that located and intercepted the target functions in the libc library, printing the function arguments and return values.
– *Ported strace*: We monitored the five target system calls using a ported version of strace. We launched and traced our test program from the command-line with

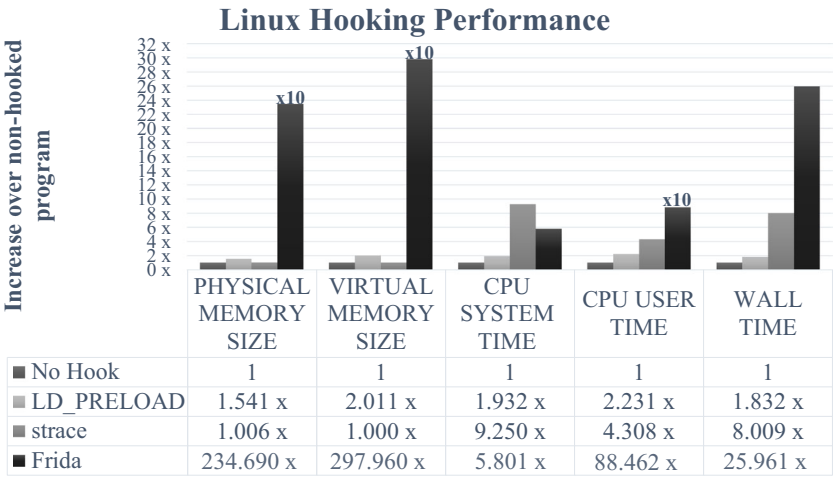


Fig. 5 Performance overhead for different hooking tools and techniques on Linux

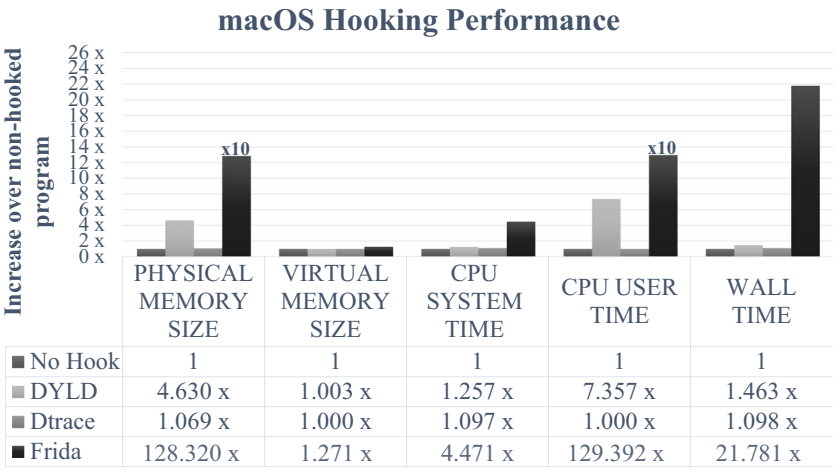


Fig. 6 Performance overhead for different hooking tools and techniques on macOS

ADB, using the *am start* command to execute the main activity class of our test program.

These experiments were conducted on rooted Android phone with a Mediatek 6580 Quad Core 1.3-GHz CPU and 768 MB of RAM with Android version 5.1.

7.3 Performance Overhead by Operating System

We analyzed the performance of the selected hooking tools and techniques on each operating system by comparing their experimental results to a non-instrumented execution of our test program.

- *Windows*: For Windows, we examined the performance of three function hooking techniques. Figure 4 shows the results of our experiments. Of the three dynamic instrumentation tools, Rohitab was the most efficient tool in terms of CPU usage, with a overhead increase of 1.01x and 2.28x for the CPU system time and user time, respectively. WinAPIOverride consumed the least physical memory, but had a larger virtual memory size than Rohitab. Frida consumed the most memory of all the tools included and increased wall-time by 2.8x, a significant performance impact. In general, because these programs are function hooking tools, they had larger impact on CPU user time than CPU system time.
- *Linux*: As Fig. 5 indicates, using the environment variable LD_PRELOAD had much better performance than Frida in terms of both memory and CPU utilization. Because Frida is a dynamic instrumentation tool, it requires more resources than LD_PRELOAD, which constitutes a static hooking technique. Furthermore, the strace utility had a negligible effect on memory usage, but significantly increased CPU and wall-time. Because strace intercepts system calls, it had a larger effect on CPU system time than user time.
- *macOS*: The results for macOS are displayed in Fig. 6. Similar to Linux, we compared two function hooking

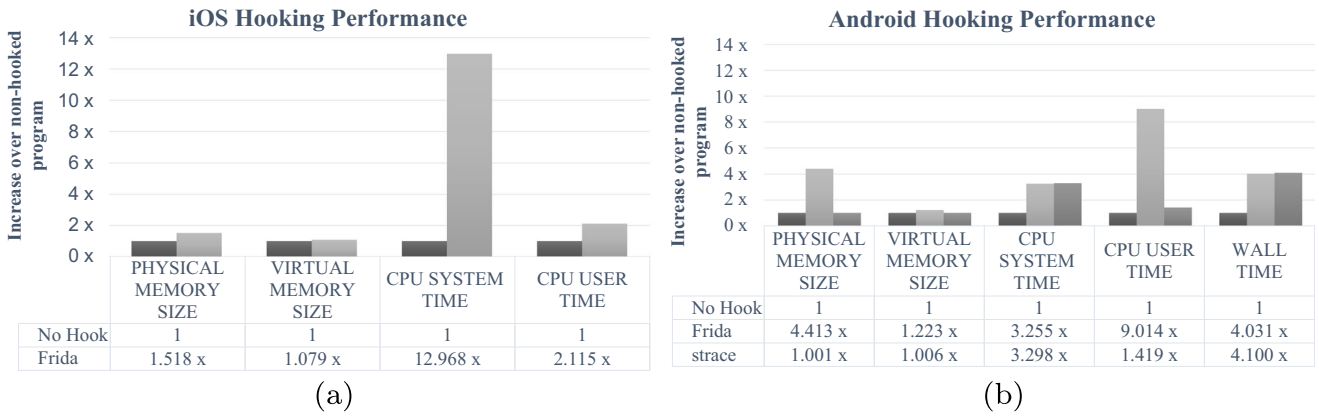
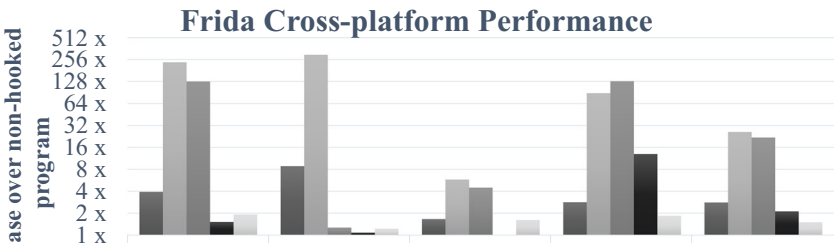


Fig. 7 Performance overhead for different hooking tools and techniques on iOS (**a**) and Android (**b**)

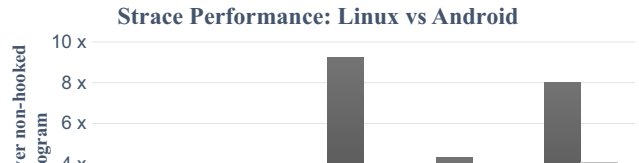


	PHYSICAL MEMORY SIZE	VIRTUAL MEMORY SIZE	CPU SYSTEM TIME	CPU USER TIME	WALL TIME
Windows	3.93 x	8.83 x	1.66 x	2.83 x	2.80 x
Linux	234.69 x	297.961 x	5.80 x	88.46 x	25.96 x
macOS	128.32 x	1.268 x	4.47 x	129.39 x	21.78 x
iOS	1.52 x	1.079 x	0	12.97 x	2.11 x
Android	1.92 x	1.223 x	1.61 x	1.84 x	1.50 x

Fig. 8 Cross-platform performance experimental results for Frida (all the platforms)

tools and one system call utility. Using the directive DYLD_INSERT_LIBRARIES proved to be much more efficient in all metrics than Frida. Interestingly, the dtrace utility incurred less overhead than modifying the run-time environment even though dtrace hooks systems calls rather than functions.

– *iOS*: On iOS, we focused exclusively on the dynamic hooking tool, Frida. The results are shown in Fig. 7a. Compared to other opearting systems, Frida performed better with a 1.52x and 1.08x increase in physical and virtual memory respectively. However, there was a concerning CPU user time increase of 12.97x and a 2.11x rise in wall-time.
– *Android*: On Android, we tested function hooking tool Frida and strace for system call hooking. As Fig. 7b shows, Frida presented a substantial memory, CPU time, and wall-time increase, especially in physical memory size. While strace's memory usage was negligible, it significantly affected the CPU system time and wall-time.



	PHYSICAL MEMORY SIZE	VIRTUAL MEMORY SIZE	CPU SYSTEM TIME	CPU USER TIME	WALL TIME
Linux	1.01 x	1.00 x	9.25 x	4.31 x	8.01 x
Android	1.00 x	1.006 x	3.30 x	1.42 x	4.10 x

Fig. 9 Cross-platform performance experimental results for strace (Linux and Android)

7.4 Cross-platform Performance Overhead Comparison

We examined the performance of the following cross-platform tools: Frida, strace, and the dynamic linker environment variables on macOS and Linux.

– *Frida*: Frida constitutes a true cross-platform hooking tool. Figure 8 shows the results of comparing Frida performance between different platforms. In terms of desktop operating systems, Frida for Windows demonstrated the best performance. Linux and macOS were the worst-performing operating systems with significant increases across all metrics, especially memory utilization and CPU user time. Android was the best performing platform in terms of CPU system time, CPU user time, and wall-time. iOS, however, consumed less memory. Overall, *Frida performed better on mobile devices than on desktop operating systems*.
– *strace*: The strace utility is available on Linux and Android. Figure 9 shows the results after comparing the performance of strace on both platforms. Memory utilization was nearly the same on both, but there was a major discrepancy in other metrics, with Android showing much better performance. Compared to Android, strace on Linux required nearly triple the CPU system time and more than double the CPU user time, resulting in almost double the wall-time.
– *Environment Variables*: We can hook functions by modifying the run-time environment variables on both Linux and macOS, using LD_PRELOAD and DYLD_INSERT_LIBRARIES, respectively. As Fig. 10 indicates, both implementations perform better on certain metrics. On Linux, there was less physical memory usage and CPU user time overhead. On the other hand, macOS had better results in terms of virtual memory utilization, CPU system time, and wall-time. We

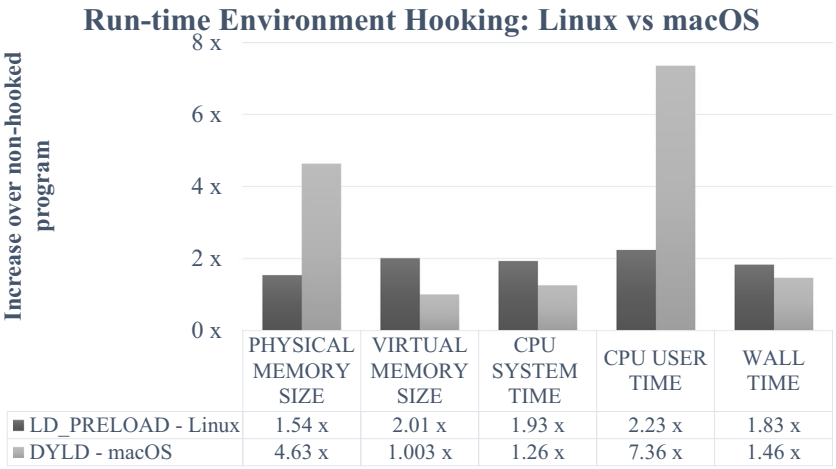


Fig. 10 Cross-platform performance experimental results for run-time environment hooking (Linux and macOS)

attribute the variance in performance to differences in how each operating system handles shared libraries at load-time.

8 Conclusions and Future Work

Hooking tools and techniques constitute a key element in several security applications like malware detection and compromised software behavior analysis. In this paper, we presented several criteria for the classification and selection of hooking tools and techniques as well as an examination of the most popular hooking approaches available for the major operating systems. Furthermore, we measured the overhead introduced by some of the most popular hooking techniques, by comparing memory utilization, CPU utilization, and process wall-clock time in all major operating systems. The results revealed that Frida was, by far, the most resource-consuming function hooking tool across all the platforms. Interestingly, Frida performed better in the case of Windows than all other desktop (Linux and macOS) and mobile (Android and iOS) operating systems. Finally, system call hooking proved to be more resource-consuming than function call hooking. To the best of our knowledge, this is the first extensive work that examines subroutine hooking tools and techniques in both user-space and kernel-space across the five most popular operating systems: Windows, Linux, macOS, iOS, and Android.

Acknowledgments This work was partly supported by the US NSF-CAREER-CNS-1453647, US DOE DE-OE0000779, and US NSF-REU-CNS-1461119. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the funding agencies.

References

1. API monitor. http://www.rohitab.com/apimonitor. [Online; accessed 22-December-2016]
2. AppInit DLLs and secure boot. https://msdn.microsoft.com/en-us/library/windows/desktop/dn280412(v=vs.85).aspx. [Online; accessed 20-December-2016]
3. BitBlaze: binary analysis for computer security. http://bitblaze.cs.berkeley.edu/. [Online; accessed 25-June-2017]
4. Cydia substrate. http://www.cydiasubstrate.com. [Online; accessed 20-December-2016]
5. dlsym(3)—linux man page. http://man7.org/linux/man-pages/man3/dlsym.3.html. [Online; accessed 16-December-2016]
6. dtruss(1m)—Mac OS X man pages. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/dtruss.1m.html. [Online; accessed 16-December-2016]
7. DYLD(1)—Mac OS X man pages. https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/dyld.1.html. [Online; accessed 16-December-2016]
8. EasyHook. https://easyhook.github.io. [Online; accessed 22-December-2016]
9. Frida. https://www.frida.re. [Online; accessed 19-December-2016]
10. Instruments user guide. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/. [Online; accessed 20-December-2016]
11. Introspy-Android. http://isecpartners.github.io/Introspy-Android/. [Online; accessed 19-December-2016]
12. ld.so(8)—linux man page. http://man7.org/linux/man-pages/man8/ld.so.8.html. [Online; accessed 22-December-2016]
13. Microsoft detours. https://www.microsoft.com/en-us/research/project/detours/. [Online; accessed 02-July-2017]
14. Nektra: advanced computing. http://www.nektra.com/products/spystudio-api-monitor/. [Online; accessed 02-July-2017]
15. POSIX.1-2008 The pen Group Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/. [Online; accessed 20-January-2017]
16. ptrace—linux manual page. http://man7.org/linux/man-pages/man2/ptrace.2.html. [Online; accessed 14-December-2016]
17. SpyStudio Overview. http://www.nektra.com/products/spystudio-api-monitor/. [Online; accessed 18-December-2016]
18. strace for android. https://github.com/alireza7991?tab=repositories. [Online; accessed 20-December-2016]
19. Strace for NT. http://seriss.com/people/erco/ftp/winnt/strace/. [Online; accessed 10-January-2017]
20. strace(1)—linux man page. https://linux.die.net/man/1/strace. [Online; accessed 14-December-2016]
21. Theos/setup. http://iphonedevwiki.net/index.php/Theos/Setup. [Online; accessed 20-December-2016]
22. WinAPIOverride. http://jacquelin.potier.free.fr/winapioverride32/. [Online; accessed 18-December-2016]

23. Windows API index. https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx. [Online; accessed 20-January-2017]
24. Windows API index. https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx. [Online; accessed 02-July-2017]
25. Xcode 8. https://developer.apple.com/xcode/. [Online; accessed 20-December-2016]
26. Xposed module repository. http://repo.xposed.info. [Online; accessed 19-December-2016]
27. Abimbola AA, Munoz JM, Buchanan WJ (2006) Nethost-sensor: monitoring a target host's application via system calls. Inf Secur Tech Rep 11(4):166–175
28. Andersson S, Clark A, Mohay G, Schatz B, Zimmermann J (2005) A framework for detecting network-based code injection attacks targeting Windows and UNIX. In: Proceedings of the 21st annual computer security applications conference, ACSAC '05. IEEE Computer Society, Washington, DC, USA, pp 49–58
29. Babun L, Aksu H, Uluagac AS (2017) Identifying counterfeit smart grid devices: a lightweight system level framework. In: 2017 international conference on communications (ICC)
30. Backes M, Gerling S, Hammer C, Maffei M, von Styp-Rekowsky P (2013) Appguard: enforcing user requirements on Android apps. In: Proceedings of the 19th international conference on tools and algorithms for the construction and analysis of systems, TACAS'13. Springer, Berlin, pp 543–548
31. Bovet D, Cesati M (2005) Understanding the Linux kernel Oreilly & Associates Inc
32. Davis B, Chen H (2013) Retroskeleton: retrofitting Android apps. In: Proceeding of the 11th annual international conference on mobile systems, applications, and services, MobiSys '13. ACM, New York, pp 181–192
33. Eder T, Rodler M, Vymazal D, Zeilinger M (2013) ANANAS - A framework for analyzing android applications. In: 2013 eighth international conference on availability, reliability and security (ARES), pp 711–719
34. Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: Proceedings of the 20th USENIX conference on security, SEC'11. USENIX Association, Berkeley, pp 21–21
35. Holy Father (2004) Hooking Windows API-Technics of hooking API functions on Windows. CodeBreakers-Journal, 1(2)
36. Garfinkel T (2003) Traps and pitfalls practical problems in system call interposition based security tools. In: Proceedings of network and distributed systems security symposium, pp 163–176
37. Gregg B, Mauro J (2011) DTrace: dynamic tracing in oracle Solaris, Mac OS X and freeBSD, 1st edn. Prentice Hall Press, Upper Saddle River
38. Guo PJ, Engler D (2011) Using system call interposition to automatically create portable software packages. In: Proceedings of the 2011 USENIX conference on USENIX annual technical conference, USENIXATC'11. USENIX Association, Berkeley, pp 21–21
39. Hunt G, Brubacher D (1999) Detours: binary interception of Win32 functions. In: Proceedings of the 3rd conference on USENIX windows NT symposium - Volume 3, WINSYM'99. USENIX Association, Berkeley, pp 14–14
40. Jeong Y, Lee H, Cho S, Han S, Park M (2014) A kernel-based monitoring approach for analyzing malicious behavior on Android. In: Proceedings of the 29th annual ACM symposium on applied computing, SAC '14. ACM, New York, pp 1737–1738
41. Keniston J, Mavinakayanahalli A, Panchamukhi P, Prasad V (2007) Ptrace, utrace, uprobes lightweight, dynamic tracing of user apps. In: Proceedings of the 2007 Linux symposium, pp 215–224
42. Kim S-W (2012) Intercepting system API calls. https://software.intel.com/en-us/articles/intercepting-system-api-calls. [Online; accessed 18-December-2016]
43. Kim T, Zeldovich N (2013) Practical and effective sandboxing for non-root users. In: Presented as part of the 2013 USENIX annual technical conference (USENIX ATC 13). USENIX, San Jose, pp 139–144
44. Liu ST, Huang Hc, Chen YM (2011) A system call analysis method with mapreduce for malware detection. In: 2011 IEEE 17th international conference on parallel and distributed systems, pp 631–637
45. Zhao F, Tan L, Zhang X (2012) Advanced operating and distributed system android and iOS platform study final report
46. Ligh MH, Adair S, Hartstein B, Richards M (2011) Malware analyst's codebook and DVD: tools and techniques for fighting malicious code. Wiley, New York
47. Madani P, Vlajic N (2016) Towards sequencing malicious system calls. In: 2016 IEEE conference on communications and network security (CNS), pp 376–377
48. Marhusin MF, Larkin H, Lokan C, Cornforth D (2008) An evaluation of API calls hooking performance. In: Proceedings of the 2008 international conference on computational intelligence and security - volume 01, CIS '08. IEEE Computer Society, Washington, pp 315–319
49. Mehdi B, Ahmed F, Khayyam SA, Farooq M (2010) Towards a theory of generalizing system call representation for in-execution malware detection. In: 2010 IEEE international conference on communications, pp 1–5
50. (2015). Microsoft. Visual studio, Microsoft portable executable and common object file format specification. Technical report, Microsoft
51. Myers DS, Bazinet AL (2004) Intercepting arbitrary functions on Windows, UNIX, and Macintosh OS X platforms. Technical report, Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies University of Maryland
52. Qin F, Wang C, Li Z, Kim Hs, Zhou Y, Wu Y (2006) Lift: a low-overhead practical information flow tracking system for detecting security attacks. In: 2006 39th annual IEEE/ACM international symposium on microarchitecture (MICRO'06), pp 135–148
53. Richter JM, Nasarre C (2007) Windows via C/C++, 5th edn. Microsoft Press, USA
54. Rubanov VV, Shatokhin EA (2011) Runtime verification of linux kernel modules based on call interception. In: 2011 fourth IEEE international conference on software testing, verification and validation, pp 180–189
55. Russello G, Jimenez AB, Naderi H, van der Mark W (2013) FireDroid: hardening security in almost-stock android. In: Proceedings of the 29th annual computer security applications conference, ACSAC '13. ACM, New York, pp 319–328
56. Russinovich ME, Solomon DA, Ionescu A (2012) Windows internals, Part 1: covering windows server 2008 R2 and Windows 7, 6th edition. Microsoft Press, USA
57. Mohd Shaid SZ, Maarof MA (2015) In memory detection of Windows API call hooking technique. In: 2015 international conference on computer, communications, and control technology (i4CT), pp 294–298
58. Silberschatz A, Galvin PB, Gagne G (2008) Operating system concepts, 8th edn. Wiley Publishing, New York
59. Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, Liang Z, Newsome J, Poosankam P, Saxena P (2008) BitBlaze: a new approach to computer security via binary analysis. In: Proceedings of the 4th international conference on information systems security. Keynote Invited paper., Hyderabad, India

60. Sun M, Zheng M, Lui JCS, Jiang X (2014) Design and implementation of an android host-based intrusion prevention system. In: Proceedings of the 30th annual computer security applications conference, ACSAC '14. ACM, New York, pp 226–235
61. Sze WK, Sekar R (2015) Provenance-based integrity protection for windows. In: Proceedings of the 31st annual computer security applications conference, ACSAC 2015. ACM, New York, pp 211–220
62. Vogl S, Pfoh J, Kittel T, Eckert C (2014) Persistent data-only malware: function hooks without code. In: NDSS
63. Wampler DR (2007) Methods for detecting Kernel Rootkits. PhD thesis, Louisville, KY, USA. AAI3293571
64. Willems C, Holz T, Freiling F (2007) Toward automated dynamic malware analysis using CWSandbox. IEEE Secur Priv 5(2):32–39
65. Wißfeld M, von Styp-Rekowsky P, Backes M Callee-side method hook injection on the new Android runtime ART
66. Xu K, Li Y, Deng RH (2016) Iccdetector: Icc-based malware detection on android. IEEE Trans Inf Forensics Secur 11(6):1252–1264
67. Xu R, Saïdi H, Anderson R (2012) Aurasium: practical policy enforcement for android applications. In: Proceedings of the 21st USENIX conference on security symposium, Security'12. USENIX Association, Berkeley, pp 27–27
68. Ye Y, Wang D, Li T, Dongyi Y (2007) IMDS intelligent malware detection system. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '07. ACM, New York, pp 1043–1047
69. Yin H, Liang Z, Song D (2008) HookFinder: identifying and understanding malware hooking behaviors. In: Proceedings of the 15th annual network and distributed system security symposium (NDSS'08)
70. Yucheng G, Peng W, Juwei L, Qingping G (2011) A way to detect computer trojan based on DLL preemptive injection. In: 2011 tenth international symposium on distributed computing and applications to business, engineering and science (DCABES), pp 255–258
71. Zdziarski J (2012) Hacking and securing iOS applications: stealing data, hijacking software, and how to prevent it. O'Reilly Media, Inc., Sebastopol