Transparent ROP Exploit Mitigation using Indirect Branch Tracing

Vasilis Pappas, Michalis Polychronakis, Angelos D. Keromytis
Columbia University

Abstract

Return-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections. ROP exploits are facilitated mainly by the lack of complete address space randomization coverage or the presence of memory disclosure vulnerabilities, necessitating additional ROP-specific mitigations.

In this paper we present a practical runtime ROP exploit prevention technique for the protection of third-party applications. Our approach is based on the detection of abnormal control transfers that take place during ROP code execution. This is achieved using hardware features of commodity processors, which incur negligible runtime overhead and allow for completely transparent operation without requiring any modifications to the protected applications. Our implementation for Windows 7, named *kBouncer*, can be selectively enabled for installed programs in the same fashion as user-friendly mitigation toolkits like Microsoft's EMET. The results of our evaluation demonstrate that kBouncer has low runtime overhead of up to 4%, when stressed with specially crafted workloads that continuously trigger its core detection component, while it has negligible overhead for actual user applications. In our experiments with in-the-wild ROP exploits, kBouncer successfully protected all tested applications, including Internet Explorer, Adobe Flash Player, and Adobe Reader.

1 Introduction

Despite considerable advances in system protection and exploit mitigation technologies, the exploitation of software vulnerabilities persists as one of the most common methods for system compromise and malware infection. Recent prominent examples include in-the-wild exploits against Internet Explorer [7], Adobe Flash Player [2], and Adobe Reader [19, 1], all capable of successfully bypassing the data execution prevention (DEP) and address space layout randomization (ASLR) protections of Windows [49], even on the most recent and fully updated (at the time of public notice) systems.

Data execution prevention and similar non-executable page protections [55], which prevent the execution of injected binary code (shellcode), can be circumvented by reusing code that already exists in the vulnerable process to achieve the same purpose. Return-oriented programming (ROP) [62], the latest advancement in the more than a decade-long evolution of code reuse attacks [30, 51, 50, 43], has become the primary exploitation technique for achieving arbitrary code execution in the presence of non-executable page protections.

Although DEP is complemented by ASLR, which is meant to prevent code reuse attacks by randomizing the load addresses of executables and DLLs, its deployment is problematic. A few code segments left in static locations can be enough for mounting a robust ROP attack, and unfortunately this is quite often the case [35, 75, 40, 54]. More importantly, even if a process is fully randomized, it might be possible to calculate the base address of a DLL at runtime [19, 61, 44, 69, 37, 66], or infer it in a brute-force way [63].

This situation has prompted active research on additional defenses against return-oriented programming. Recent proposals can be broadly classified in static software hardening and runtime monitoring solutions. Schemes of the former type include compiler extensions for the protection of indirect control transfers [45, 52], which break the chaining of the "gadgets" that comprise a return-oriented program, and code diversification techniques based on static binary rewriting [70, 53], which randomize the locations or the outcome of the available gadgets. The lack of source code for proprietary software hinders the deployment of compiler-based approaches. Depending on the applied code transformations, static binary rewriting approaches may be applied on stripped binaries, but their outcome depends on the accuracy

of code disassembly and control flow graph extraction, while the rewriting phase is time-consuming. Depending on the vulnerable program, fine-grained code randomization may be circumvented by dynamically building the ROP payload at the time of exploitation [66, 16]. Runtime solutions monitor execution at the instruction level to apply various protection approaches, such as performing anomaly detection by checking for an unusually high frequency of `ret` instructions [24, 28], ensuring the integrity of the stack [29], or randomizing the locations of code fragments [36]. The use of dynamic binary instrumentation allows these systems to be transparent to the protected applications, but is also their main drawback, as it incurs a prohibitively high runtime overhead.

Transparency is a key factor for enabling the practical applicability of techniques that aim to protect proprietary software. The absence of any need for modifications to existing binaries ensures an easy deployment process, and can even enable the protection of applications that are already installed on end-user systems [47]. At the same time, to be practical, mitigation techniques should introduce minimal overhead, and should not affect the proper execution of the protected applications due to incompatibility issues or false positives.

Aiming to fulfill the above requirements, in this paper we present a fully transparent runtime ROP exploit mitigation technique for the protection of third-party applications. Our approach is based on monitoring the executed indirect branches at critical points during the lifetime of a process, and identifying abnormal control flow transfers that are inherently exhibited during the execution of ROP code. The technique is built around Last Branch Recording (LBR), a recent feature of Intel processors. Relying mainly on hardware for instruction-level monitoring allows for minimal runtime overhead and completely transparent operation, without requiring any modifications to the protected applications.

Inspired by application hardening toolkits like Microsoft's EMET [47], our prototype implementation for Windows 7, named *kBouncer*, can be selectively enabled for the protection of already installed applications. Besides typical ROP code, kBouncer can also identify the execution of "jump-oriented" code that uses gadgets ending with indirect `jmp` or `call` instructions. To minimize context switching overhead, branch analysis is performed only before critical system operations that could cause any harm. To verify that kBouncer introduces minimal overhead, we stress-tested our implementation with workloads that trigger excessively the protected system functions. In the worst case, the average measured overhead was 1%, and it never exceeded 4%. As the protected operations occur several orders of magnitude less frequently in regular applications, the performance impact of kBouncer in practice is negligible. We evaluated the effectiveness and practical applicability of our technique using publicly available ROP exploits against widely used software, including Internet Explorer, Adobe Flash Player, and Adobe Reader. In all cases, kBouncer blocks the exploit successfully, and notifies the user through a standard error message window.

The main contributions of our work are:

- We present a *practical* and *transparent* ROP exploit mitigation technique based on runtime monitoring of indirect branch instructions using the LBR feature of recent CPUs.
- We have implemented the proposed approach as a self-contained toolkit for Windows 7, and describe in detail its design and implementation.
- We provide a quantitative analysis of the robustness of the proposed ROP code execution prevention technique against potential evasion attempts.
- We have experimentally evaluated the performance and effectiveness of kBouncer, and demonstrate that it can prevent in-the-wild exploits against popular applications with negligible runtime overhead.

2 Practical Indirect Branch Tracing for ROP Prevention

The proposed approach uses runtime process monitoring to block the execution of code that exhibits return-oriented behavior. In contrast to typical program code, the code used in ROP exploits consists of several small instruction sequences, called *gadgets*, scattered through the executable segments of the vulnerable process. Gadgets end with an indirect branch instruction that transfers control to the following gadget according to a sequence of gadget addresses contained in the "payload" that is injected during the attack. As the name of the technique implies, gadgets typically end with a `ret` instruction, although any combination of indirect control transfer instructions can be used [23].

The key observation behind our approach is that the execution behavior of ROP code has some inherent attributes that differentiate it from the execution of legitimate code. By monitoring the execution of a process while focusing on those properties, kBouncer can identify and block a ROP exploit before its code accomplishes any critical operation.

In this section, we discuss in detail how kBouncer leverages the Last Branch Recording feature of recent processors to retrieve the sequence of the most recent indirect branch instructions that took place right before the invocation of a system function. In the following section, we discuss how kBouncer uses this information to identify the execution of ROP code. As the vast majority of in-the-wild ROP exploits target Windows software,

Technique	Overhead	Requir.	Compat.	Deployment
Compiler-level	med	source	some	hard
Binary rewrit.	med	pdb	no	med
Dynamic Instr.	high	-	yes	med
LBR monitoring	low	-	yes	easy

Table 1: Qualitative comparison of alternative techniques for runtime branch monitoring.

our design focuses on achieving transparent operation for existing Windows applications without raising any compatibility issues or false alerts.

2.1 Branch Tracing vs. Other Approaches

Execution monitoring at the instruction level usually comes with an increased runtime overhead. Even when tracking only a particular subset of instructions, e.g., in our case only indirect control transfer instructions, the overhead of interrupting the normal flow of control and updating the necessary accounting information is prohibitive for production systems. There are several different approaches that can be followed for monitoring the execution of indirect branch instructions, each of them having different requirements, performance overhead, transparency level, and deployment effort.

Extending the compiler to generate and embed runtime checks in the executable binary at compile time is one of the simplest techniques [52]. However, the high frequency of control transfer instructions in typical code means that a lot of additional instrumentation code must be added. Also, deployment requires a huge effort as all programs have to be recompiled. Another option is static binary rewriting. Its main advantage over compiler-level techniques is that no source code is required, but only debug symbols (e.g., PDB files) [17]. Still, all control transfers need to be checked. Even worse, it breaks self-checksumming or signed code and cannot be applied to self-modifying programs. Dynamic binary instrumentation is another alternative that can handle even stripped binaries (no need for source code or debug symbols), but the runtime performance overhead of existing binary instrumentation frameworks slows down the normal execution of an application by a factor of a few times [29].

In contrast to the above approaches, our system monitors the executed indirect branch instructions using Last Branch Recording (LBR) [39, Sec. 17.4], a recent feature of Intel processors introduced in the Nehalem architecture. When LBR is enabled, the CPU tracks the last N (16 for the CPU model we used) most recent branches in a set of 64-bit model-specific registers (MSR). Each branch record consists of two MSR registers, which hold the linear addresses of the branch instruction and its target instruction, respectively. Records from the LBR stack can be retrieved using a special instruction (`rdmsr`) from privileged mode. The processor can be configured to track only a subset of branches based on their type: relative/indirect calls/jumps, returns, and so on.

Table 1 shows a summarized comparison of the alternative strategies discussed above. For our particular case, the use of LBR has several advantages: it incurs zero overhead for storing the branches; it is fully transparent to the running applications; is does not cause any incompatibility issues as it is completely decoupled from the actual execution; it does not require source code or debug symbols; and it can be dynamically enabled for already installed applications—there is no need for recompilation or instruction-level instrumentation.

2.2 Using Last Branch Recording for ROP Prevention

Although the CPU continuously records the most recent branches in the LBR stack with zero overhead, accessing the LBR registers and retrieving the recorded information unavoidably adds some overhead. Considering the limited size (16 entries) of the LBR stack, and that it can be accessed only from kernel-level code, checking the targets of all indirect control transfer instructions would incur a prohibitively high performance overhead. Indirect branches occur very frequently in typical programs, and a monitored process should be interrupted once every 16 branches with a context switch. In fact, the implementation of such a scheme is not facilitated by the current design of the LBR feature, as it does not provide any means of interrupting execution whenever the stack gets full after retrieving its previous 16 records.

Fortunately, when considering the actual operations of a ROP exploit, it is possible to dramatically reduce the number of control transfer instructions that need to be inspected. The typical end goal of malicious code is to give the attacker full control of the victim system. This usually involves just a few simple operations, such as dropping and executing a malicious executable on the victim system, which unavoidably require interaction with the OS through the system call interface. Based on this observation, we can refine the set of indirect branches that need to be inspected to only those along the final part of the execution path that lead to a system call invocation. (Depending on the vulnerable program, exploitation might be possible without invoking any system call, e.g., by modifying a user authentication variable [25], but such attacks are rarely found in the client-side applications that are typically targeted by current ROP exploits, and are outside the scope of this work.)

Figure 1 illustrates this approach. Vertical bars correspond to snapshots of the address space of a process, and arrows correspond to indirect control transfers. The ver-

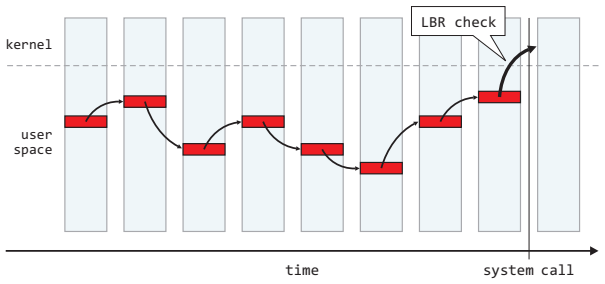

Figure 1: Illustration of a basic scheme for ROP code detection. Whenever control is transferred from user to kernel space (vertical line), the system inspects the most recent indirect branches to decide whether the system call was invoked by ROP code or by the actual program.

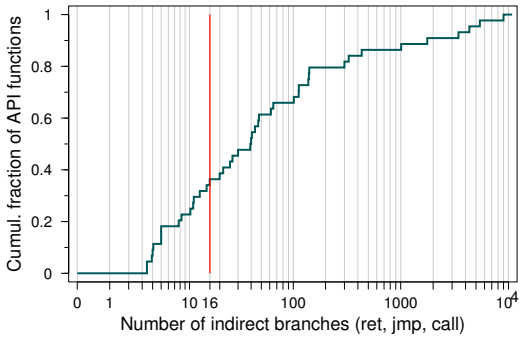

Figure 2: LBR overwriting due to indirect branches that take place within Windows API functions, prior to the execution of a system call.

tical line denotes the point at which the flow of control is transferred from user space to kernel space through a system call. At this point, by interposing at the OS's system call handler, the system can access the LBR stack and retrieve the targets of the indirect branches that led to the system call. It can then check the control flow path for abnormal control transfers and distinctive properties of ROP-like behavior using the techniques that will be described in Sec. 3, and decide whether the system call is part of malicious ROP code, or it is being invoked legitimately by the actual program.

2.2.1 System Calls vs. API Calls

User-level programs interact with the underlying system mainly through system calls. Unix-like systems provide to applications wrapper functions for the available system calls (often using the same name as the system call they invoke) as part of the standard library. In contrast, Windows does not expose the system call interface directly to user-level programs. Instead, programs interact with the OS through the Windows API [13], which is organized into several DLLs according to different kinds of functionality. In turn, those DLLs call functions from the undocumented Native API [59], implemented in `ntdll.dll`, to invoke kernel-level services.

Exploit code rarely relies on the Native API for several reasons. One problem is that system call numbers change between Windows versions and service pack levels [18, 14], reducing the reliability of the exploit across different targets (or increasing attack complexity by having to adjust the exploit according to the victim's OS version). Most importantly, the desired functionality is often not conveniently exposed at all through the Native API, as for example is the case with the socket API [65]. Typically, the purpose of ROP code is to give execute permission to a second-stage shellcode using `VirtualProtect` or a similar API function [31, 27, 1, 6, 7, 2]. The second-stage shellcode can be avoided altogether by implementing all the necessary functionality solely using ROP code, as is the case with a recent exploit against Adobe Reader XI, in which the ROP code calls directly the `fsopen`, `write`, `fclose`, and `LoadLibraryW` functions to drop and execute a malicious DLL [19].

The implementation of many of the functions exported by the Windows API is quite complex, and often involves several internal functions that are executed before the invocation of the intended system call. Due to the limited size of the LBR stack, this means that by the time execution reaches the actual system call, the LBR stack might be filled with indirect branches that took place *after* the Windows API function was called. To assess the extent of this effect, we measured the average number of indirect branch instructions (`ret`, `jmp`, and `call`) that are executed between the first instruction of a Windows API function and the system call it invokes, for a set of 52 "sensitive" functions that are commonly used in Windows shellcode and ROP code implementations (a complete list of the tested functions is provided in the appendix). As shown in Fig. 2, about 34% of the API functions execute less that 16 indirect branches, while the rest of them completely overwrite the LBR stack.

As these branches are made as part of legitimate execution paths, calling a function that completely overwrites the LBR stack would allow ROP code to evade detection. However, this scheme can be improved to provide robust detection of ROP code that calls *any* sensitive API function, irrespectively of the extent of overwriting in the LBR stack due to code in the function body.

2.2.2 LBR Stack Inspection on API Function Entry

Given that i) exploit code usually calls Windows API functions instead of directly invoking system calls, and ii) most API functions overwrite the LBR stack with legitimate indirect branches before invoking a system call,

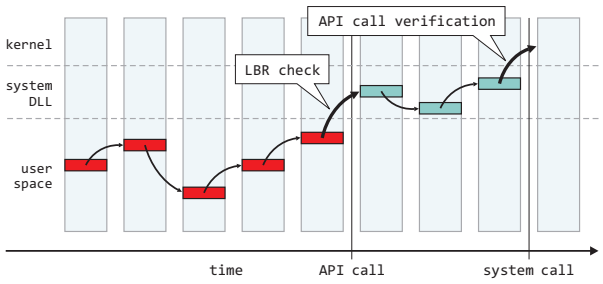

Figure 3: Overview of the detection scheme of kBouncer. Before the invocation of protected Windows API functions, the system inspects the LBR stack to identify whether the execution path that led to the call was part of ROP code, and writes a checkpoint. To account for ROP code that would bypass the check by jumping over kBouncer's function hook, the system then verifies the entry point of the API function at the time of the corresponding system call invocation.

kBouncer inspects the LBR stack at the time an API function is called, instead upon system call invocation. This allows the detection of ROP code that uses any sensitive API function, irrespectively of the number of legitimate indirect branches executed within its body. In case an API function is called by ROP code, all entries in the LBR stack at the time of function entry will correspond to the indirect branches of the gadgets that lead to the function call, as depicted in Fig. 3.

Still, without any additional precautions, this scheme would allow an attacker to bypass the LBR check at the entry point of a function. An implementation of the LBR check in the system call handler—within the kernel—safeguards it from user-level code and any bypass attempt. In contrast, implementing the LBR check as a hook to a user-level function's entry point does not provide the same level of protection. An attacker could avoid the check by jumping over the hook at the function's prologue, instead of jumping at its main entry point, and then normally executing the function body. Alternatively, by trading off some of its reliability, the ROP code could avoid calling the API function altogether by invoking directly the relevant Native API call.

Fortunately, as the Native API is not exposed to user-level programs, i.e., applications never invoke Native API calls directly; we can solve this issue by ensuring that system calls are *always* invoked solely through their respective Windows API functions. After a clear LBR check at an API function's entry point, kBouncer writes a checkpoint that denotes a legitimate invocation of that particular function. When the respective system call is later invoked, the system call handler verifies that a proper checkpoint was previously set by the expected API function, and clears it. If the checkpoint was not set, then this means that the flow of control did not pass through the proper API function preamble, and kBouncer reports a violation.

We should note that user-level ROP code cannot bypass kBouncer's checks by faking a checkpoint. The code for setting a checkpoint can only run with kernel privileges, and the checkpoint itself is stored in kernel space so that i) the system call handler can later access it, and ii) any user-level code (and consequently the ROP code itself) cannot tamper with it. The checkpoint code is tied with and comes right after the code that inspects the LBR stack, and both run in an atomic way at kernel level, i.e., the checkpoint cannot be set without previously analyzing the LBR for the presence of ROP code. This prevents any ROP code from faking a checkpoint without being detected—the part of the ROP code with the task of setting the checkpoint would be detected by the LBR check before the checkpoint is actually set.

3 Identifying the Execution Behavior of ROP Code

Before allowing a Windows API function call to proceed, kBouncer analyzes the most recent indirect branches that were recorded in the LBR cache prior to the function call. LBR is configured to record only `ret`, indirect `jmp`, and indirect `call` instructions. The execution of ROP code is identified by looking for two prominent attributes of its runtime behavior: i) illegal `ret` instructions that target locations not preceded by call sites, and ii) sequences of relatively short code fragments "chained" through any kind of indirect branches.

Returns that do not transfer control right after call sites is an illegitimate behavior exhibited by all publicly available ROP exploits against Windows software, which rely mainly on gadgets ending with `ret` instructions (`ret` conveniently manipulates both the program counter and the stack pointer). The second, more generic attribute captures an inherent property of not only purely return-oriented code, but also of advanced (and admittedly harder to construct) jump-oriented code (or even "hybrid" ROP/JOP code that might use any combination of gadgets ending with `jmp`, `call`, and `ret` instructions).

3.1 Illegal Returns

When focusing on the control flow behavior of ROP code at the instruction level, we expect to observe the successive execution of several `ret` instructions, which correspond to the transfer of control from each gadget to the next one. Although this control flow pattern is quite distinctive, the same pattern can also be observed in legitimate code, e.g., when a series of functions consecutively

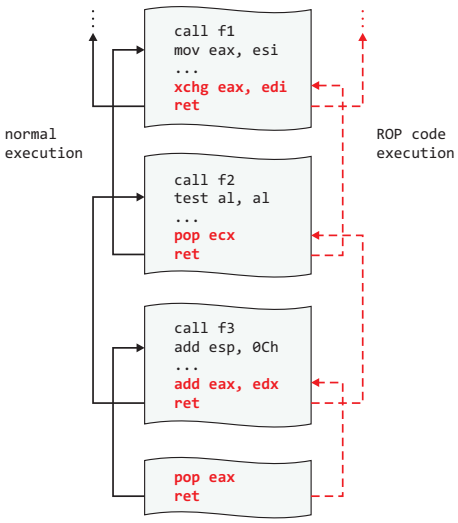

Figure 4: In normal code, `ret` instructions target valid call sites (left), while in ROP code, they target gadgets found in arbitrary locations (right).

return to their callers. However, when considering the *targets* of `ret` instructions, there is a crucial difference.

In a typical program, `ret` instructions are paired with `call` instructions, and thus the target of a legitimate `ret` corresponds to the location right after the call site of the respective caller function, i.e., an instruction that follows a `call` instruction, as illustrated in the left part of Fig. 4. In contrast, a `ret` instruction at the end of a gadget transfers control to the first instruction of the following gadget, which is unlikely to be preceded by a `call` instruction. This is because gadgets are found in arbitrary locations across the code image of a process, and often may correspond to non-intended instruction sequences that happen to exist due to overlapping instructions [62].

At runtime, the `ret` instructions of ROP code can be easily distinguished from the legitimate return instructions of a benign program by checking their targets. A `ret` instruction that transfers control to an instruction not preceded by a `call` is considered illegal, and the observation of an illegal `ret` is flagged by kBouncer as an indication of ROP code execution.

Ensuring `call-ret` pairing by verifying caller-callee semantics, e.g., using a shadow stack [29], constrains the control flow of a process in a much stricter way than the proposed scheme. In practice, though, enforcing such a strict policy is problematic, due to the use of `setjmp/longjmp` constructs, `call/pop` "getPC" code commonly found in position-independent executables, tail call optimizations, and lightweight user-level threads such as Windows fibers, in which the context switch function called by the current thread returns to the thread that is scheduled next.

Instead of enforcing a strict control flow, kBouncer simply makes sure that `ret` instructions always target *any* among all valid call sites (even those that correspond to non-intended `call` instructions). This is a more relaxed constraint that is not expected to be violated (and which did not, for the set of applications tested as part of our experimental evaluation) even in programs that use constructs like the above. Its implementation is also much simpler, as there is no need to track the execution of `call` instructions—checking that the target of each `ret` falls right after a `call` is enough.

Call-preceded Gadgets Although the above scheme prohibits the execution of illegal returns, which are prominently exhibited by typical ROP exploits, an attacker might still be able to construct functional ROP code using gadgets that begin right after `call` instructions, to which we refer as *`call`-preceded gadgets*. Note that `call`-preceded gadgets may begin after either intended or unintended `call` instructions. As kBouncer cannot know which `call` instructions were actually emitted by the compiler, if any of the possible valid instructions immediately preceding the instruction at a target address is a `call` instruction, then that address may correspond to the beginning of a `call`-preceded gadget.

The observation of a `ret` that targets an instruction located right after a `call` is considered by kBouncer as normal, and thus ROP code comprising only `call`-preceded gadgets would not be identified based on the first ROP code attribute kBouncer looks for during branch analysis. Although such code would still be identified due to its "chained gadgets" behavior, which we will discuss below, we first briefly explore the feasibility of such an attempt.

For our analysis we use a set of typical Windows applications, detailed in Table 2. The data is collected using a purpose-built execution analysis framework, described in Sec. 4.2. We consider as a gadget any (intended or unintended) instruction sequence that ends with an indirect branch, and which does not contain any privileged or invalid instruction. In contrast to the gadget constraints typically considered in relevant studies [62, 23, 60, 24, 73, 53, 36, 70] and the actual gadgets used in real exploits [27, 19, 1, 6, 7, 2], i.e., contiguous instruction sequences no longer than five instructions, we follow a more conservative approach and consider gadgets that i) may be *split* into several fragments due to internal conditional or unconditional relative jumps, and ii) have a maximum length of *20 instructions*.

Figure 5 shows the fraction of `call`-preceded gadgets among all gadgets that end with a `ret` instruction, for different Windows applications. In the worst case, only 6.4% of the gadgets begin right after call sites, a percentage much smaller compared to all available `ret` gad-

Application	Workload	Indirect Branches jump	call	ret	System Calls	Protected API Calls
Windows Media Player	Music playback for ~30 secs	7.3M	7.5M	30.0M	196K	5150
Internet Explorer 9	Browse to google.com	3.4M	4.8M	17.7M	87K	7092
Adobe Flash Player	Watch a youtube video	9.6M	21.7M	94.1M	317K	46658
Microsoft Word	Scroll through a document	3.3M	11.5M	38.8M	178K	5425
Microsoft Excel	Open a rich spreadsheet	6.3M	18.1M	54.5M	212K	3957
Microsoft PowerPoint	View a presentation	10.2M	19.3M	68.8M	275K	6577
Adobe Reader XI	Scroll through a few pages	9.7M	19.5M	100.6M	101K	5026

Table 2: Details about the dataset used for gadget analysis.

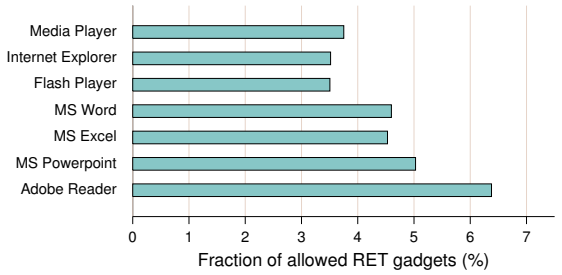

Figure 5: Among all gadgets that end with a `ret` instruction, only a small fraction (6.4% in the worst case for Adobe Reader) begin right after call sites.

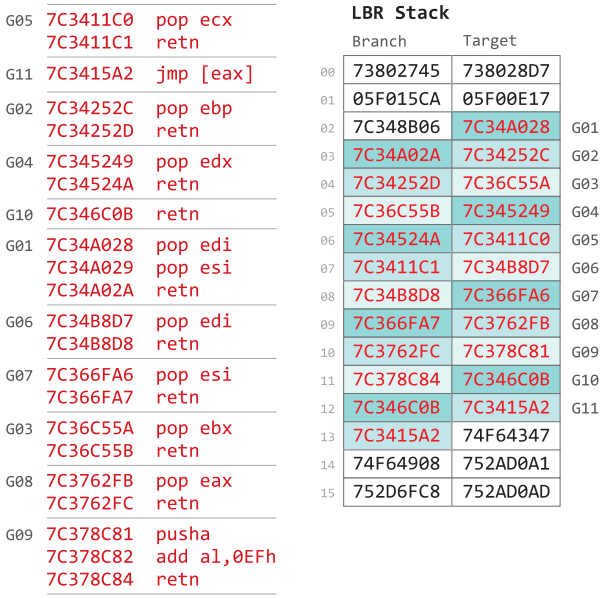

Figure 6: The state of the LBR stack at the time kBouncer blocks an exploit against Adobe Flash [2]. Diagonal pairs of addresses with the same shade correspond to the first and last instruction of each gadget.

gets. Given that many of them are longer than the typical gadget size, and are thus harder to use in ROP code due to the many different operations and register or memory state changes they incur, an attacker would be left with a severely limited set of gadgets to work with. For comparison, the ROP payloads of the exploits we used in our evaluation, listed in Table 4, collectively use 44 unique gadgets with an average length of just 2.25 instructions, and only *three* of them happen to be `call`-preceded—the rest of them would all result in illegal returns.

3.2 Gadget Chaining

It is clear from the previous section that even a "lighter" version of kBouncer that would just prohibit the execution of illegal returns would still significantly raise the bar, as i) it would prevent the execution of the ROP code typically found in publicly available Windows exploits, and more importantly, ii) it would force attackers to either use only a limited set of `ret` gadgets, or resort to jump-oriented code—options of increased complexity.

To account for potential future exploits of these sorts, the second attribute that kBouncer uses to identify the execution of ROP code is an inherent characteristic of its construction: the observation of several short instruction sequences *chained* through indirect branches. This is a generic constraint that holds for both return-oriented and jump-oriented code (or potential combinations—in the rest of this section we refer to both techniques as ROP).

Although legitimate programs also contain an abundance of code fragments linked with indirect branches, these fragments are typically much larger than gadgets, and more importantly, they do not tend to form long uninterrupted sequences (as we show below).

The CPU records in-sequence all executed indirect branches, enabling kBouncer to reconstruct the chain of gadgets used by any ROP code. Each LBR record $R[b,t]$ contains the address of the branch (b) and the address of its target (t), or from the viewpoint of ROP code, the *end* of a gadget and the *beginning* of the following one.

Figure 6 illustrates the contents of the LBR stack at the time kBouncer blocks the ROP code of an exploit against Adobe Flash [2] (although kBouncer blocks this exploit due to illegal returns, we use it for illustrative purposes, as we are not aware of any publicly available JOP exploit). Starting with the most recent (bottom-most) record, the detection algorithm checks whether the tar-

get (located at address $R_{n-1}[t]$) of the previous branch, is an instruction that precedes the branch (located at address $R_n[b]$) of the current record. If starting from address $R_{n-1}[t]$, there exists an uninterrupted sequence of at most 20 instructions that ends with the indirect branch at address $R_n[b]$, then the sequence is considered as a gadget. Recall that kBouncer treats as gadgets even fragmented instruction sequences linked through conditional or unconditional relative jumps. The same process repeats with the previous records, moving upwards, as long as chained gadgets are found.

The ROP code in this example consists of 11 gadgets, all ending with a `ret` instruction except the final one (G11), which is a single-instruction gadget with an indirect `jmp` that transfers control to `VirtualProtect` in `kernel32.dll` (note the difference in the high bytes of the target address in record 13). The two bottom-most records in the LBR stack correspond to kBouncer's function hook (from `VirtualProtect` to `DeviceIoControl`, which signals the kernel component), and a `ret` from `__SEH_prolog4` which is called by `DeviceIoControl`.

A crucial question for the effectiveness of the above algorithm is whether legitimate code could be misclassified as ROP code due to excessively long chains of gadget-like instruction sequences. To assess this possibility, we measured the length of the gadget chains observed across all inspected LBR stack instances for the applications and workloads listed in Table 2. As described in Sec. 2.2.2, kBouncer inspects the LBR stack right before the execution of a sensitive Windows API function. In total, kBouncer inspected 79,885 LBR stack instances, i.e., the tested applications legitimately invoked a sensitive API function 79,885 times.

Figure 7 (solid line) shows the percentage of instances with a given gadget chain length. In the worst case, there is just one instance with a chain of five gadgets, and there are no instances with six or more gadgets. On the other hand, complex ROP code that would rely on `call`-preceded or non-`ret` gadgets would result in excessively long gadget chains, filling the LBR stack. Indicatively, a jump-oriented Turing-complete JOP implementation for Linux uses 34 gadgets [23]. Furthermore, current JOP code implementations rely on a special dispatcher gadget that always executes between useful gadgets, at least doubling the amount of executed gadgets.

Although we can never rule out the possibility that benign code in some other application might result in a false positive, to ascertain that this possibility is unlikely, we also analyzed 97,554,189 LBR stack instances taken at the entry points of all executed functions during the lifetime of the same tested applications. In this orders-of-magnitude larger data set, the maximum gadget chain length observed is nine (dashed line), which is still far

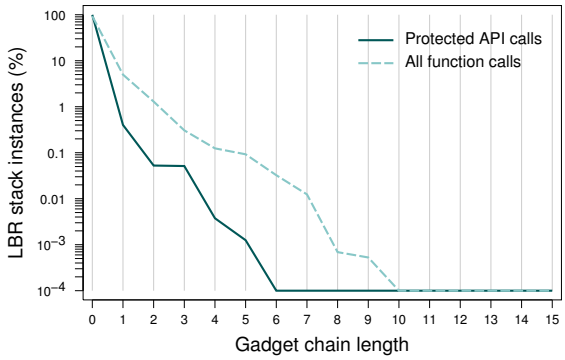

Figure 7: Percentage of LBR stack instances with a given gadget chain length for i) the instances inspected by kBouncer at the entry points of protected API function calls, and ii) the instances taken at the entry points of all function calls.

from filling up the LBR stack. This means that even if there is a need in the future to protect more API functions, or perform LBR checks in other parts of a program, we will more than likely still be able to set a robust detection threshold that will not result in false positives. For the current set of protected functions we use a threshold of eight gadgets, which allows for increased resilience to false positives.

Finally, note that in the above benign executions, the vast majority of the gadget-like chains stem from our conservative choice of considering *fragmented* gadgets of up to *20 instructions* long—significantly more complex and longer than the gadgets used in actual exploits. Although we could choose more reasonable constraints about what is considered as a gadget, we preferred to stress the limits of the proposed approach.

4 Implementation

4.1 kBouncer

To demonstrate the effectiveness of our proposed approach, we developed a prototype implementation for the x86 64-bit version of Windows 7 Professional SP1. Our prototype, kBouncer, consists of three components: i) an offline gadget extraction and analysis toolkit, ii) a user-space thin interposition layer between the applications and Windows API functions, and iii) a kernel module.

For the executable segments of a protected application, the gadget extraction toolkit identifies any instruction sequence ending in an indirect branch, starting from each and every byte of a segment. In the current version of our prototype we assume that the complete set of an application's modules is available in advance. However, it is possible to trivially relax this assumption by process-

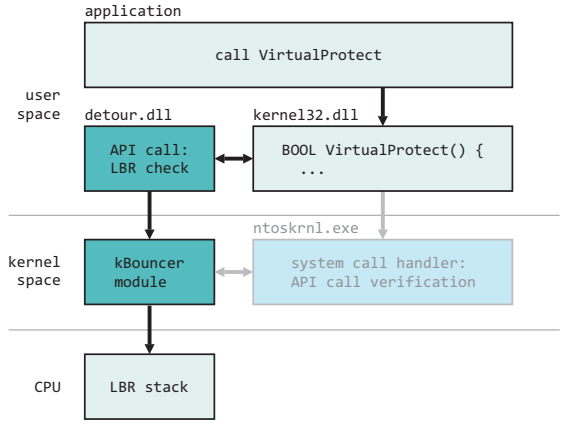

Figure 8: Overview of kBouncer's implementation. At the entry point of Windows API functions, kBouncer detours the execution, inspects the LBR stack in kernel mode, and then returns control back to the application.

ing new modules on-the-fly at the time they are loaded by a protected application. The maximum gadget length is given as a parameter—in our experiments we conservatively used a length of 20 instructions. As discussed in Sec. 3.1, our extraction algorithm differs from previous approaches as it considers even instruction sequences that contain conditional or unconditional relative jumps. For this reason, code analysis explores all possible paths from every offset within a code segment, and follows recursively any conditional branches. The output of the analysis phase is two hash tables: one containing the offsets of `call`-preceded gadgets, and another containing the rest of the found gadgets. In the future, we will consider switching to Bloom filters to save space.

The overall operation of the runtime system is depicted in Fig. 8. The interposition component is implemented on top of the Detours framework [38], which provides a library call interception mechanism for the Windows platform. During initialization, it requests by kBouncer's kernel module to enable the LBR feature on the CPU. The two components communicate through control messages over a pseudo-device that is exported by the kernel module (using the `DeviceIoControl` API function). Then, it selectively hooks the set of the protected Windows API functions. Each time a protected function is called, the detour code sends a control message to the kernel component, instructing it to inspect the contents of the LBR stack for abnormal control transfers.

The kernel module is responsible for three main tasks: i) enabling or disabling the LBR facility, ii) analyzing the recorded indirect branches, and iii) writing and verifying the appropriate checkpoint before allowing a system call to proceed. The first task involves reading and writing a few Model Specific Registers (MSR) using the `rdmsr` and `wrmsr` instructions. For the second task, whenever a control request is received from the user-space component, kBouncer analyzes the contents of the LBR stack, looking for the attributes described in Sec. 3. The MSR registers that hold the recorded information and configuration parameters are considered part of the running process context, and are preserved during context switches.

To identify illegal return instructions, the kernel module fetches a few bytes before each return target and attempts to decode any `call` instruction located right before the target instruction (call site check). Gadget chaining patterns are identified as follows: starting from the most recent branch in the LBR stack, the number of consecutive targets that point to gadgets are counted. Any `ret` targets are looked up in the `call`-preceded gadgets hash table, whereas `call` or `jmp` targets are looked up in both hash tables, `call`-preceded or not. The most recent branch target is not considered, as it does not point to a gadget, but to the protected API function. To protect the kernel-level component from potential crashes when accessing invalid user-level locations, we use the `ProbeForRead` function of the Windows kernel API.

Unfortunately, the final task for API call verification has been only partly implemented, as it is not possible to perform system-call interposition in the current version of Windows 7. A recently added kernel feature in the 64-bit version of Windows, called PatchGuard [32], protects against kernel-level rootkits by preventing any changes to critical data structures, such as the System Service Descriptor Table (SSDT). Although this is effective against rootkits, PatchGuard removed the ability of legitimate applications, such as antivirus software, to intercept system calls. In response, Microsoft added a set of kernel-level APIs for filtering network and file-system operations (Windows Filtering Platform [48]). Hopefully, future OS versions will provide system call filtering capabilities as well.

Still, we did verify the correct operation of checkpoint verification by simulating it using the dataset of Table 2. We should note that this is not a design limitation, but only an implementation issue stemming from our choice of the target platform. For example, this would not have been an issue had we decided to implement kBouner for Linux, or any other open platform. For now, we plan to implement the checkpointing functionality for 32-bit applications by hooking system calls at user level through the WOW64 layer [4] (which, however, will not provide the same protection guarantees as an actual kernel-level implementation).

In case an attack attempt is detected after the analysis of the recorded branches, the process is terminated and the user is informed with an alert message, as shown in Fig. 9. In this example, kBouncer blocks a malicious PDF sample that exploits an (at the time of writing)

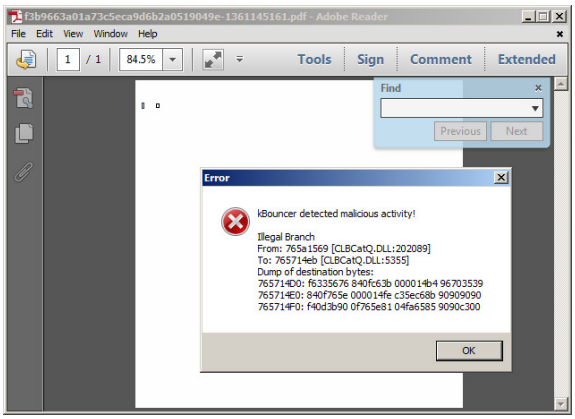

Figure 9: A screen capture of kBouncer in action, blocking a zero-day exploit against Adobe Reader XI [19].

unpatched vulnerability in the latest version of Adobe Reader XI [19]. The displayed information, such as branch locations and targets, is supplied from the kernel-level module.

4.2 Analysis Framework

Moving from the basic concept to a functional prototype required a number of decisions that were mostly based on analyzing the behavior of large applications. To ease the effort required to perform this type of analysis, we developed an LBR analysis framework. Its goal is to provide a way to iterate over the LBR instances during the lifetime of an application, while at the same time providing useful information, such as translating addresses to function or image names. The framework is split in two parts: data gathering and analysis.

The data-gathering component is based on dynamic binary instrumentation. Although the runtime overhead of dynamic instrumentation is quite high (as discussed in Sec. 2.1), we use it here only for data gathering, which is an off-line and one-time operation. The tool we developed is built on top of Pin [64, 46], and records the following information during process execution: i) the file path and starting and ending address of any loaded executable image, ii) the location and name of any recognized function (e.g., exported functions), iii) the thread ID, location, and target of executed indirect branches (`ret`, `call` or `jmp`), iv) the thread ID, location, and number of system calls, and v) the thread ID, location, and return address of any identified function that was called.

The analysis part is a set of Python scripts that process the gathered data for each application. It provides a configurable LBR iterator which simulates different scenarios, such as returning LBR stack instances before system calls or certain function calls, or even after each branch is executed. To avoid mixing branches from different system threads in the same LBR instance, it internally keeps a list of separate LBRs per thread id. Finally, it provides convenient methods to translate addresses to function or image names when available.

Type	# iter.	Avg. Total Time ms (stddev)	Single ns
HashLookup	1B	8231.6 (9.8)	8.2
IllegalRet	1B	10889.9 (312.9)	10.8
SysNull	10M	5145.0 (66.0)	514.5
SysLBR	10M	19981.8 (504.5)	1998.1
SysRead	10M	47267.7 (30925.6)	4726.7

Table 3: Microbenchmarks.

5 Evaluation

In this section we present the results of our experimental evaluation of kBouncer in terms of runtime overhead and effectiveness against real-world ROP exploits. All experiments were performed on a computer with the following specifications: Intel i7 2600S CPU, 8GB RAM, 128GB SSD, 64-bit Windows 7 Professional SP1.

5.1 Performance Overhead

5.1.1 Microbenchmarks

We started with some micro-benchmarks of different parts of kBouncer's functionality. Specifically, we measure the average time needed for the following operations, also listed in Table 3: hash table lookups ("HashLookup"), checks for illegal returns ("IllegalRet"), performing a system call ("SysNull"), reading the contents of the LBR stack ("SysLBR"), and reading parts of a process' address space ("SysRead").

In each case, we isolated the measured operation and tried to make the experiment as realistic as possible. For example, we extracted the hash table characteristics (domain size, hash table size, hit ratio) based on the dataset shown in Table 2. The data we used for the illegal return checks come from `kernel32.dll`, and use a worst-case workload by treating each location in its code segment as a possible return target. The next three experiments where measured in kernel level, as opposed to the first two. We measured the time needed to perform a no-op system call, a system call that only reads the LBR stack contents, and finally, a system call that in addition to reading the LBR stack, also fetches data from the sources and targets of each branch.

Table 3 shows the results of these benchmarks. Each benchmark runs the number of operations shown in the second column ten times, and calculates the average and

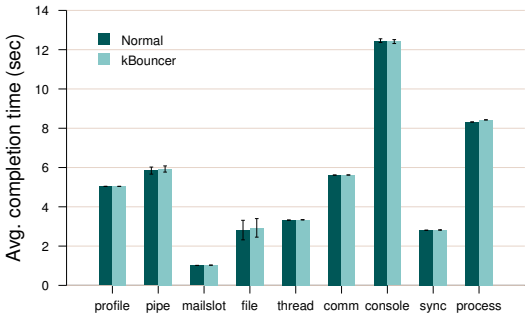

Figure 10: Execution time with and without kBouncer for Wine's `kernel32.dll` test suite, which resulted in the invocation of about 100K monitored Windows API functions. The average runtime overhead is 1%.

standard deviation (next two columns). The last column shows the average time for a single operation. As we can see, looking up the hash table and checking for an illegal return are both very fast operations, in the order of a few nanoseconds. Performing a system call and reading the LBR stack are relatively more expensive, but still, in the order of a few microseconds. When attempting to access the instructions located at the source and target addresses of each branch record, the measured duration starts to fluctuate. We are not sure whether this behavior is normal, or it is a result of non-optimal use of the kernel API for accessing user-level memory. Overall, these microbenchmarks show that kBouncer's LBR stack analysis on each protected API function call takes on average no more than 5 microseconds.

5.1.2 Runtime Overhead

Measuring the performance overhead impact on interactive applications, such as web browsers and document viewers, is a challenging task. Instead, we decided to measure the performance overhead on programs that stress the core functionality of kBouncer, by making heavy use of the monitored Windows API functions. For this purpose, we used a subset of the tests provided in the test suite of Wine [15], which repeatedly call Windows API functions with different arguments. To get more confident timing results, we kept only tests that do not interfere with external factors, such as network communication. The final set we used performs about 100,000 calls to Windows API functions that are protected by kBouncer, which is 20 times more than the protected calls made by the actual applications we previously tested (listed in Table 2).

Figure 10 shows the completion time for each of the different tests, with and without kBouncer. The average runtime overhead is 1%, with the maximum being 4% in the worst case. The total extra time spent across all tests when enabling kBouncer was 0.3 sec, a result consistent with the average cost of 5 μs per check based on our microbenchmarks (100,000 calls $\times$ 5 μs = 0.5 sec). Based on these results, which show that the performance overhead is negligible even for workloads that continuously trigger the core detection component, we believe that kBouncer is not likely to cause any noticeable impact on user experience.

Application	Vulnerability
Adobe Reader v11.0.1	Function pointer overwrite [19]
Adobe Reader v9.3.4	Stack-based overflow [1]
MPlayer Lite r33064	SEH pointer overwrite [6]
Internet Explorer 8	Use-after-free vulnerability [8]
Internet Explorer 9	Use-after-free vulnerability [7]
Adobe Flash 11.3.300	Integer overflow [2]

Table 4: Tested ROP exploits.

5.2 Effectiveness

In the final part of our evaluation, we tested whether our prototype can effectively protect applications that are typically targeted by in-the-wild attacks, using the ROP exploits shown in Table 4. All exploits except the ones against Internet Explorer work on the latest and up-to-date version of Windows 7 Professional SP1 64-bit. For the IE exploits to work, we had to uninstall the updates that fixed the relevant vulnerabilities (KB2744842 and KB2799329). We also had to tweak the ROP payload of the MPlayer exploit to correctly calculate the offset of `VirtualProtect` for the latest version of `kernel32.dll`, as the public version of the exploit was based on a previous version of that DLL.

The ROP code in the exploit against Adobe Reader v9.3.4 creates a file (`CreateFileA`), memory-maps the file in RWX mode (`CreateFileMappingA`, `MapViewOfFile`), copies the shellcode in the newly mapped area, and executes it. Similarly, the MPlayer and IE 8 exploits change the permissions of the memory region where the shellcode resides to RWX (`VirtualProtect`) and execute it. What is interesting about the IE 8 ROP code, is that it is constructed from the statically loaded Skype protocol handler DLL (`skype4com.dll`). The last two exploits in Table 4 were generated using the Metasploit Framework [5]. For vulnerable applications that include widely used non-ASLR modules (like Java's `msvcrt71.dll`, which is loaded in Internet Explorer), Metasploit uses the same ROP payload based on `msvcrt71.dll`, which has been pre-generated by Mona [27]. This payload is similar to the one used in the MPlayer exploit, as it also uses `VirtualProtect` to bypass Data Execution Prevention (DEP). Finally, the Adobe Reader XI (v11.0.1) exploit is more complex,

as it is the first in-the-wild exploit that uses ROP-only code, i.e., it does not carry any shellcode [19]. The malicious sample we tested ("Visaform Turkey.pdf") exploits a first vulnerability to escape from Reader's sandboxed process, and a second one to hijack the execution of its privileged process by loading a malicious DLL using `LoadLibraryW`.

In the first five exploits, the embedded shellcode simply invokes `calc.exe` using `WinExec`. The Reader XI exploit drops a malicious DLL. In all cases, we verified that the exploits worked properly on our testbed, by confirming that the calculator was successfully launched, or, for the Reader XI exploit, that the malicious DLL was loaded successfully. When kBouncer was enabled, it successfully blocked all exploits due to the identification of illegal returns at the time one of the `CreateFileA`, `VirtualProtect` or `LoadLibraryW` functions was invoked by the ROP code in each case.

6 Limitations

The Last Branch Recording feature of recent Intel processors is what enables kBouncer to achieve its transparent and low-overhead operation. Many of our design decisions are corollaries of the very limited size of the LBR stack, which in the most recent processors holds only 16 records. Given that previous processor generations had even more size-constrained LBR implementations, this is definitely a significant improvement, and hopefully future processors will support even larger LBR stacks. This would allow kBouncer to achieve even higher accuracy by inspecting longer execution paths, making potential evasion attempts even harder.

Currently, an attacker could evade kBouncer by ensuring that the final 16 executed gadgets before the invocation of an API function are considered legitimate. Specifically, given that kBouncer looks for both illegal returns and gadget chaining in parallel, this would require i) all 16 gadgets to be either `call`-preceded or non-`ret` gadgets, and ii) at least one out of every eight of them (eight is our current gadget chaining detection threshold) to be longer than 20 instructions.

A more thorough analysis on the feasibility of constructing such a payload for typical applications is part of our future work. Our preliminary evidence (Section 3.1), however, shows that only 6.4% of all gadgets ending with `ret` are `call`-preceded, and this is when considering even fragmented gadgets up to 20 instructions long (this percentage drops to 3% when considering gadgets with at most five instructions). On the other hand, ROP compilers like Q [60] typically take into account non-fragmented gadgets up to five instructions long. Longer gadgets incur more CPU state changes, which complicate the (either manual or automated) gadget arrangement process. Indicatively, for a similar set of applications, even when 20% of all gadgets are available, Q could not generate a functional payload [53]. Note that the selection of a maximum gadget length of 20 instructions was arbitrary—four times the typically used standard seemed enough. If evasion becomes an issue, longer gadgets could be considered during the gadget chaining analysis of an LBR snapshot.

Alternatively, an attacker could look for a long-enough execution path that leads to the desired API call as part of the application's logic. Such a path should satisfy the following constraints: i) contain at least 16 indirect branches, the targets of which happen to lead to the execution of the desired API function, and ii) the executed code along the path should not alter the state or the function arguments set by the previously executed ROP code. Finding such a path seems quite challenging, as in many cases the desired function might not be imported at all, and the path should end up with the appropriate register values and arguments to properly invoke the function. This is even more difficult in 64-bit systems, where the first four parameters are passed trough registers, as opposed to the 32-bit standard calling conventions in which parameters are passed through the stack.

Our selection of sensitive Windows API functions was made empirically based on a large set of different shellcode and ROP payload implementations [5, 3, 56, 12, 27, 60]. A list of the 52 currently protected functions is provided in the appendix. Although current ROP exploits rely mainly on only a handful of API functions (see Sec. 5.2), we have included many others that have been used in the past in legacy shellcode, as some exploits might implement their whole functionality using purely ROP code (as demonstrated recently by an exploit against the latest version of Adobe Reader XI [19]). The set of protected functions can be easily extended with any additional potentially sensitive functions that we might have left out. Although it would be possible to protect all Windows API calls, we believe that this would not offer any additional protection benefits, and would just introduce unnecessary overhead.

7 Related Work

Address Space Randomization and Code Diversification As code-reuse attacks require precise knowledge of the structure and location of the code to be reused, diversifying the execution environment or even the program code itself is a core concept in preventing code-reuse exploits [26, 33]. Address space layout randomization [55, 49] is probably one of the most widely deployed countermeasures against code-reuse attacks. However, it's effectiveness is hindered by code segments left in static locations [35, 75, 40], while, depending on the ran-

domization entropy, it might be possible to circumvent it using brute-force guessing [63]. Even if all the code segments of a process are fully randomized, vulnerabilities that allow the leakage of memory contents can enable the calculation of the base address of a DLL at runtime [19, 61, 44, 69, 37, 66].

Intra-DLL randomization at the function [20, 21, 42, 9], basic block [11, 10], or instruction level [53, 36, 70] can provide protection for executables that do not support ASLR, or against de-randomization attacks through memory leaks. The practical deployment of these techniques for the protection of third-party applications depends on the availability of source code [20, 21, 42, 9], debug symbols [11, 10], or the accuracy of disassembly and control flow graph extraction [53, 36, 70, 74].

As kBouncer is completely transparent to user applications, it can complement all above randomization techniques as an additional mitigation layer against ROP exploits, while it does not depend on source code, debug symbols, or code disassembly.

Control Flow Integrity and Indirect Branch Protection The execution of ROP code disrupts the normal call path of typical programs, resulting to an unanticipated flow of control. Control flow integrity [17] can confine program execution within the bounds of a precomputed profile of allowed control flow paths, and thus can prevent most of the irregular control flow transfers that connect the gadgets of a ROP exploit. Depending on program complexity, however, deriving an accurate view of the control flow graph is often challenging. Alternative approaches against return-oriented programming enforce a more relaxed policy for the integrity of indirect control transfers [52, 45, 22]. Using code transformations, these techniques eliminate the occurrence of unintended indirect branch instructions in the generated code, and safeguard all legitimate indirect branches using cookies or additional levels of indirection.

The main factor that limits the practical applicability of the above techniques is that they require the recompilation of the target application, which is usually not possible for the popular proprietary applications that are commonly targeted by ROP exploits. In contrast, kBouncer is completely transparent to applications and does not require any modification to their code.

Runtime Execution Monitoring Many defenses against return-oriented programming are based on monitoring program execution at the instruction level. A widely used mechanism for this purpose is dynamic binary instrumentation (DBI), using frameworks such as Pin [46]. DROP [24] and DynIMA [28] follow this approach to monitor the frequency of `ret` instructions, and raise an alert in case irregularly many of them are observed within a small window of executed instructions. ROPdefender [29] also uses DBI to keep a shadow stack that is updated by instrumenting `call` and `ret` instructions. A disruption of the expected `call`-`ret` pairs due to ROP code is detected by comparing the shadow stack with the system's stack on every function exit. A limitation of the above techniques is that they cannot prevent exploits that use gadgets ending with indirect `jmp` or `call` instructions. More importantly, though, the significant runtime overhead imposed by the additional instrumentation instructions and the DBI framework itself limit their practical applicability.

Similarly to kBouncer, ROPGuard [34] is based on the observation that a ROP exploit will eventually invoke critical API functions, and performs various checks before such a function is called. These include checking whether `esp` is within the proper stack boundaries, whether a proper return address is present at the top of the stack, the consistency of stack frames, and other function-specific attributes. Although ROPGuard focuses only on non-JOP code, and some of its checks can result in false positives or can be easily evaded [58, 57], they are effective against current in-the-wild exploits, and some have been integrated in EMET [47].

Last branch recording is only one of the available instruction tracing facilities available in modern CPUs. Branch Trace Storage (BTS) is a debugging mechanism that enables the recording of all branch instructions in a user-defined memory area. However, the overhead due to the significant number of memory accesses, combined with the overall slower operation of the processor due to the special debug mode in which it enters when BTS is enabled, result to slowdowns typically in the range of 20–40× [67]. Consequently, systems that use BTS and similar mechanisms for control flow integrity [72, 73] or execution recording [68] suffer from significant runtime overheads. In contrast, LBR uses on-chip registers to store the traced branches with no additional overhead.

A recent technique against kernel-level ROP uses the processor's performance counters to raise an interrupt after a number of mispredicted `ret` instructions, an indication of possible ROP code execution [71]. To rule out mispredicitons caused by legitimate code, upon an interrupt, the LBR stack is used to check whether the targets of the previously executed `ret` instructions are preceded by a `call` instruction. The use of JOP or call-preceded gadgets, however, can circumvent this protection.

Branch regulation [41] is a proposal for extending current processor architectures with a protection mechanism against ROP attacks. Besides maintaining a secondary call stack, the technique restricts the allowed targets of indirect `jmp` instructions to locations within the same function, or to the entry point of any other function, and only the latter for `call` instructions. Besides being quite

restrictive for many legitimate programs, this approach requires protected binaries to go through a static binary instrumentation phase for annotating function boundaries, a process that requires precise code disassembly.

8 Conclusion

Exploit mitigation add-ons that can be readily enabled for the protection of already installed applications are among the most practical ways for deploying additional layers of defenses on existing systems. To be usable in practice, any such solution should be completely transparent and should not impact in any way the normal operation of the protected applications.

Starting on this basis, we have presented the design and implementation of kBouncer, a transparent ROP exploit mitigation based on the identification of distinctive attributes of return-oriented or jump-oriented code that are inherently exhibited during execution. Built on top of the Last Branch Recording (LBR) feature of recent processors for tracking the execution of indirect branches at critical points during the lifetime of a process, kBouncer introduces negligible runtime overhead, and does not require any modifications to the protected applications. We believe that the most important advantage of the proposed approach is its practical applicability. We demonstrate that our prototype implementation for Windows 7 can effectively protect complex, widely used applications, including Internet Explorer, Adobe Flash Player, and Adobe Reader, against in-the-wild ROP exploits, without any false positives.

As part of our future work, we plan to perform a more extensive evaluation with real applications to ensure the compatibility of the detection checks with existing code, assess the feasibility of constructing ROP payloads that could evade the currently implemented checks, and port our prototype implementation to Linux.

Acknowledgements

This work was supported by DARPA, the US Air Force, and ONR through Contracts DARPA-FA8750-10-2-0253, AFRL-FA8650-10-C-7024 and N00014-12-1-0166, respectively, with additional support from Intel. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, the Air Force, ONR, or Intel.

References

[1] Adobe CoolType SING Table "uniqueName" Stack Buffer Overflow. http://www.metasploit.com/modules/exploit/windows/fileformat/adobe_cooltype_sing.

[2] Adobe Flash Player 11.3 Kern Table Parsing Integer Overflow. http://www.metasploit.com/modules/exploit/windows/browser/adobe_flash_otf_font.

[3] Common Shellcode Naming Initiative. http://nepenthes.carnivore.it/csni.

[4] Intercepting System Calls on x86_64 Windows. http://jbremer.org/intercepting-system-calls-on-x86_64-windows/.

[5] Metasploit framework. http://www.metasploit.com.

[6] Mplayer (r33064 lite) buffer overflow + rop exploit. http://www.exploit-db.com/exploits/17124/.

[7] MS12-063 Microsoft Internet Explorer execCommand Use-After-Free Vulnerability. http://www.metasploit.com/modules/exploit/windows/browser/ie_execcommand_uaf.

[8] MS13-008 Microsoft Internet Explorer CButton Use-After-Free Vulnerability. http://www.greyhathacker.net/?p=641.

[9] /ORDER (put functions in order). http://msdn.microsoft.com/en-us/library/00kh39zz.aspx.

[10] Profile-guided optimizations. http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx.

[11] Syzygy - profile guided, post-link executable reordering. http://code.google.com/p/sawbuck/wiki/SyzygyDesign.

[12] White Phosphorus Exploit Pack. http://www.whitephosphorus.org/.

[13] Windows api list. http://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx.

[14] Windows X86 System Call Table. http://j00ru.vexillium.org/ntapi/.

[15] Wine. http://www.winehq.org.

[16] MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit, 2013. http://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/.

[17] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005.

[18] Piotr Bania. Windows Syscall Shellcode, 2005. http://www.securityfocus.com/infocus/1844.

[19] James Bennett, Yichong Lin, and Thoufique Haq. The Number of the Beast, 2013. http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html.

[20] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, 2003.

[21] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[22] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[23] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, 2010.

[24] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security (ICISS)*, 2009.

[25] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[26] Frederick B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12:565–584, October 1993.

[27] Corelan Team. Mona. `http://redmine.corelan.be/projects/mona`.

[28] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable Trusted Computing (STC)*, 2009.

[29] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A practical protection tool to protect against return-oriented programming. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[30] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`.

[31] Úlfar Erlingsson. Low-level software security: Attack and defenses. Technical Report MSR-TR-07-153, Microsoft Research, 2007. `http://research.microsoft.com/pubs/64363/tr-2007-153.pdf`.

[32] Scott Field. An introduction to kernel patch protection. `http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx`.

[33] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[34] Ivan Fratric. Runtime prevention of return-oriented programming attacks, 2012. `https://code.google.com/p/ropguard/`.

[35] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

[36] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.

[37] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.

[38] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.

[39] Intel. Intel 64 and IA-32 architectures software developer's manual, volume 3B: System programming guide, part 2. `http://www.intel.com`.

[40] Richard Johnson. A castle made of sand: Adobe Reader X sandbox. CanSecWest, 2011.

[41] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 94 –105, 2012.

[42] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[43] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `http://www.suse.de/~krahmer/no-nx.pdf`.

[44] Haifei Li. Understanding and exploiting Flash ActionScript vulnerabilities. CanSecWest, 2011.

[45] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European conference on Computer Systems (EuroSys)*, 2010.

[46] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[47] Microsoft. The Enhanced Mitigation Experience Toolkit. `http://www.microsoft.com/emet`.

[48] Microsoft. Windows filtering platform. `http://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx`.

[49] Matt Miller, Tim Burrell, and Michael Howard. Mitigating software vulnerabilities, July 2011. `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788`.

[50] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), December 2001.

[51] Tim Newsham. Non-exec stack, 2000. `http://seclists.org/bugtraq/2000/May/90`.

[52] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[53] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, 2012.

[54] Parvez. Bypassing Microsoft Windows ASLR with a little help by MS-Help, August 2012. `http://www.greyhathacker.net/?p=585`.

[55] PaX Team. Address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`.

[56] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

[57] Aaron Portnoy. Bypassing all of the things. SummerCon, 2013.

[58] Dan Rosenberg. Defeating Windows 8 ROP Mitigation, 2011. `http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/`.

[59] Mark Russinovich. Inside native applications, November 2006. `http://technet.microsoft.com/en-us/sysinternals/bb897447.aspx`.

[60] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[61] Fermin J. Serna. CVE-2012-0769, the case of the perfect info leak, February 2012. `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf`.

[62] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, 2007.

[63] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security (CCS)*, 2004.

[64] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *International Symposium on Performance Analysis of Software and Systems*, 2010.

[65] Skape. Understanding windows shellcode, 2003. `http://www.hick.org/code/skape/papers/win32-shellcode.pdf`.

[66] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.

[67] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[68] A. Vasudevan, Ning Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS)*, 2011.

[69] Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. `http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf`.

[70] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, October 2012.

[71] Georg Wicherski. Taming ROP on Sandy Bridge. SyScan, 2013.

[72] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[73] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security breaches as PMU deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys)*, 2011.

[74] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.

[75] Dino A. Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010.

Appendix

In our current prototype implementation, kBouncer protects the following 52 Windows API functions:

```
kernel32.dll

  CloseHandle
  CreateFileA
  CreateFileMappingA
  CreateFileMappingW
  CreateFileW
  CreateProcessA
  CreateProcessW
  DeleteFileA
  DeleteFileW
  DuplicateHandle
  ExitProcess
  ExitThread
  GetCurrentProcess
  GetProcAddress
  GetSystemDirectoryA
  GetSystemDirectoryW
  GetTempPathA
  GetTempPathW
  LoadLibraryA
  LoadLibraryW
  PeekNamedPipe
  ReadFile
  SetUnhandledExceptionFilter
  Sleep
  VirtualAlloc
  VirtualProtect
  WaitForSingleObject
  WinExec
  WriteFile

ws2_32.dll

  accept
  bind
  closesocket
  connect
  ioctlsocket
  listen
  recv
  send
  socket
  WSASocketA
  WSASocketW
  WSAStartup

wininet.dll

  InternetOpenA
  InternetOpenUrlA
  InternetOpenUrlW
  InternetOpenW
  InternetReadFile

msvcrt.dll

  _execv
  fclose
  fopen
  fwrite

urlmon.dll

  URLDownloadToFileA
  URLDownloadToFileW
```