LoGos: Internet-Explorer-Based Malicious Webpage Detection

Sungjin Kim, Sungkyu Kim, and Dohoon Kim

Malware propagated via the World Wide Web is one of the most dangerous tools in the realm of cyber-attacks. Its methodologies are effective, relatively easy to use, and are developing constantly in an unexpected manner. As a result, rapidly detecting malware propagation websites from a myriad of webpages is a difficult task. In this paper, we present LoGos, an automated high-interaction dynamic analyzer optimized for a browser-based Windows virtual machine environment. LoGos utilizes Internet Explorer injection and API hooks, and scrutinizes malicious behaviors such as new network connections, unused open ports, registry modifications, and file creation. Based on the obtained results, LoGos can determine the maliciousness level. This model forms a very lightweight system. Thus, it is approximately 10 to 18 times faster than systems proposed in previous work. In addition, it provides high detection rates that are equal to those of state-of-the-art tools. LoGos is a closed tool that can detect an extensive array of malicious webpages. We prove the efficiency and effectiveness of the tool by analyzing almost 0.36 M domains and 3.2 M webpages on a daily basis.

Keywords: API hook, Dynamic analysis, DLL injection.

Manuscript received Nov. 14, 2016; revised Apr. 1, 2017; accepted Apr. 9, 2017.

Sungjin Kim (r3dzon3@kaist.ac.kr) is with the Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Rep. of Korea.

Sungkyu Kim (ormuzd98@ncubelab.com) is with Information Sharing and Analysis Center, NcubeLab, Seoul, Rep. of Korea.

Dohoon Kim (corresponding author, karmy01@add.re.kr) is with the ITMD, Agency for Defense Development, Daejeon, Rep. of Korea.

I. Introduction

Over the years, malware has become increasingly complex. It infects systems by creating malicious processes or system files, or by altering code to impede the kernel, firmware, or hypervisor through "drive-by downloads." These trends characterize most current attack techniques.

In particular, these attacks have enabled attackers to launch advanced and persistent threat attacks while hiding their identities and infiltrating systems using IP addresses or URLs in webpages. These attacks are propagated through the web and spread toward the target location. These attacks, which target systems of all types, are becoming increasingly common.

For personal users, anti-virus software provides safety to some extent. However, current anti-virus software installed on tens of millions of user PCs requires many other resources such as signatures to detect downloaded malware and protect against malicious web access. Under these circumstances, these widely used anti-virus tools cannot prevent current malware attacks when users surf the Internet or access email because adversaries propagate malware after verifying whether antivirus tools can evade malware [1], [2]. These circumstances allow attackers to bypass a user's virus detection tool. Moreover, there exist vast numbers of webpages. Manually inspecting these webpages is impossible. Thus, we need an accurate, speedy, and automated malicious webpage detection system for supporting rapid pattern (or signature) updates.

Despite prior trials on remediating this issue, there is still a need for a model that effectively addresses both high-speed scanning and high detection rates. Systems proposed in previous work [3]–[5] exhibit some limitations in terms of performance and detection rate.

In an effort to overcome these limitations, our proposed system utilizes methods similar to those found in other virtual machine (VM)-based systems. However, compared with previous models, our model is significantly faster. It is also proactive and does not wait for a malicious attack on the system. It utilizes dynamic-link libray (DLL) injection-based API hooking. Although the application programming interface (API) hooking method is not a new technique, we have improved the technique to allow it to meaningfully detect malicious webpages with increased performance and detection rates.

The main contributions of this study are as follows:

1. We introduce a malicious webpage detection system that is 10 to 18 times faster than those in previous studies.
2. We demonstrate that our efficient approach is applicable to actual production environments.

The remainder of this paper is organized as follows. We describe related research in Section II. We provide an overview of the proposed model in Section III. In Section IV, the technical details of our implementation are explained. Section V describes the experimental setup, and reports on the experimental results. We discuss limitations of our framework in Section VI. Our conclusions are outlined in Section VII.

II. Related Work

In this research, we analyzed two types of malicious webpage detection techniques.

In high-interaction dynamic analysis, Capture-HPC [6] and Anubis [3] enable systematic state-based detection of a client attack by monitoring file changes and network state transitions. They are very effective in detecting unknown malware attacks on clients. Similarly, CWSandbox [7] and Norman Sandbox [8] provide analysis of malicious behaviors. In particular, in API hooking with inline code overwriting, CWSandbox overwrites "six-byte code" for surveillance, including files, registry entries, events, and handles. These dynamic analysis systems focus on malware itself, whereas we are concerned with harmful websites perpetrated through a browser. These models provide high detection rates; however, the time trade-off is very high. That is, these dynamic methods involve intensive methodologies because they are installed, loaded, and monitored in an unpatched VM-based system. They achieve high detection rates in environments that run portable executable files; however, they have considerable limitations owing to performance issues.

In this regard, new alternatives are needed to overcome current performance issues. Thus, we adopted an API hooking technique, but we hooked Internet Explorer (IE) browsers running in multiple VMs to detect original malicious websites producing malware.

We monitor the process ID (PID), file creation, and network connections because malicious websites can spawn new processes or connect to a remote server. Under this circumstance, the detection process should be procedural. Individual processes are not often considered malicious. LoGos's heuristic approach is sequentially coded according to loading processes. To manage these processes, we monitor PIDs that are created, and determine the upper browser handler related to those PIDs. To simultaneously achieve this objective, we maintain a history of file creations and events and store them in each VM. System call sequences are also compared against predefined malicious profiles (details in Section IV).

Wepawet [9] performs low-interaction dynamic analysis to classify malicious code. Thug [5] provides a YARA rule-based emulator. Wepawet shows difficulties in detecting various web exploit toolkits. Thug is unstable in landing *ActiveXObjects*. It occasionally stops and delays its analysis (we prove this trend in Section V.) In terms of detection rate and performance, we hooked IE safely. This hooking did not incur any system crashes or critical system errors during the experiments, in which myriads of websites were inspected. Our model hooks millions of loaded IE browsers, but it is safe. This system is also secure because we run webpages on a secure container, and we revert to a new VM image when malicious webpages are detected. Our detection mode is designed to be deployed in a structure that can tolerate a performance overhead with multiple IE browsers and multiple VMs, permitting important insights into the characteristics of malicious code.

CAMP [10] is built into Chrome and relies on the reputation data of Google's Safe Browsing API. ZOZZLE [11] statistically analyzes unobfuscated codes generated by manipulating the function in the *Jscript.dll* library of IE. This approach provides quick detection and high accuracy, but also provides a low detection rate for the shell code detection in Flash ActionScripts and Java applets. In another approach, Blade [12] redirects browser downloads into a secure zone, but only in the case of ".exe" file types. It does not support other file types (for example, scripts). Such studies use the infrastructure of specific browsers and only identify some malicious characteristics; however, our approach can be deployed in various browsers and it can validate a variety of attack types.

To do this, we adopted DLL injection and an API hooking method. DLL injection is a technique for forcibly inserting and loading a DLL file within the address space of IE to manipulate the DLL according to our purposes. This can replace IE with other types of browsers such as Firefox or Chrome. As in previous studies, it does not require the modification of a browser or Windows DLL files. We only need our DLL file.

This method is especially helpful when scanning massive domains and checking validations. It can indicate whether a webpage contains malicious code, even when link tags are obfuscated. For JavaScript deobfuscation, previous studies [5], [13] used a JavaScript engine (SpiderMonkey [14], PhantomJS [15], Google V8 [16], PyV8 [17]). Meanwhile, we used IE, which is one of the most-used browsers that are targeted by attackers. Adopting a real browser increases the attack surface for detection.

III. Architecture

LoGos operates in IE-based VMs. We offer brief information and sample output for LoGos at this website: *https://drive.google.com/file/d/0Bwjmbj3-p7V_OEVxc3RBcXR6bXM/view*

1. Design Overview

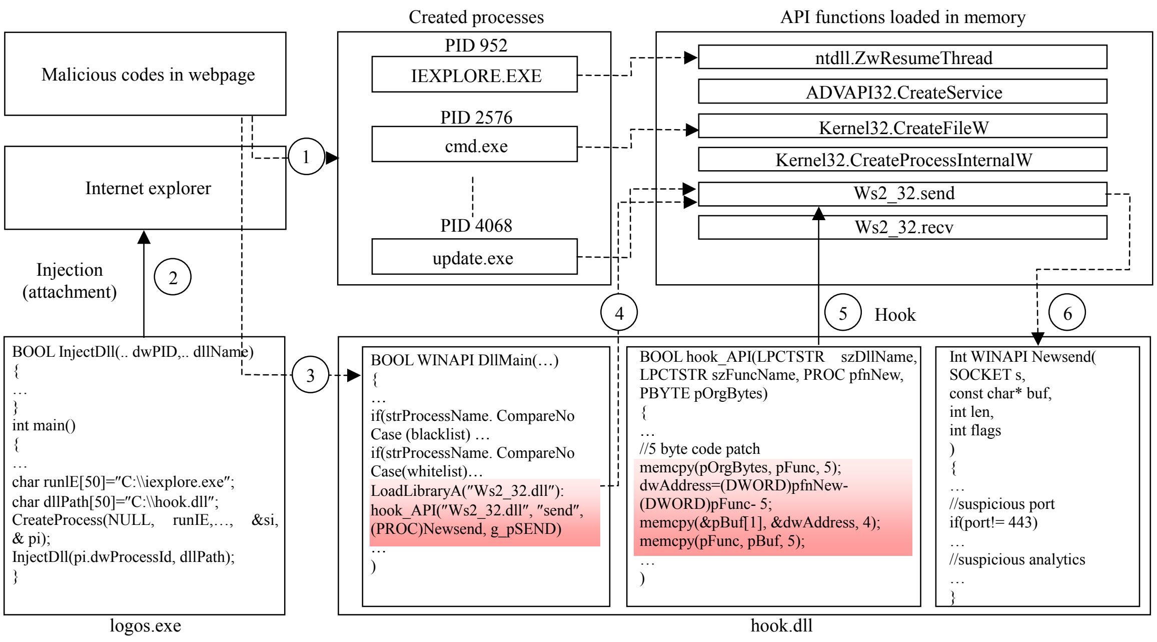
In this section, we explain the detailed architecture of LoGos. Adversaries follow two steps to initiate an attack.

First, they insert redirection links into landing pages that reach malware distribution webpages, using exploit kits such as Neutrino, Redkit, or Rig. They then activate the webpages to exploit the user's system via Java Applets, Adobe Flash Player, ActiveX controls, XML, IE, and other Web plugins with vulnerabilities. Attack attempts to disrupt web users are based on the exploit kits in webpages [18]–[20]. These kits enable users to easily allocate heap memory space. In web attack cases, this heap spray [21] is still widely used (bypassing ASLR and DEP [22], [23]). These types of attacks are commonly implemented in VBScript, ActionScript, and HTML5 [24]–[27]. In this regard, a detection system should respond to various exploit kits that are widely used and are newly emerging. To detect these toolkits, LoGos injects a DLL file when IE is loaded. Subsequently, the DLL captures artifacts that characterize malicious activities that occur when an IE is loaded. For instance, the DLL extracts processes that have suspicious filenames. We describe the analysis modules in a DLL file. To use the modules, we adopt the API hooking method.

LoGos's hooking is unlike a typical one, which manipulates the memory address of related functions for intercepting events or messages passed between software components [28]. Meanwhile, we first attach a DLL file and manipulate the API functions' addresses in the DLL.

Our hooking follows these steps: i) injection, ii) API function calls, iii) API hooking, iv) a 5-byte code patch, and v) attack detection. As shown in Fig. 1, ② LoGos attaches our DLL file (*hook.dll*) to each IE (*iexplore.exe*), and then ③ calls API functions we want to observe. ① This hooking changes the workflow from ① to ③. In general, malicious webpages directly accesses system DLL files via IE; however, when hooking is applied, the workflow passes through the *hook.dll*. These API function accesses via IE are deemed malicious because general webpages do not call these predefined API functions via a browser. Under these circumstances, ③ the injected DLL file is loaded prior to other DLL files when IE is launched. In the DLL file, we define ④ "a white/blacklist detection module," "API function calls," ⑤ "an API hooking function with 5-byte code patch," ⑥ "API function analysis modules for attack detection," and an "unhooking module." In this hooking, a "precedence procedure" to attach "*hook.dll*" to IE is the most important step in the hooking process. From this injection, *hook.dll* is considered as one of the system libraries used by IE browsers.

Injection and hooking are achieved using two files, namely, *logos.exe* and *hook.dll*, respectively. When *logos.exe* is executed,



```
BOOL InjectDll(.. dwPID,.. dllName)
{
…
}
int main()
{
…
char runlE[50]="C:\\iexplore.exe";
char dllPath[50]="C:\\hook.dll";
CreateProcess(NULL, runIE,…, &si,
& pi);
InjectDll(pi.dwProcessId, dllPath);
}
```

```
BOOL WINAPI DllMain(…)
{
…
if(strProcessName. CompareNo
Case (blacklist) …
if(strProcessName. CompareNo
Case(whitelist)…
LoadLibraryA("Ws2_32.dll"):
hook_API("Ws2_32.dll", "send",
(PROC)Newsend, g_pSEND)
…
)
```

```
BOOL hook_API(LPCTSTR szDllName,
LPCTSTR szFuncName, PROC pfnNew,
PBYTE pOrgBytes)
{
…
//5 byte code patch
memcpy(pOrgBytes, pFunc, 5);
dwAddress=(DWORD)pfnNew-
(DWORD)pFunc- 5;
memcpy(&pBuf[1], &dwAddress, 4);
memcpy(pFunc, pBuf, 5);
…
)
```

```
Int WINAPI Newsend(
SOCKET s,
const char* buf,
int len,
int flags
)
{
…
//suspicious port
if(port!= 443)
…
//suspicious analytics
…
}
```

Fig. 1. LoGos's hooking process.

it opens a *hook.dll* with an “rb” option. IE is activated using CreateProcess(). Then, InjectDll() function attaches *hook.dll* with PID of IE to be loaded into IE’s memory address space.

If injection is completed, DllMain() in *hook.dll* begins. In DllMain(), we use a black/whitelist detection module and hook_API() calls. In the DLL, we also define hook_API and unhook_API functions, and API analysis modules for attack detection.

There are two detection approaches in terms of timing: before and after drive-by downloads. That is, antivirus software is an example that detects malware evidence created after a drive-by download. By contrast, Thug searches for malicious footprints prior to drive-by download attacks. This system mainly tracks the existence of malicious code called “exploit kits.” Our model focuses on a study that classifies all malicious symptoms that occur before and after drive-by downloads. To detect them, LoGos accesses candidate domains, renders them, and inspects all life cycles during contamination.

Algorithm 1. LoGos Workflow

```
procedure BrowserControl(d, h, r)
    m_bContinue : bool = true
    OnRun()
    GetUrl(out strUrl:string) ← QueueServer
    DeleteInternetFolder()     // Delete cookies
    BrowserExecute()           // Hooked
        _EnableNTPrivilege(inszPrivilege:unsigned ...)
        IsWow64(in dwPID : int)
        InjectDll(in dwPID : int, inszDLLName64 : ...)
        CreateDirectory(strTemp, NULL)
        T = WaitForSingleObject(pi.hProcess, 15 × 1,000)
        while do(T == TRUE || BHO == FALSE)
            store log
            GetWinHandle(in pid : unsigned)
            ProcIDFromWnd(in hwnd : HWND)
            PostMessage(hwnd, WM_CLOSE, 0, 0)
            ProcessKill(in dwPid : int)
            ProcessKill(in strProcess : string)
        end while
OnStop()
end procedure
```

Algorithm 1 shows the entire LoGos workflow. In detail, LoGos obtains a candidate domain (or a URL) from a queue server. It then deletes cookies and history files in the temporary folder because adversaries check items in a user’s web browsing history, such as cookies or IP addresses. Users are protected against revisits and reinstalls. If these exist, the malicious webpage does not respond. Thus, we delete cookie and history information before accessing candidate domains.

After that, multiple IE browsers are executed with each given domain. When the IE browsers are loaded, our DLL file is injected into each IE. LoGos then creates a directory folder and stores all related logs and source code there if attacks are identified for a 15-s waiting time (we can manipulate the timeout according to the network environment).

Adversaries often inject obfuscated attack code in webpages in order to mask its presence. LoGos deobfuscates the contents via an IE *JScript*, and the web content is converted to CSS/DOM format through the parser of a browser engine. Then, malicious webpages open redirection links, load vulnerable Java/Flash, create a new process or files, modify the registry, and attempt to open a new network connection. All of the opened links (or created processes) are analyzed in terms of their dynamic behavior. IE plays a critical role in locating malicious links, whereby API function calls are activated and LoGos identifies malicious traits within webpages.

A hooked IE renders webpages of accessed domains and carries out surveillance of all symptoms obtained from the hidden malicious links. Thus, IE and API hooking help in distinguishing malicious URLs from benign ones.

LoGos is composed of several parts, including the main code section for IE landing and browser injection, a DLL file for global API function hooking and attack surveillance, browser helper object (BHO), load balancing, and log file creation/file management. It contains 6,927 lines of code in total. Our proposed system was developed using C/C++, and includes some shell script codes for reverting a VM image. It is deployed on multiple VMs to support in-depth behavioral monitoring environments. IE affects the detection rate because it provides a real browser environment.

2. System Platform

Our system was designed on a machine with an Intel® Xeon® E5-2620 v2 six-core 2.1-GHz Processor, 32-GB memory, and a 551.5-GB HDD. Our platform allocates 58.57 GB per VM and utilizes almost 26.62 GB per VM. It has 1-Gbps NICs equipped with the 82599 chipset, but uses 100-Mbps connections.

This system is implemented using VMware ESXi 6.0.0 [29]; each VM runs on Windows XP, 7, 8, 10 and other Windows Platforms. Four VM images are used, and each VM loads 15 IE browsers at a time; 4 GB of RAM is allocated to each image. Plugins installed on each VM are unpatched versions such as Java 7.0.100, MS IE 8.0, Adobe Flash Player 15.0.0.167, and Silverlight 4. One of the properties of our platform is that various plug-ins can be installed without limitations. The system uses a RabbitMQ version3.3.5 queue server [30].

To ensure efficient performance, the LoGos system distributes its workload after multi-instance creation (for example, 15 browsers per VM). Each of the four VMs sequentially receives candidate domains from the RabbitMQ queue, which distributes a fixed subset of candidate domains;

each VM stores them in a local directory. Each VM executes this task repeatedly until all domains are read by the multiple browsers. This platform automatically reboots when the VM is powered off; that is, it reverts to a specific snapshot if the Windows OS is compromised while analyzing malicious webpages.

IV. Implementation

LoGos monitors the following suspicious traces: file creation and modification, command line execution, registry changes, and new network connection trials. In this section, we discuss the implementation of this approach.

1. Initialization

A. Set Configuration

LoGos reads configuration values from each VM; these include a queue server IP, port, user name, and queue name obtained from a local config.ini file. It connects to a queue server and receives a set of domain lists, and stores them in a local directory. If the received domain lists are used, LoGos reconnects to a queue server to obtain additional domain lists. This process is constantly repeated. All VMs look for the same queue server; however, they use different domain lists based on different queue names. To manage domains that are suspended because of the uncertain conditions of browsers on VMs, a queue server is associated with each VM; this server manages a history of used domains and sends the remaining domains to each VM.

B. Performance Management

To perform load balancing, LoGos receives the counter values of the system's CPU and memory from the CPerfMon class. The MAX_CPU_USE of LoGos allocates 60 values by default. If CPU usage exceeds the default value, the system sleeps for a random number of seconds. In general, this CPU management affects the performance of the entire system.

C. Browser Loading

LoGos emulates user Internet access. Unlike prior studies [5], [9], our approach closely simulates a typical user's Internet access. Instead of a user entering a URL in IE, LoGos automatically loads IE browsers as commanding domains such as "*iexplore.exe* http://www.domain.com." In fact, by calling CreateProcess(), LoGos runs IE (*iexplore.exe*) using domain lists stored in a local directory.

We do not face DOM parsing difficulties caused by unsafe browser engine customization, or problems related to JavaScript parsing, add-on plugin compiling, and insufficient functions. This provides optimized safety. Furthermore, our hooking method provides a lightweight load to IE such that it renders a total of 60 IE instances on four Windows 7 VM images simultaneously. This simultaneous rendering enhances the overall performance.

In the system design for browser loading, we need to delete CSIDL_COOKIES, CSIDL_INTERNET_CACHE, and CSIDL_HISTORY. By periodically visiting the same malicious webpages, adversaries block access to them. They check the browsing history to deny continuous access. Hence, we remove this history using the SHGetFolderPath and FolderDelete functions.

2. Browser Helper Object

To rapidly distinguish malicious domains from the numerous available domains, an increase in the crawling performance is required. In this regard, we applied BHO to quickly close the browsers after the completion of browsing tasks.

Normally, adversaries attack users with tiny files in a short time because users disconnect their browsers when accessed websites take a long time to load. In particular, attackers tend to operate malicious websites for only a short time, and we must respond to as many websites as possible. In this circumstance, BHO closes browsers when their loading is finished or when an HTTP status code 404 is returned. IE waits 15 s to reflect the maximum loading time. In general, browser loading time is less than 10 s in benign domain access. However, many malicious websites execute various covert actions during loading time. This results in a slight delay compared to benign webpage access. Thus, to improve the system performance, if

```
#define INET_E_RESOURCE_NOT_FOUND INET_NOT
#define HTTP_STATUS_NOT_FOUND HTTP_NOT
#define HTTP_STATUS_FORBIDDEN FORBIDDEN

STDMETHODIMP BHO::Invoke(DISPID dispidMember, ...)
{
    switch(dispidMember)
    {
      case DISPID_DOCUMENTCOMPLETE:
              TerminateProcess(GetCurrentProcess(), 0);
      break;

      case DISPID_NAVIGATEERROR:
         CString strTemp;
         VARIANT*vt_statuscode=pDispParams->rgvarg[1]. pvarVal;
         DWORD dwStatusCode = vt_statuscode->lVal;

         if(m_bIsError && (dwStatusCode == INET_NOT || dwStatusCode ==
HTTP_NOT || dwStatusCode == FORBIDDEN))
         {
            HWND hWindowHwnd = GetWinHandle (GetCurrentProcessId());
               if(hWindowHwnd != NULL)
                    ::PostMessage(hWindowHwnd,WM_CLOSE,0,0);
         }
      break;
    }
}
```

Fig. 2. BHO code example.

an IE browser completes all jobs in less than 15 s, the BHO immediately closes the browser. LoGos detects a malicious website within 0.01 s to 15 s in general cases.

As shown in Fig. 2, we close the window of the related IE handle after calling GetWinHandle() with the current PIDs (for example, ProcIDFromWnd()). The IE window finishes with PostMessage(). Otherwise, the BHO can be replaced with OnDocumentComplete(). All processes are terminated within 1 min after new process creation.

3. API Function Hooking

Our hooking procedure is largely composed of DLL injection and a 5-byte code patch.

A. DLL Injection

LoGos utilizes DLL injection, which is a technique to insert custom DLL code (for example, *hook.dll*) into the address space of a running process (for example, *iexplore.exe*).

This DLL injection offers Windows API manipulation, which allows us to debug them for our purposes. Injection uses the "*hook.dll*" pathname and IE PID as parameters. To carry out this injection process, we follow five steps.

First, we offer certain access rights to perform our tasks. We adjust privileges with an _EnablePrivilege(SE_DEBUG_NAME, SE_PRIVILEGE_ENABLED) call including OpenProcessToken, LookupPrivilegeValue, and AdjustTokenPrivileges functions.

Second, we call the InjectDll(pi.dwProcessId, dllPath) function. pi.dwProcessId is the IE PID, and dllPath is the *hook.dll* directory pathname.

Third, in the InjectDll function, we open the running process (*iexplor.exe*) for injection using the PROCESS_ALL_ACCESS parameter of OpenProcess(). This function obtains a handle for the process.

Next, we allocate memory by calling VirtualAllockEx(), which allocates enough memory to locate a *hook.dll* path string. The DLL becomes a virtual protect with the PAGE_READWRITE parameter. Then, LoGos copies the DLL path into the allocated memory space of the IE process using WriteProcessMemory().

Lastly, when LoadLibraryA() is called, it jumps to the DllMain() of the *hook.dll*. This is achieved by GetProcAddress(GetModuleHandle ("kernel32.dll"), "LoadLibraryA") and CreateRemoteThread() for running *hook.dll* attached to the *iexplor.exe*.

We also call WaitForSingleObject(hThread, INFINITE) to wait for a malicious thread. Consequently, injection is a procedure for calling DllMain() and preparing API hooking within *hook.dll*.

B. Code Patch

A code Patch is applied in all threads and API functions that we observe.

First, to surveil the behaviors of all created threads, we hook ZwResumeThread of "*ntdll.dll*." LoGos obtains the handler of "*ntdll.dll*" by calling GetModuleHandle() with the "*ntdll.dll*" name. Next, our system obtains the address of the ZwResumeThread function by calling GetProcAddress(). Then, we execute a 5-byte code patch (JMP + address) using memcpy(), which starts at 0xE9 (JMP opcode) with the memory address of the *thread* detection module as its parameter. This "code patch" helps alter an address location to our surveillance handler (for example, NewZwResumeThread). This enables us to monitor all threads. We obtain the code patch address from this formula: *Address = address of NewZwResumeThread function - address of ZwResumeThread function - 5.*

LoGos also hooks other API function addresses of Windows system DLL files that are mapped with IE (it does not hook API address in the Import Address Table or Export Address Table). That is, it hooks the API function address, which is related to the network, registry, or process, for attack surveillance. Thus, this model can intercept all malicious information via *hook.dll*. For example, it searches the "*Ws2_32.send*" API function address, and modifies the 5-byte code of the API function (for example, MOV EAX, OAD to JMP 10011101). The JMP address indicates a detection module [for example, Newsend()] for HTTP GET/POST traffic collection.

Last, LoGos supports the unhook process to revert to the original address bytes using memcpy(OriginDllAddress, OrginalByte, 5) when IE terminates tasks.

4. Detection Method

LoGos has three detection approaches: process names-based blacklist/whitelist, suspicious API function calls, and analytics. Each has detection modules. They are defined in BOOL WINAPI DllMain() and user-defined functions (for example, int WINAPI Newsend()) in *hook.dll*. The entire flow of the detection method is described in Fig. 3.

A. Blacklist and Whitelist Detection

The malicious webpages manipulate processes and read, write, and modify files through them. They download files and command the code snippets. To address these malicious behaviors, we defined the detection modules in DllMain().

DllMain() has two cases: DLL_PROCESS_ATTACH and DLL_PROCESS_DETACH. In the first case, we describe blacklist/whitelist *process* detection modules. In whitelist detection, LoGos regards these processes such as *iedw.exe*,

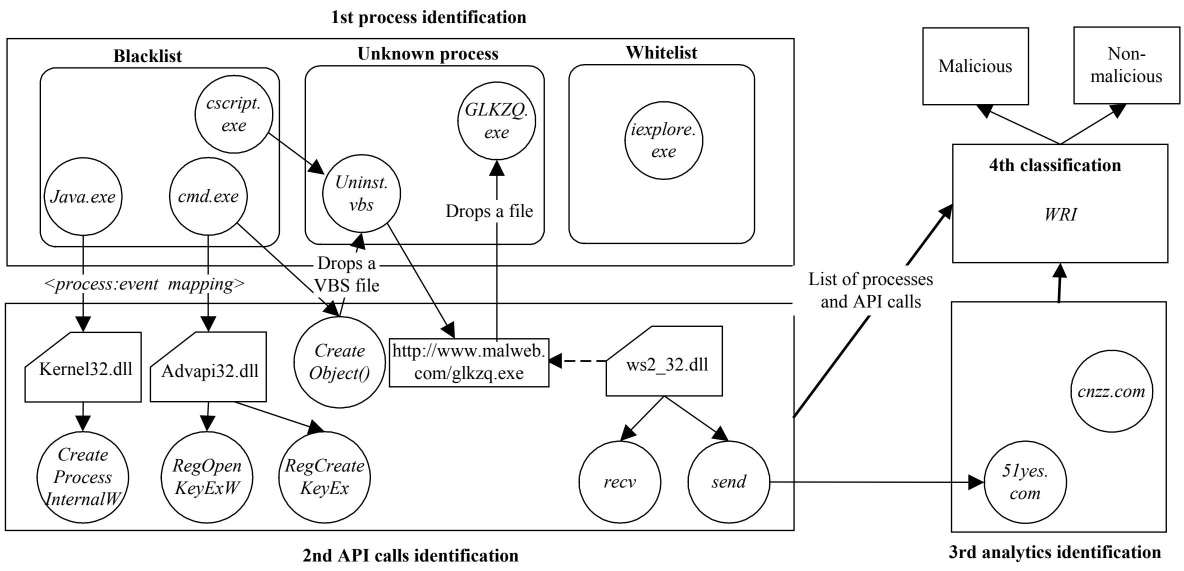


Fig. 3. Block diagram of the operation of LoGos detection engine.

werfault.exe, *dwwin.exe*, *drwtsn32.exe*, *wuapp.exe*, *and vsjitdebugger.exe as* benign. LoGos bypasses processes in the whitelist because these processes are used as Windows core system files or as debuggers. Of course, there exist exceptional cases such as "*Windows Updates*" and "*harmless plugins* (for example, for banking)." In these cases, we bypass them with the whitelist.

Next, LoGos checks the blacklists. If a created process is not "*iexplore.exe*" or a whitelist, LoGos identifies whether the process name exists on the blacklist. Before drive-by downloads, malicious websites force users to load plugins such as Java or Flash, and then download VB scripts, open command prompts, execute command lines, and create new files. These process-related activities occur in rapid succession. Hence, LoGos intensively observes opened processes such as the cmd shell (known as "*cmd.exe*") and other executable programs (for example, *java.exe*, *jp2launcher.exe*, *javaw.exe*, *wscript.exe*, *cscript.exe*, *regsvr32.exe*, and *powershell.exe*). These processes are closely related to malicious attacks. Given this backdrop, this single process is considered to be malicious, and a high weight is allocated. Of course, other processes except the blacklist/whitelist are considered to be suspicious. We assign a low weight to each process to estimate riskiness.

LoGos determines severity based on pairs of processes and events. For example, there are attack process trees such as *iexplore.exe*/HttpOpenRequestA, *iexplore.exe*/URLDownloadToFile, and *malware.exe*/CreateFileA. If *regsvr32.exe* emerges in a serial process tree and calls RegOpenKeyExA(), our model generates an alert about this process. Further, if *cmd.exe* calls the CreateObject() function, this behavior is considered as malicious. We inspect command logs by GetCommandLine(). Likewise, LoGos inspects the *process sequence* and *API function calls* of the opened, loaded, and created processes. We maintain a pool of running processes and suspicious API functions defined as malicious.

B. Suspicious API Function Call Detection

LoGos surveils API functions used by malicious websites. For example, "RegOpenKeyExW" denotes "signatures" of malicious behaviors because general webpages do not open the registry key.

To monitor these suspicious API functions, we defined the API function calls in dllMain(). For instance, *hook_API*("*Ws2_32.dll*," "*send*," (*PROC*)*Newsend*, *g_pSEND*); In a *hook_API* function, we state "5-byte code patch" to point out the initial address of the *Newsend*() function. Therefore, from this indication, LoGos can intercept "attack behaviors" of all "*Ws2_32.dll.send*" function calls exported via IE browsers or created processes.

In addition, we declared other API functions such as thread (*ntdll.ZwResumeThread*), mouse event (*user32.CallNextHookEx*), network (*Ws2_32.recv*), crypt (*ADVAPI32.CryptEncrypt* and *wininet.dll*), and registry (*ADVAPI32.Reg CreateKeyExA*). We also identified unauthorized processes created through "*kernel32.Create ProcessInternalW*."

In particular, LoGos monitors changes in registry locations by hooking *ADVAPI32.dll*. In this case, our model surveils RegOpenKeyExA, or RegOpenKeyExW calls, which are key values for Run, Associations, and Start Page.

To read the network traffic, we hook "*send*" and "*WSASend*" of *Ws2_32.dll*. By hooking winsock's *send()* and *recv()*, we can monitor port 443 and unfamiliar ports for unknown request methods (except HTTP methods such as POST, HEAD, PUT, DELETE, TRACE, and JQSM) and unknown HTTPS protocols (excluding 0x16, 0x14, 0x01 in (BYTE*)buf, or encryptedString). From this traffic information, LoGos also detects suspicious *analytics* such as *nd.qq.com*, *XXyes.com*, and *cnzz.com*. Attackers often insert *analytics* in webpages to gather the victim's access information. This is commonly accompanied by malicious URLs. Further, we can extract malicious links that are concealed in the referrer and URI of the HTTP header.

Similarly, we define a NewCallNextHookEx() function that is used to manipulate a left mouse event. Thus, the Windows popup messages for "Install," "Save," and "Run" are controlled.

LoGos calculates the sum of the product of each opened process and its API function calls for estimating the riskiness of accessed domains. For the risk assessment, we follow these steps: The eigenvector $\boldsymbol{w}$ matching the maximum eigenvalue $\lambda_{\max}$ of the pairwise comparison matrix $\boldsymbol{A}$ is the final expression of the *preferences* among the processes (related to the blacklist and non-blacklist in our method). To determine the eigenvector, we need to find the solution of the characteristic equation of matrix $\boldsymbol{A}$. The characteristic function is as follows:

$$f(\lambda)=|A-\lambda I|=\begin{vmatrix} a_{11}-\lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22}-\lambda & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n1} & \cdots & a_{nn}-\lambda \end{vmatrix}. \quad (1)$$

Its respective characteristic equation $f(\lambda) = |\boldsymbol{A} - \lambda I| = 0$ is presented in the form of the polynomial $c_0\lambda^n + c_1\lambda^{n-1} + \cdots + c_{n-1}\lambda + c_n = 0$. The eigenvectors of matrix $\boldsymbol{A}$ are the columns and nonzero vector $\boldsymbol{X}_i$, for which the following equality holds: $(\boldsymbol{A} - \lambda_i)\boldsymbol{X}_i = 0$. If we assume that $\boldsymbol{X}_i = \boldsymbol{w}$ for eigenvector $\lambda_{\max}$, we need to find the solution to the equation $\boldsymbol{Aw} = \lambda_{\max}\boldsymbol{w}$. Below are three methods for finding the eigenvector corresponding to $\lambda_{\max}$ in our risk estimation. Our proposed model is defined as a method of normalized arithmetic averages. The prepared pairwise comparison matrix is normalized. As a result of the normalization, matrix $\boldsymbol{A}$ is transformed into matrix $\boldsymbol{B} = [b_{ij}]$. The malware of matrix $\boldsymbol{B}$ are calculated according to the following formula:

$$b_{ij} = \frac{a_{ij}}{\sum_{i=1}^{n} a_{ij}}. \quad (2)$$

Calculation of the preference among the malware under investigation (eigenvector $w = [\mathbf{w}_i]$) is performed by calculating the arithmetic averages of the rows of the normalized comparison matrix. The components of this vector are calculated according to the following formula:

$$w_j = \frac{\sum_{j=1}^{n} b_{ij}}{n}. \quad (3)$$

The maximum eigenvector is calculated according to the following equation:

$$\lambda_{\max} = \frac{1}{n}\sum_{i=1}^{n}\frac{(\boldsymbol{Aw})_i}{w_i}. \quad (4)$$

The above method for calculating the eigenvector and eigenvalue gives good results when there is high consistency in the pairwise comparisons. This $\lambda_{\max}$ is a preference list for opened processes n. This output becomes the input of p_i. We multiply each process p_i with all a_j (called API functions). Thus, the website risk index (WRI) can be normalized as follows:

$$WRI = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} p_i a_j}{n}. \quad (5)$$

A risk index above the threshold (*WRI* ≥ *threshold*) is a critical factor in identifying maliciousness.

V. Evaluation

In this section, we evaluate the detection effectiveness and performance of LoGos. First, we compare its detection rate with those of the systems proposed in previous studies. Second, we evaluate the overall performance of LoGos using a large-scale dataset.

1. Comparison with Previous Work

To compare our system with those from prior studies, we selected Thug 0.7.2 and Cuckoo Sandbox 2.0. Both are state-of-the-art tools for detecting malicious activities. In particular, Thug is a low-interaction, Python-based honeyclient that emulates a web browser environment for analyzing malicious web content. Cuckoo is a sandbox tool that mimics virtual system environments to facilitate the in-depth analysis needed to detect reconnaissance and drive-by download symptoms caused by malicious URLs and files. LoGos has adapted the properties of both systems; namely, malicious webpage detection based on some profiled patterns and dynamic analysis via the Windows API hook method. We compared the two systems with our proposed model. Both tools were updated to contain their most recent functionalities.

We first compared detection rates with these tools. Before testing, we collected a labeled dataset composed of both malicious and benign pages. The dataset comprises 57 domains that are known to trigger drive-by download attacks. These domains were extracted from DNS-BH [31], malware domain list [32], and a third-party vendor; they used exploit kits such as Rig, CK VIP, SweetOrange, Gongdad, and Blackhole. We performed manual verification to confirm that these domains

Table 1. Comparison with previous work.

Work	False negative (%)	False positive (%)
Thug [3]	26.315	39.048
LoGos	1.754	1.905
Cuckoo [2]	1.754	14.286

are indeed working as malicious websites; VirusTotal was used as a reference [33].

One hundred benign domains from the Alexa Top 250 [34] were also included in our sampled domains. In addition, we added five benign domains related to Windows updates and ActiveX-based downloads. Therefore, 105 samples were benign domains. We know that these Alexa Top domains are often used for benign website tests because these sites are well managed to prevent being compromised by attacks. Nonetheless, we confirmed the maliciousness of these domains using VirusTotal.

Note that we unchecked options that are offered in Cuckoo for providing rapid test results. We also note that one IE and one VM were used in this test for LoGos. This circumstance allowed us to fairly evaluate the effectiveness of test tools.

The results of the test are presented in Table 1. The detailed results show that LoGos exhibits a high detection rate and a low FN/FP rate. Moreover, our exceptional processing is highlighted even in benign update sites; however, we could still not handle some ActiveXs (we resolved this problem in a subsequent revision, but Cuckoo still has this issue). Both systems effectively detected malicious files created by drive-by downloads.

Thug showed a FN/FP rate of about 26% and 39%, respectively. Its FN seems to have a lack of pattern rules. YARA rules [35] such as g01Pack 2 also produced high FP. When these rules were not applied, the FP rate was reduced to 9.524%. Cuckoo showed a high detection rate in a Windows 7 64-bit VM environment. However, it still returns a high FP rate for benign ActiveX downloads and some downloaded files. A FN of LoGos and Cuckoo caused improper plug-in versions.

LoGos was tested with 4 GB of memory, 1 CPU, and a Windows 7 VM image on a PC with an Intel i7-3610QM CPU. Thug and Cuckoo were tested on a server with Intel(R) Xeon(R) CPU E5620 at 2.40 GHz and with 32 GB memory.

Interestingly, our model yields low FP and FN rates. We substantially reduced the FNs because our model closely emulates a general Internet surfing environment. This means that if vulnerable circumstances are satisfied, our model shows an almost 100% detection rate. This model is not absolutely dependent on pattern rules or new features.

2. Performance Test

We evaluated the performance of each system as shown in

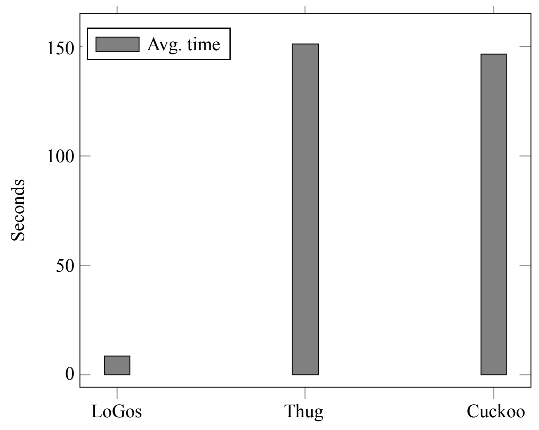


Fig. 4. Average analysis time comparison with previous work per domain.

Fig. 4. LoGos needed an average of 8.234 s to process every domain. Thug and Cuckoo required 151.503 and 146.579 s, respectively.

In this evaluation, Thug showed slight instability when loading ActiveXObjects such as Microsoft.XMLHTTP and wscript.shell, or JavaScript. These events were improperly handled and resulted in excessive time consumption. If it did not require such loading times, Thug could reduce the detection time to within an average of 40 s, but tended to be stuck in some domains when executing consecutive analyses. It needed more than 30 min in some cases, although it provides fast processing in general. Thus, in our test, the performance of Thug was lower than that of high-interaction Cuckoo. Furthermore, in our per-webpage analysis, the average analysis time of LoGos was 0.624 s.

3. Large-Scale Performance Evaluation

This test was conducted over a six-month period by running a large-scale dataset, which comprised the top 1 million sites from Alexa.

In the large-scale performance evaluation, LoGos used a local queue server. LoGos communicated with the queue server for downloading candidate domains, and each VM sequentially received respective domains. LoGos was composed of four VMs and 15 IE 8.0 instances per VM with Windows 7. Thus, 60 IE instances were loaded simultaneously on an ESXi platform. We allocated 4 GB of memory and 60 GB of HDD per VM.

LoGos processed an average of 364,356 domains daily. Its tasks included malicious domain detection, malicious URL detection, and created/modified/accessed file extraction. It was able to evaluate approximately 3,264,629 webpages per day, including the collection of all redirection information. This large-scale evaluation can be compared with Prophiler [36], which evaluated 18,939,908 webpages over 60 days. Our model can complete the job within only six days. On the contrary, Thug was unstable in a large-scale test. It showed performance delays of about 4 h, even in the Alexa Top 200 domains, because it was stuck in some domains. Cuckoo evaluated fewer than 600 domains on a daily basis.

Most VM systems exhibit poor performance owing to the limited number of simultaneous VMs and long loading times. By contrast, LoGos shows very high performance and provides high-level safety and robustness despite the use of VMs. LoGos alleviates problems with FNs and augments detection coverage. The result reflects that LoGos is highly effective in practice. In this test, our model's total CPU usage was between 60% and 65%, and it used between 10,853 MB and 12,675 MB of memory.

VI. Discussion

This study revealed the need for i) supporting a stable in-depth behavior analysis model and ii) high performance. To fulfill these objectives, we employed an IE-based API hooking.

Nonetheless, LoGos can provide uncertain results when exploiting code in webpages meets improper plug-in versions. This problem is common in VM-based analysis systems, even though we utilized optimized application versions to elevate detection rates. In this regard, we can intelligently change VM images to different versions and inspect various malicious website attacks. Specifically, our model can perform four different inspections of the same website using four different image versions. These vulnerable applications can be easily added to each VM image. However, we also realize that various vulnerable versions degrade the detection opportunities of the entire domain.

The crawling time and crawling page depth affect the detection rate and performance. Indeed, a full-depth crawling process has a negative effect on malicious website detection because full-depth link scanning of webpages produces uncontrollably heavy loads as a result. Hence, the system degrades the entire detection opportunities because of time consumed to scan unnecessary webpages. Most malicious websites spread their malware during a relatively short lifetime. Attackers briefly open malicious links, and they disappear a short time later. For these attackers' tactics, full crawling is negative. However, there exists a dilemma because some exploit codes remain at deep page depths. Thus, further studies are needed to solve the relationship between page depth and crawling performance.

We also aim to improve the processing power of our model. Despite our efforts, our system's ability to discover large numbers of malicious webpages is still insufficient. The loading of IE browsers requires significant resources. In the future, a more lightweight browser can be used to increase system performance. Thus, our goal is to further enhance system performance and detection accuracy.

Consequently, by utilizing multi-API hooking and multi-VM and multi-IE configurations, LoGos can detect threats in webpages that contain various types of malware. In addition, this model offers effective results because of its stability and high performance. This is especially useful when scanning massive domains as a practical system.

VII. Conclusions

LoGos detects malicious webpages and binary executables by introducing three key concepts: IE-based tracking, API hooking-based detection, and sequential detection model construction. Its IE-based detection scheme replicates the Internet access of users. This model sequentially accesses a large number of domains and identifies attacks. LoGos detects malicious websites over ten times faster than existing tools. This tool is an efficient and lightweight system, which contrasts with legacy static analysis tools and other dynamic tools that use comparative techniques. Similar to most dynamic analysis systems, it provides a high degree of accuracy and reliability in detecting various malicious types. It has proven to have stable operation, and provides scalability with multi-VM and multi-server systems; it is highly practical. Above all, it shows that high-interaction systems can have high performance.

References

[1] B. Eshete and V.N. Venkatakrishnan, "WebWindow: Leveraging Exploit Kit Workflows to Detect Malicious Urls," *Proc. ACM Conf. Data Applicat. Security Privacy*, San Antonio, TX, USA, Mar. 3–5, 2014, pp. 305–312.

[2] B. Eshete et al., "EKHunter: a Counter-Offensive Toolkit for Exploit Kit Infiltration," *Netw. Distrib. Security Symp.*, San Diego, CA, USA, Feb. 8–11, 2015, pp. 1–15.

[3] Anubis, Accessed Nov. 11, 2016. http://anubis.iseclab.org/

[4] Cuckoo Sandbox, Accessed Nov. 11, 2016. https://cuckoosandbox.org/

[5] Thug, Accessed Nov. 11, 2016. http://buffer.github.io/thug/

[6] Capture-HPC, Accessed Nov. 11, 2016. https://projects.honeynet.org/capture-hpc

[7] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security Privacy*, vol. 5, no. 2, Apr. 2007, pp. 32–39.

[8] Norman SandBox, Accessed Nov. 11, 2016. http://sandbox.norman.no

[9] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code,"

Proc. Int. Conf. World Wide Web, Raleigh, NC, USA, Apr. 26–30, 2010, pp. 281–290.

[10] M.A. Rajab et al., "CAMP: Content-Agnostic Malware Protection," *Netw. Distrib. Security Symp.*, San Diego, CA, USA, Feb. 24–27, 2013, pp. 1–15.

[11] C. Curtsinger et al., "ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection," *Proc. USENIX Conf. Security*, San Francisco, CA, USA, Aug. 8–12, 2011, p. 3.

[12] L. Lu et al., "Blade: an Attack-Agnostic Approach for Preventing Drive-by Malware Infections," *Proc. ACM Conf. Comput. Commun. Security*, Chicago, IL, USA, Oct. 2010, pp. 440–450.

[13] A. Dewald, T. Holz, and F.C. Freiling, "ADSandbox: Sandboxing JavaScript to Fight Malicious Websites," *Proc. ACM Symp. Appl. Comput.*, Sierre, Switzerland, 2010, pp. 1859–1864.

[14] SpiderMonkey, Accessed Jan. 25, 2017. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[15] Phantomjs, Accessed Jan. 25, 2017. http://phantomjs.org/

[16] Chrome V8, Accessed Jan. 25, 2017. https://developers.google.com/v8/

[17] PyV8, Accessed Jan. 25, 2017. https://pypi.python.org/pypi/PyV8

[18] T. Taylor et al., "Detecting Malicious Exploit Kits Using Tree-Based Similarity Searches," *Proc. ACM Conf. Data Applicat. Security Privacy*, New Orleans, LA, USA, 2016, pp. 255–266.

[19] B. Stock, B. Livshits, and B. Zorn, "Kizzle: a Signature Compiler for Detecting Exploit Kits," *Annu., IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Toulouse, France, 2016, pp. 455–466.

[20] A. Nappa, M.Z. Rafique, and J. Caballero, "Driving in the Cloud: An Analysis of Drive-by Download Operations and Abuse Reporting," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Heidelberg, Berlin, Germany: Springer, 2013, pp. 1–20.

[21] Heap Spraying, Accessed Nov. 11, 2016. https://en.wikipedia.org/wiki/Heap_spraying

[22] Address Space Layout Randomization, Accessed Nov. 11, 2016. http://en.wikipedia.org/wiki/Address space layout randomization

[23] Data Execution Prevention, Accessed Nov. 11, 2016. https://en.wikipedia.org/w/index.php?title=Data_Execution_Prevention&redirect=no

[24] N. Jagpal et al., "Trends and Lessons from Three Years Fighting Malicious Extensions," *Proc. USENIX Conf. Security Symp.*, Washington, D.C., USA, Aug. 12–14, 2015, pp. 579–593.

[25] G. Stringhini et al., "Shady Paths: Leveraging Surfing Crowds to Detect Malicious Web Pages," *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, Berlin, Germany, Nov. 4–8, 2013, pp. 133–144.

[26] Z. Li et al., "Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising," *Proc. ACM Conf. Comput. Commun. Security*, Raleigh, NC, USA, Oct. 16–18, 2012, pp. 674–686.

[27] G. Wang et al., "Detecting Malicious Landing Pages in Malware Distribution Networks," *Auun. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Budapest, Hungary, June 24–27, 2013, pp. 1–11.

[28] Hooking, Accessed Jan. 25, 2017. https://en.wikipedia.org/wiki/Hooking

[29] VMware ESXi, Accessed Nov. 11, 2016. https://www.vmware.com/products/esxi-and-esx/overview

[30] RabbitMQ, Accessed Nov. 11, 2016. https://www.rabbitmq.com/

[31] Malware Domain Blocklist, Accessed Nov. 11, 2016. http://www.malwaredomains.com/

[32] Malware Domain List, Accessed Nov. 11, 2016. https://www.malwaredomainlist.com/

[33] VirusTotal, Accessed Nov. 11, 2016. https://www.virustotal.com/

[34] Alexa, Accessed Nov. 11, 2016. http://www.alexa.com/topsites

[35] YARA, Accessed Nov. 11, 2016. http://plusvic.github.io/yara/

[36] D. Canali et al., "Prophiler: a Fast Filter for the Large-Scale Detection of Malicious Web Pages Categories and Subject Descriptors," *Proc. Int. Conf. World Wide Web*, Hyderabad, India, Mar. 28–Apr. 1, 2011, pp. 197–206.

Sungjin Kim is currently a PhD student in the Graduate School of Information Security at the Korea Advanced Institute of Science and Technology, Daejeon, Rep. of Korea. He received his BS and MS degrees in computer science from Ohio State University, Columbus, USA and Sogang University, Seoul, Rep. of Korea, respectively. His current research interests include network security, machine learning, big data analytics, social network analysis, web security, and malware detection and analysis.

Sungkyu Kim received his BS degrees from the College of Engineering at Hoseo University, Seoul, Rep. of Korea in 2004. He is currently working as a software developer at NcubeLab, Seoul, Rep. of Korea. His research interests are focused on malware detection using Windows APIs and kernel security.

Dohoon Kim received his BS degrees in mathematics and in computer science & engineering at Korea University, Seoul, Rep. of Korea, in 2005. He received a PhD degree from the College of Information and Communication at Korea University in 2012. He is an information security reader and senior researcher in the IT Management & Support Office at the Agency for Defense Development, Daejeon, Rep. of Korea. His current research interests are network security, risk management, cognitive radio networks, software engineering, situational awareness, future Internet, and forecast engineering.