Evaluating the Effectiveness of Current Anti-ROP Defenses

Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz

Horst Görtz Institute for IT-Security (HGI), Ruhr-Universität Bochum

Abstract. Recently, many defenses against the offensive technique of *return-oriented programming* (ROP) have been developed. Prominently among them are *kBouncer*, *ROPecker*, and *ROPGuard* which all target legacy binary software while requiring no or only minimal binary code rewriting.

In this paper, we evaluate the effectiveness of these Anti-ROP defenses. Our basic insight is that all three only analyze a limited number of recent (and upcoming) branches in an application's control flow on certain events. As a consequence, an adversary can perform dummy operations to bypass all employed heuristics. We show that it is possible to generically bypass kBouncer, ROPecker, and ROPGuard with little extra effort in practice. In the cases of kBouncer and ROPGuard on Windows, we show that all required code sequences can already be found in the executable module of a minimal 32-bit C/C++ application with an empty main() function. To demonstrate the viability of our attack approaches, we implemented several proof-of-concept exploits for recent vulnerabilities in popular applications; e. g., Internet Explorer 10 on Windows 8.

Keywords: ROP, Exploit Mitigation, Memory Corruptions

1 Introduction

Defensive measures against memory corruption and control flow hijacking attacks have been considerably improved recently, especially on the software side. Widely deployed techniques such as *address space layout randomization* (ASLR) and *data execution prevention* (DEP) often greatly hamper successful exploitation of a given vulnerability or even succeed in preventing the attack at all. In many cases, however, an advanced and dedicated attacker is still able to ultimately bypass these defenses and achieve reliable exploitation [1, 7, 16, 27].

An offensive technique that is used in many of today's successful attacks is called *return-oriented programming* (ROP) [17, 28]. With this technique, an attacker does not inject her own shellcode as part of the attack payload, but she reuses existing code and chains small code fragments (so called *gadgets*) together that perform malicious computations. Due to the effectiveness of this approach, it comes as no surprise that in the last few years many defensive mechanisms specifically targeting ROP have been proposed. Three recent representatives of such methods are *kBouncer* [25], *ROPecker* [8], and *ROPGuard* [10].

In this paper, we evaluate the effectiveness of these proposed methods and demonstrate their limitations. We first analyze kBouncer, a defensive mechanism that aims at detecting and preventing ROP-based attacks against user mode applications on the Windows operating system. kBouncer leverages the *last branch recording* (LBR) feature incorporated in current AMD and Intel x86-64 CPUs [3, 15] to check for suspicious control flows. kBouncer received broad attention not only from the research community when its first version [24] was announced as the $200,000 winner of the *Microsoft BlueHat Prize* [2]. We show that kBouncer's latest version [25] can be circumvented in virtually all realistic 32-bit and 64-bit attack scenarios with little extra effort. More specifically, we demonstrate how three recent ROP-based exploits—e. g., for Microsoft *Internet Explorer* on Windows 8—can be modified to bypass kBouncer. Furthermore, we show that even the `.text` section of a minimal 32-bit C/C++ application compiled with Microsoft's Visual Studio contains all necessary gadgets required to bypass kBouncer. We demonstrate how successful attacks against kBouncer in practice often also circumvent ROPGuard. This method placed second at the BlueHat Prize and has since been incorporated into Microsoft's *Enhanced Mitigation Experience Toolkit* (EMET).

The third defensive measure we examine is ROPecker [8]. This approach was presented in 2014 and it also leverages the LBR feature to protect applications on Linux from ROP-based attacks. We show that ROPecker suffers from conceptual weaknesses similar to kBouncer. In its published form, ROPecker can be circumvented in a generic way by an adversary. We empirically verify our attack and demonstrate a successful low-overhead bypass for a recent vulnerability of the popular web server software *Nginx*.

In summary, the contributions of this paper are as follows:

- We discuss several kinds of commonly available 32-bit and 64-bit gadgets that an attacker can utilize to perform malicious computations in a way resembling benign control flow.
- We demonstrate that ROP defenses based on analyzing a limited number of branches can be bypassed by an attacker in a generic way and that such a bypass requires little extra effort.
- We empirically verify our proposed approaches and present successful attacks against the three recent ROP defenses *kBouncer*, *ROPecker*, and *ROPGuard*.
- We assess the practical susceptibility of kBouncer and ROPecker to false positive detections of attacks using independently implemented emulators. We discover for both schemes that false positives are not unlikely to occur for at least certain popular applications.

2 Technical Background

We briefly review the basic concepts behind *return-oriented programming* and *last branch recording* that are fundamental to understand the rest of the paper.

2.1 Return-Oriented Programming

Generally speaking, an attacker's goal is the execution of certain code (referred to as *shellcode*) of her choice in the context of a vulnerable application. Typically, an attacker initially exploits some kind of software bug (e.g., a memory corruption vulnerability or a dangling pointer) to hijack an application's control flow.

As DEP has become prevalent, adversaries often resort to reusing (native) code already present in an application instead of directly injecting new shellcode, for example via exploitation techniques such as *return-to-libc* [23] or *return-oriented programming* (ROP) [17, 28]. With ROP, small code fragments—called *gadgets*—ending in a *return* instruction are consecutively executed: an attacker can "program" the desired semantics by writing a chain of addresses of gadgets to the stack of one of the target application's threads in such a way that each gadget "returns" to its successor. This chain of gadgets is often referred to as *ROP chain*. On x86-64 platforms, gadgets can be *aligned* as well as *unaligned* with the original instruction stream produced by the compiler, as instructions may start at any offset into a code page. Typically, suitable gadgets for ROP attacks exist in sufficient quantities in most non-trivial applications [9,13]. There are also ROP-compilers [14, 31, 32] that can for example automatically convert a given shellcode into an application-specific ROP chain. Advanced techniques closely related to ROP have been presented that leverage gadgets not ending in return instructions but typically some kind of indirect jumps [4,6]. Accordingly, these techniques are also known as *jump-oriented programming* (JOP). In the following, we use the term *ROP* inclusively for *JOP*.

One widely deployed generic countermeasure against ROP is ASLR: modules are loaded at pseudo-random base addresses resulting in the whereabouts of gadgets being hard to predict. If not stated otherwise, we assume in the following discussions that the attacker has ways to gain knowledge on the base address of at least one executable module of sufficient size. We stress that this is no ambitious assumption as it is generally fulfilled for real-world ROP-based exploits [1, 7, 16].

2.2 Last Branch Recording

kBouncer and ROPecker rely on the *Last Branch Recording* (LBR) feature of contemporary AMD and Intel processors [3,15] to examine an application's past control flow on certain events.

The LBR can only be enabled and accessed from kernel mode. It can be configured to only track certain types of branches. Both kBouncer and ROPecker utilize this feature and they limit the LBR to indirect branches in user mode. For each recorded branch, an entry containing the *start* and *destination* address is written to the corresponding CPU core's *LBR stack*. In Intel's latest Haswell architecture, an LBR stack is limited to only 16 entries. For each newly recorded branch, the oldest entry in an LBR stack is overwritten. At any given time, an LBR stack may not only contain entries from a single process/thread, but from multiple ones running on the same core [8]. In the following, we do not consider this effect, though, it might in practice facilitate attacks. Instead, for simplicity, we assume that the LBR stack is always saved/restored on context switches.

3 Security Assessment of kBouncer

The latest version of the kBouncer runtime ROP exploit mitigation approach was presented by Pappas et al. in 2013 [25]. kBouncer checks for suspicious branch sequences hinting at a ROP exploit whenever a Windows API (WinAPI) [29] function considered as possibly harmful is invoked in a monitored process. kBouncer's authors list 52 WinAPI functions which they consider as possibly harmful. Among these functions are for example `VirtualAlloc()` and `VirtualProtect()` that are notoriously abused by attackers. Pappas et al. acknowledge that the list is possibly not complete and could be extended in the future.

kBouncer is composed of a user mode component and a kernel driver. The user mode component hooks all to-be-protected WinAPI functions in a monitored process. Whenever the control flow reaches one of these hooks, the kernel driver is informed via the WinAPI function `DeviceIoControl()`. Subsequently, the driver examines the LBR stack for traces of a ROP chain. Since kBouncer's user mode component uses two indirect branches to inform the driver, only 14 of the LBR stack's 16 entries are of value to the driver's ROP detection logic [25]. In case no attack is detected, the driver saves a corresponding "checkpoint" in kernel memory for the respective thread. Whenever a system call corresponding to a hooked WinAPI function is invoked, the driver consumes the matching checkpoint; if none is found, an attack is reported. According to Pappas et al., the purpose of the checkpoint system is to prevent exploit code from simply skipping over the top-level WinAPI functions and calling similar lower level functions (e. g., `NtCreateFile()` instead of `CreateFileW()`). The reason for kBouncer not monitoring system calls directly is the observation that between WinAPI functions' and their corresponding system call often many legitimate indirect branches are executed that would often overwrite traces of ROP chains in the LBR stack [25].

In order to evaluate kBouncer's practical applicability and defensive strength, we created a standalone emulator for kBouncer based on certain pieces of source code generously provided to us by Pappas et al. The emulator uses the *Pin* [18] dynamic analysis framework to instrument monitored applications at runtime. To the best of our judgment, the emulator accurately captures all of kBouncer's core concepts as described by Pappas et al. [25].

3.1 Examination of Indirect Branch Sequences

When examining the LBR stack corresponding to the invocation of a WinAPI function, kBouncer's kernel driver assumes an attack if at least one of the following is encountered: *(i)* a *return* to an instruction not preceded by a `call` instruction or *(ii)* a chain of a certain number of *gadgets* ending in the latest LBR stack entry. For kBouncer, gadgets are up to 20 instructions long and may contain conditional or unconditional relative jumps [25]. In the following, we refer to gadgets under this definition as k-gadgets. Gadgets outside this definition are conversely denoted as non-k-gadgets.

Listing 1.1. Simple recursive C function calculating the factorial of an integer

```
int factorial (int n)
{
 if (n <= 1) return 1;
 return factorial(n-1)*n;
}
```

Listing 1.2. Disassembly of epilogue of function `factorial()`

```
[...]
lea     ecx, [edi-1]
call    factorial
mul     edi
pop     edi
retn
```

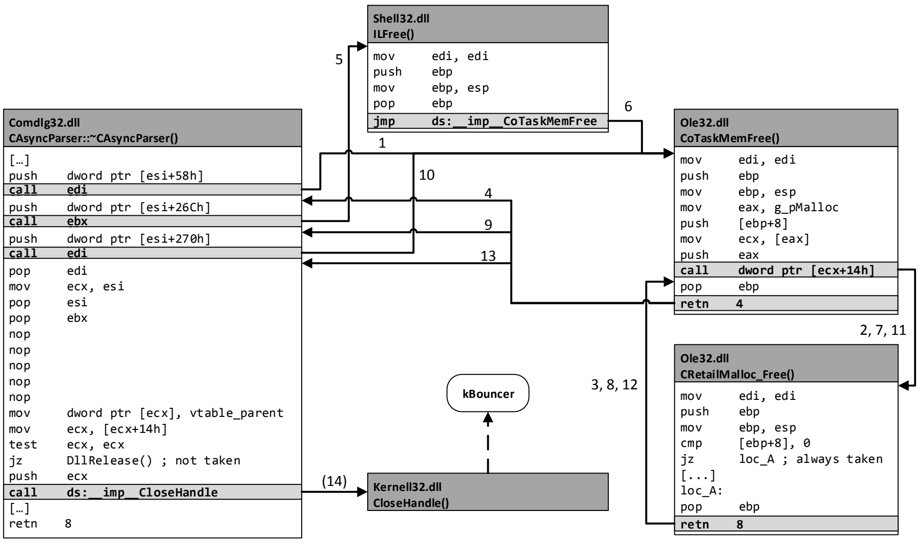
Gadget Chain Detection Threshold The maximum gadget chain length kBouncer can identify is 13. This is due to only 14 LBR stack entries being of value to kBouncer's detection logic and the latest effective entry always corresponding to a branch to a WinAPI function [25].

In order to determine a suitable detection threshold for the length of gadget chains, Pappas et al. examined a set of popular Windows applications (e. g., Microsoft Word and Internet Explorer) at runtime while executing certain tasks [25]. They report on having found the LBR stack to contain chains of at most *five* k-gadgets on entry to any of the 52 possibly harmful WinAPI functions across their experiments. As a result, Pappas et al. defined kBouncer to consider chains of *eight* or more k-gadgets as harmful, leaving a security margin of three against false positives.

However, longer chains of k-gadgets can easily occur in practice in benign and unsuspicious control flows. Consider for example a simple recursive function calculating the factorial of an integer as shown in Listing 1.1 and Listing 1.2. After the termination of `factorial(n)`, the LBR stack contains a legitimate chain of $n - 1$ k-gadgets of the form `mul edi; pop edi; retn;`, making the control flow appear to contain a ROP chain under the kBouncer definition. Many other possible scenarios exists where legitimate control flow resembles a ROP chain under the kBouncer definition as well.

In fact, our kBouncer emulator detected k-gadget chains longer than the given detection threshold for all non-trivial applications we executed on Windows 7 SP1 64-bit while monitoring the discussed 52 WinApi functions. For example, saving a text file using the popular editor Notepad++ 5.9.8 (32-bit) reliably resulted in one detected chain of the maximum length 13. The chain is depicted in Figure 1: the chain starts towards the end of the destructor of the class `CAsyncParser` in comdlg32.dll and spans over ole32.dll and shell32.dll before ending in the protected WinAPI function `CloseHandle()`. The characteristic of the chain is that several short functions are invoked in a nested manner using indirect calls only.

Note that the discrepancy in quality and quantity of false positives detected by our emulator and the original kBouncer could have many reasons. Possibly, the dynamic disassembly provided by Pin to our emulator is more comprehensive than the static disassembly available to kBouncer's *offline gadget extraction toolkit*. It is also very well possible that kBouncer employs certain additional filtering techniques in practice. Of course we can also not entirely rule out inaccurate assumptions on our side.



Comdlg32.dll
CAsyncParser::~CAsyncParser()

```
[…]
push    dword ptr [esi+58h]
call    edi
push    dword ptr [esi+26Ch]
call    ebx
push    dword ptr [esi+270h]
call    edi
pop     edi
mov     ecx, esi
pop     esi
pop     ebx
nop
nop
nop
nop
nop
mov     dword ptr [ecx], vtable_parent
mov     ecx, [ecx+14h]
test    ecx, ecx
jz      DllRelease() ; not taken
push    ecx
call    ds:__imp__CloseHandle
[…]
retn    8
```

Shell32.dll
ILFree()

```
mov     edi, edi
push    ebp
mov     ebp, esp
pop     ebp
jmp     ds:__imp__CoTaskMemFree
```

Ole32.dll
CoTaskMemFree()

```
mov     edi, edi
push    ebp
mov     ebp, esp
mov     eax, g_pMalloc
push    [ebp+8]
mov     ecx, [eax]
push    eax
call    dword ptr [ecx+14h]
pop     ebp
retn    4
```

Ole32.dll
CRetailMalloc_Free()

```
mov     edi, edi
push    ebp
mov     ebp, esp
cmp     [ebp+8], 0
jz      loc_A ; always taken
[...]
loc_A:
pop     ebp
retn    8
```

Kernell32.dll
CloseHandle()

Fig. 1. Exemplary false positive chain of 13 k-gadgets as detected by our kBouncer emulator for the "Save File As" dialogue in Notepad++ 5.9.8 (32-bit) on Windows 7 64-bit. Taken indirect branches are highlighted in light gray. Branches are labeled according to the order they are executed.

3.2 Circumventing kBouncer

We now explore ways an *aware attacker* can follow to circumvent kBouncer. We consider kBouncer as bypassed when it is possible (with respect to the actual limits imposed by a vulnerability) to reliably and repeatedly conduct the following two consecutive steps without kBouncer noticing:

S1 execution of arbitrary ROP chain
S2 successful invocation of a WinAPI function protected by kBouncer

Obviously kBouncer can be safely bypassed if the last 14 indirect branches leading to a protected WinAPI function cannot be distinguished from benign control flow; regardless of the actually deployed gadget chain detection policy. This is due to kBouncer's driver being effectively only able to look at most 14 LBR stack entries into the past.

In view of this fact, Pappas et al. discuss the possibility of an attack based on a seemingly legitimate gadget chain (returns leading to call-preceded locations only and at least every eighth gadget being a non-k-gadget). They allude that such an attack would be difficult and state that "if evasion becomes an issue, longer gadgets could be considered during the gadget chaining analysis of an LBR snapshot" [25]. Furthermore, they also discuss the possibility of an attacker looking "[...] for a long-enough execution path that leads to the desired API call as part of the applications logic". They expect this kind of attack to be "[...] 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".

We find that an attacker could instead also employ a simpler *third* method: the code executed between a ROP chain (step S1) and a protected WinAPI function (step S2) does not necessarily need to be *meaningful*; not in the context of the ROP chain and neither in the context of the attacked application. Hence, an attacker can simply execute arbitrary *meaningless* code between both steps in order to flush the LBR stack prior to the inspection through kBouncer's driver. The only requirements such *LBR-flushing code* has to fulfill are:

- Sufficiently many (e. g., 14) unsuspicious indirect branches must be executed.
- The arguments to the to-be-invoked WinAPI function must not be altered.
- Other WinAPI functions protected by kBouncer must not be invoked.
- The execution environment must not be rendered uncontrollable; e. g., by access violation exceptions or manipulation of the ROP chain on the stack.

In the following we *(i)* discuss suitable LBR-flushing code sequences and *(ii)* explain how attackers can generically circumvent kBouncer by incorporating them into ROP chains. Attacks for 32-bit and 64-bit environments are discussed separately as they require slightly different approaches due to divergent default calling conventions: in 32-bit applications, arguments to WinAPI functions are passed over the stack (`stdcall` calling convention), whereas the first four arguments are passed in registers in 64-bit applications (`fastcall` calling convention) [20].

We limit ourselves to gadgets/code sequences that are likely to be present in almost every process on Windows. In fact, all required gadgets/code sequences can be found in standard Windows libraries and, at least for 32-bit, in every C/C++ program created with default/common compiler and linker settings (at least *Release* or *Debug* configuration; `/Od`, `/O1`, or `/O2` optimization) using Microsoft Visual Studio versions 2010, 2012, or 2013. This is even valid for the minimal C/C++ program with an empty `main()` whose `.text` section typically has an effective size of under 1 KB. We refer to this executable (*Release*, `/O2`) as minpe-32 and minpe-64, respectively. All code that is present in minpe-32/minpe-64 should also be present in virtually every other program compiled and linked with default settings using Visual Studio.

3.3 Circumvention for 32-Bit Applications

LBR-Flushing Code Sequences For 32-bit programs, finding suitable LBR-flushing code sequences is easy: basically most functions that make a certain amount of sub-calls (each sub-call terminates in an indirect branch) and do not much depend on or interfere with the global state of a program comply with the listed requirements. In the following, we refer to a function with these properties as *LBR-flushing function* (lbr-ff). We found for example `lstrcmpiW()`[1] in ker-

[1] `lstrcmpiW()` compares two Unicode strings in a case-insensitive manner.

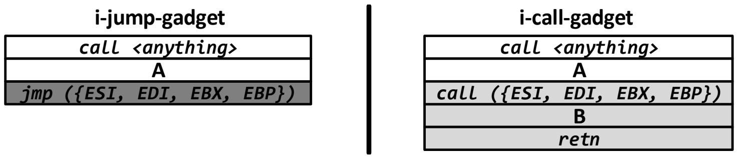


Fig. 2. Formats of the 32-bit invocation gadget types i-jump-gadget (left) and i-call-gadget (right); blocks labeled **A** and **B** may be empty or contain any sequence of instructions not rendering the execution context uncontrollable.

nel32.dll to be such a function. When supplied with two identical pointers to (almost) arbitrary data as arguments, we found that it reliably executed more than 20 unsuspicious indirect branches. The fact that the function expects two arguments is of course disadvantageous for an attacker, as this wastes precious space on the (fake) stack. In practice, an attacker could ideally choose an lbr-ff without arguments. E. g., we identified the two standard runtime library functions `pre_c_init()` (statically contained in minpe-32) and `EtwInitializeProcess()` (contained in ntdll.dll) as lbr-ffs with zero arguments. It should be clear that suitable lbr-ffs are available in abundance in most real-world applications.

Invocation Gadgets Given an lbr-ff, the attacker's goal is to execute it between the ROP chain (step S1) and the invocation of a protected WinAPI function (step S2) in order to flush the LBR stack just before kBouncer's detection logic is triggered. Executing the lbr-ff itself is trivial: it can be part of the ROP chain just like any other gadget. Obviously though, the lbr-ff cannot simply "return" in ROP-manner to the entry point of a protected WinAPI function; kBouncer would certainly detect an attack, as entry points of WinAPI functions are never preceded by a `call` in the static instruction stream.

Instead, the control flow needs to transition from the lbr-ff to the protected WinAPI function in such a way that kBouncer cannot distinguish it from legitimate control flow. We found that for an attacker to achieve this, the availability of a call-preceded and controllable jump-based or call-based *invocation gadget* as depicted in Figure 2 is sufficient. In the following, we refer to gadgets of these formats as i-jump-gadgets and i-call-gadgets, respectively.

Given an i-jump-gadget or an i-call-gadget, a protected WinAPI function can be invoked right after an lbr-ff in such a way that the control flow appears legitimate to kBouncer. Figure 3 schematically depicts the control flows for both types of gadgets: ⓪ From the ROP chain, the control flow is transferred to the lbr-ff of choice via a traditional `retn` terminated gadget. We need to make sure that at this point the address of the instruction sequence **A** (see Figure 2) lies on top of the stack. ① This makes the lbr-ff return to **A** right behind the leading dummy `call` instruction of the i-jump-gadget/i-call-gadget. ② The protected WinAPI function is then invoked via the indirect `jmp/call` instruction following **A**. Typically, this instruction should branch relative to the registers `esi`, `edi`, `ebx`, or `ebp` (e. g., `jmp [ebx*4+edi]` or `call esi`). These registers are

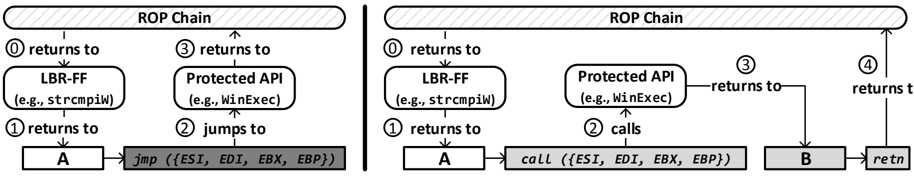


Fig. 3. Schematic control flow of the invocation of a protected WinAPI (32-bit); **left:** i-jump-gadget **right:** i-call-gadget

premiere choices here, because they are defined to be callee-saved in all common C/C++ calling conventions for x86 [20]. Hence, these registers can be assumed to be unaltered by the invocation of virtually any lbr-ff. This allows the attacker to set the registers using regular gadgets (*before* step ⓪). ③, ④ Depending on the invocation gadget type, the WinAPI function either returns directly to the ROP chain (i-jump-gadget) or a detour is taken over the instruction sequence **B** (i-call-gadget).

kBouncer's detection logic is triggered between steps ② and ③. At this point kBouncer cannot detect an attack anymore, as the LBR stack exclusively contains entries corresponding to branches executed *after* step ⓪. Note that the instruction sequence **A** is call-preceded. Hence, the return from the (legitimate) lbr-ff to **A** is unsuspicious to kBouncer.

Passing of Arguments. Typically, the attacker would align arguments to the WinAPI function on the stack prior to executing the lbr-ff (before step ⓪). Depending on the nature of an invocation gadget though, arguments might also be written to the stack by the instruction sequence **A**. Of course it is a requirement that the instruction sequence **A** does not alter the stack or register values in such a way that the WinAPI function cannot be invoked as intended or the control flow cannot properly resume afterward. For example, the i-jump-gadget `call <anything>; push 0; jmp edi;` would allow to invoke a WinAPI function but would inevitably lead to the function returning to the invalid address `0`. Also, instructions triggering exceptions/interrupts must of course not be present in **A**. Naturally, similar requirements apply to the trailing instruction sequence **B** of the i-call-gadget.

Gadget Examples. An example for a suitable i-jump-gadget is given in Listing 1.3. The gadget's **A** sequence (lines 2–6) is composed of `xor` operations on general purpose registers. This should be unproblematic for the attacker in almost all cases.

We implemented a Python script to statically identify this and multiple other suitable i-jump-gadgets and i-call-gadgets in common Windows DLLs in an automated manner. We found this particular i-jump-gadget to be present in the 32-bit versions of kernel32.dll, kernelbase.dll, ntdll.dll, user32.dll, msvcr100.dll, msvcr110.dll, msvcr120.dll, and msvcrt.dll of both Windows 7 and Windows 8.

Listing 1.3. Aligned i-jump-gadget in `TransferToHandler()` found in multiple Windows DLLs

```
call    sub_7DD9D8F5
xor     eax, eax
xor     ebx, ebx
xor     ecx, ecx
xor     edx, edx
xor     edi, edi
jmp     esi
```

Listing 1.4. Aligned i-call-gadget in `_onexit()` of the standard Visual C/C++ library

```
 1  call    esi
 2  mov     __onexitbegin, eax
 3  push    dword ptr [ebp-20h]
 4  call    esi
 5  mov     __onexitend, eax
 6  mov     dword ptr [ebp-4], 0FFFFFFFEh
 7  call    $+10h
 8  mov     eax, edi
 9  call    _SEH_epilog4
10  retn
```

All these DLLs are without doubt among the most frequently used ones on Windows. In fact, ntdll.dll can be found in every Windows user mode process [29]. An example for an i-call-gadget is given in Listing 1.4. We discovered this gadget in the static runtime library function `_onexit()` [21] contained in minpe-32 (and other executables). While also allowing to generically bypass kBouncer, we found the gadget to be slightly more complicated to handle than the i-jump-gadget in Listing 1.3. Reasons are the presence of the `push` instruction in the gadget's **A** sequence (lines 2–3) and the presence of the two static calls in the **B** sequence (lines 5–9).

Obviously, one of these two gadgets should be available to the attacker in most scenarios. If not, it should in the uttermost cases be simple to find comparable gadgets given that the i-call-gadget was found in less than 1 KB of code. Knowledge of these two gadgets proved to be sufficient when we adapted high-profile real world exploits to be undetectable by kBouncer (see § 3.5).

3.4 Circumvention for 64-Bit Applications

The described 32-bit approach for bypassing kBouncer is only to some extent applicable to 64-bit. In the default 64-bit calling convention on Windows, the first four arguments to a function are not passed over the stack but in the registers `rcx`, `rdx`, `r8`, and `r9` [20]. Accordingly, an attacker would in most cases need to preload these registers *before* the invocation of the lbr-ff if the 32-bit approach was followed here. As these four registers are explicitly not callee-saved, they are likely to be altered by almost all lbr-ff. Hence, a different approach is needed for 64-bit systems.

Loop Invocation Gadget We found a certain type of 64-bit gadget to be especially suited for both the flushing of the LBR stack and the invocation of protected WinAPI functions. A specimen contained in minpe-64 is given in Listing 1.5. The gadget is comparable to the *dispatcher gadget* that was discussed as foundation for *jump-oriented programming* by Bletsch et al. [4]. The gadget interprets `rbx` as an index into a table of code pointers. `rbx` is gradually increased and all pointers are called until `rbx` equals `rdi`. The gadget allows

Listing 1.5. Aligned i-loop-gadget in `RTC_Initialize()` of the standard Visual C/C++ library

```
@loop:
mov     rax, [rbx]
test    rax, rax
jz      @skip
call    rax
@skip:
add     rbx, 8
cmp     rbx, rdi
jb      @loop
mov     rbx, [rsp+28h+arg_0]
add     rsp, 20h
pop     rdi
retn
```

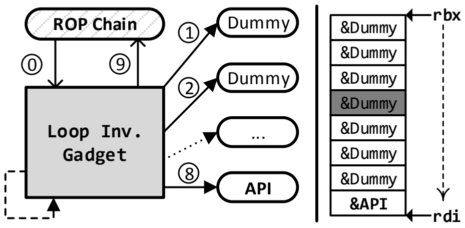


Fig. 4. Schematic control of the invocation of a WinApi function (i-loop-gadget)

an attacker to execute an arbitrary number of gadgets/functions in a manner that replicates benign control flow. Of course invoked gadgets must generally not alter `rbx` or `rdi`. A very similar loop invocation gadget is for example also contained in `LdrpCallTlsInitializers()` in the 64-bit ntdll.dll. We refer to this type of gadget as i-loop-gadget. An i-loop-gadget can be used to flush the LBR stack and to invoke a protected API subsequently as depicted in Figure 4: if a return-succeeded *dummy* gadget is executed at least seven times before the invocation of a protected API, the LBR stack does not contain any traces of the actual ROP chain when kBouncer's detection logic is triggered (for each dummy gadget an indirect call/return pair is executed). However, finding a suitable dummy gadget is not as easy as it might seem. Obviously, the dummy gadget must be a non-k-gadget as the i-loop-gadget in Listing 1.5 already is a k-gadget. If both are k-gadgets, then an attack is detected by kBouncer. Furthermore, the dummy gadget must neither alter the registers `rbx` and `rdi` nor the registers `rcx`, `rdx`, `r8`, and `r9` carrying the arguments for the WinAPI function. Also, the dummy gadget of course must not render the program state uncontrollable to the attacker. We implemented a Python script to identify appropriate dummy gadgets in standard 64-bit Windows DLLs. We found a variety of long and aligned math related gadgets/functions in ntdll.dll and msvcr*.dll accessing (almost) exclusively the specialized SSE [15] floating-point registers `xmm0` to `xmm7`. For example, `_remainder_piby2_cw_forAsm()` in msvcr120.dll contains a gadget that does not write to memory and only touches SSE registers and `rax` while executing at least 26 instructions. We also found several long sequences (20+) of `nop` instructions terminated by a return in ntdll.dll. Unfortunately, we did not find a suitable dummy gadget in the `.text` section of minpe-64.

In practice, the attacker might very well interleave dummy gadgets with meaningful k-gadgets, which do not alter `rbx` or `rdi`, in the invocation loop. In fact, as kBouncer per default only considers chains of more than seven k-gadgets harmful, it would be sufficient to execute a single dummy gadget at the fourth position (marked dark gray in Figure 4). This would enable the attacker to use the last three gadgets before the invocation of the WinAPI function to conveniently write arguments to the registers `rcx`, `rdx`, `r8`, or `r9`. This would result in less constraints regarding register usage for the employed dummy gadget. Generically bypassing kBouncer using an i-loop-gadget is also possible for 32-bit applications. We found for example the 32-bit equivalent of the i-loop-gadget in Listing 1.5 to be also present in minpe-32. Using the i-jump-gadgets or i-call-gadgets discussed in § 3.3 should though in most cases incur less overhead in 32-bit environments. Also, we found suitable dummy gadgets to be relatively sparse compared to lbr-ffs.

3.5 Example Exploits

To demonstrate the practicality of the described kBouncer bypasses and to assess the resulting overhead, we developed a set of example exploits which we briefly discuss now. As it is tradition, our exploits launch the Windows calculator via an invocation of `WinExec()`. We stress that in all cases much more complicated exploits with multiple WinAPI calls would have been easily possible. No standard Windows defensive mechanisms like ASLR and DEP were disabled or manipulated. We confirmed that our exploits would indeed circumvent kBouncer using our emulator where possible. Due to technical constraints we resorted to manual confirmation using a debugger for Internet Explorer and Firefox.

Minimal Vulnerable Programs. We extended the discussed minimal executables minpe-32 and minpe-64 to contain a simple buffer overflow vulnerability. We assumed that the attacker knew the base addresses of the main module and msvcr120.dll. In both cases we used common gadgets from msvcr120.dll like `pop eax; ret;` to construct a conventional ROP chain. We then used the discussed i-call-gadget and the lbr-ff in minpe-32 to invoke `WinExec()`; respectively for the 64-bit variant we leveraged the i-loop-gadget in minpe-64 and the discussed dummy gadget in msvcr120.dll. For 32-bit ten extra dwords (32-bit words) were needed in the ROP payload to bypass kBouncer (25 dwords vs. 35 dwords); for 64-bit 20 additional qwords (64-bit words) were required (29 qwords vs. 49 qwords). The relatively large overhead for 64-bit stems from the inclusion of the eight qword long code pointer table.

Details on the basic and augmented ROP chains for 32-bit and 64-bit can be found in our technical report corresponding to this paper [30].

MPlayer Lite. Pappas et al. used a stack buffer overflow vulnerability in *MPlayer Lite* version r33064 for Windows [1] to evaluate the effectiveness of kBouncer. MPlayer Lite is compiled with MinGW's GCC version 4.5.1. We used gadgets from the bundled avcodec-52.dll to build a conventional ROP-based exploit for the same vulnerability. To circumvent kBouncer, we augmented the ROP chain by an i-loop-gadget located in the static runtime library function `TlsCallback_0()` in mplayer.exe. As corresponding *dummy gadget* we chose another one of MinGW's static runtime library function. Altogether, 37 additional dwords were needed for the augmented ROP chain (21 dwords vs. 58 dwords). We found similar gadgets also in binaries compiled with different MinGW GCC versions.

Internet Explorer 10. We modified a publicly available exploit for an integer signedness error in Internet Explorer 10 32-bit for Windows 8 by *VUPEN Security* [16]. The original exploit was a winning entry at the popular 2013 Pwn2Own contest. It uses JavaScript code to dynamically construct a ROP chain consisting of 10 dwords to invoke `WinExec()`. In our modified version, four extra dwords are used to incorporate the i-jump-gadget in Listing 1.3 (kernel32.dll) and `lstrcmpiW()` as lbr-ff.

TorBrowser Bundle / Firefox 17. We modified the exploit allegedly used by the FBI to target users of the *TorBrowser Bundle* [7]. The TorBrowser Bundle is based on Firefox version 17.0.6 for Windows 7 32-bit. We use a ROP payload of 54 dwords to invoke `WinExec()`. The version bypassing kBouncer includes five additional dwords and uses the i-jump-gadget in Listing 1.3 (ntdll.dll) and `EtwInitializeProcess()` (ntdll.dll) as lbr-ff.

3.6 Possible Improvements

We now briefly review three potential improvements to address our bypasses and discuss their effectiveness.

Broadening of Gadget Definition. Pappas et al. propose that kBouncer could be improved by considering gadgets longer than 20 instructions if evasion became an issue [25]. We note that such an extension could not substantially tackle the described 32-bit attacks using i-jump-gadgets or i-call-gadgets in conjunction with lbr-ffs (see § 3.3): when kBouncer's detection logic is triggered, the effective LBR stack contains one entry corresponding to the invocation gadget and 13 to the lbr-ff. The lbr-ff's LBR entries cannot reasonably be distinguished from benign control flow, as the lbr-ff is a legit function of the attacked application (e. g., `lstrcmpiW()`). A broader definition of k-gadgets could make it harder to find dummy gadgets suitable for the (64-bit) attack approach based on i-loop-gadgets (see § 3.4). In practice though, increasing the maximum gadget length such that most suitable dummy gadgets are eliminated, would probably result in unacceptable high numbers of overall false positives. Even for a maximum length of 20, entire non-trivial functions fall already under the k-gadget definition.

Larger LBR Stack. Pappas et al. suggest that future CPU generations with larger LBR stacks "would allow kBouncer to achieve even higher accuracy by inspecting longer execution paths [...]" [25]. In such a case, our described approaches could easily be adapted to create longer sequences of indirect branches resembling benign ones. For example, the described i-loop-gadget can be used to create such sequences of almost arbitrary length. Also, finding lbr-ffs which do so is easy. The discussed `lstrcmpiW()` can for example be used to create dozens of legit indirect branches.

Heuristic Detection of Invocation Gadgets. One could attempt to extend kBouncer to heuristically check for LBR entries corresponding to the discussed types of invocation gadgets. This could, depending on the actual implementation, very well fend off the described attacks. However, we expect high numbers of false positives from such a measure, as the same invocation patterns can very well occur for benign control flows.

4 Security Assessment of ROPGuard

ROPGuard is a runtime ROP detection approach for user mode applications on Windows [10]. It placed 2$^{\text{nd}}$ to kBouncer at the BlueHat Prize and is incorporated into the *Enhanced Mitigation Experience Toolkit* (EMET) [22] that is provided as optional security enhancement for Windows. Hence, ROPGuard can be considered as the most widely spread advanced ROP countermeasure for Windows applications.

Similar to kBouncer, ROPGuard hooks a set of critical WinAPI functions in user mode processes. Whenever such a hook is triggered, ROPGuard as implemented in EMET 4.1—the most recent version at the time of this writing—tries to detect ROP-based exploits via a variety of checks. We describe the two most relevant ones now briefly [10, 19]:

- *Past and Future Control Flow Analysis*: ROPGuard verifies that the return address of a protected WinAPI function is call-preceded. Furthermore, it simulates the control flow in a simple manner from the return address onwards and checks for future non call-preceded returns. Simulation is performed until a certain threshold number of future instructions was examined or any call or jump instruction is encountered.
- *Stack Checks*: ROPGuard checks if the stack pointer points within the expected memory range for the given thread. It is common practice for attackers to divert the stack pointer to a memory region (e. g., the heap) under their control. ROPGuard also blocks attempts to make the stack executable.

Reports on how to bypass ROPGuard's implementation in EMET have already been published on the Internet (e. g., [26]). In fact, ROPGuard's original author Ivan Fratric suggests that an attacker who is aware of it "would be able to construct special ROP chains that would [...] push ROPGuard off guard" [11]. We found that our kBouncer example exploits that rely either on i-call-gadgets or on i-loop-gadgets (both *minimal vulnerable programs* and *MPlayer*) already bypassed ROPGuard's implementation in EMET. In turn, ROPGuard successfully stopped all of the three corresponding unmodified exploits. For ROPGuard, the discussed i-call-gadgets and i-loop-gadgets invoke the protected `WinExec()` via seemingly legitimate calls. These gadgets also make ROPGuard's future control flow simulation stop early due to subsequent jumps/calls. The stack-related checks do not apply to our example exploits.

5 Security Assessment of ROPecker

ROPecker, a runtime ROP exploit mitigation system, was presented by Cheng et al. in 2014 [8]. ROPecker aperiodically checks for abnormal branch sequences in an application's control flow. For that, ROPecker combines kBouncer-like examination of the LBR stack with ROPGuard-like future control flow simulation. Cheng et al. specifically report on a prototype implementation of ROPecker as a kernel module for 32-bit x86 Linux systems. Hence, we also only consider this platform. For evaluation purposes, we implemented an experimental standalone Pin-based emulator for ROPecker. We are confident that this emulator accurately captures most of ROPecker's aspects. All experiments we report on in the following were conducted on either Ubuntu 12.0.4 or Debian 7.4.0 systems.

5.1 Triggering of Detection Logic

Other than comparable approaches, ROPecker does not apply any form of binary rewriting such as API function hooking to inspect an application's control flow. Instead, ROPecker ensures that only a small fixed-size dynamic set of code pages is executable at any given time within a process. ROPecker's ROP detection logic is invoked every time an access violation is triggered due to the target application's control flow reaching a new page *outside* the set of executable pages. If no attack is detected, ROPecker replaces the oldest page in the set of executable pages with the newly reached page and resumes the execution of the corresponding thread/process. Cheng et al. refer to this technique as a "sliding window mechanism". They suggest using a window/set size of two to four executable pages, corresponding to 8 to 16 KB of executable code, because it is supposedly hard to find enough gadgets for a meaningful attack in less than 20 KB of code [8]. The pages inside the sliding window do not necessarily need to be adjacent.

For our emulator, we use a fixed sliding window size of exactly one page to achieve fine-granular capturing. Note that a smaller sliding window size results in ROPecker's detection logic being triggered more often. Hence, chances for false negatives decrease while in turn chances for false positives increase.

5.2 Examination of Indirect Branch Sequences

Each time it is triggered, ROPecker's detection logic tries to identify attacks by analyzing the past and the (simulated) future control flow of a thread/process for chains of ROP gadgets. Per default, ROPecker considers a sequence of instructions to be a gadget in case it meets the following criteria [8]: *(i)* the last instruction is an indirect branch; *(ii)* no other branch (e. g., `call` or `jnz`) is contained; *(iii)* it consists of at most six instructions. This limit was arbitrarily chosen by Cheng et al. ROPecker can be configured to consider longer gadgets. We refer to gadgets that comply with ROPecker's definition as r-gadgets.

Table 1. Exemplary max_{nor} as determined by our ROPecker emulator.

Application	max_{nor}	Activity
Nginx 1.4.0	5	delivery of small web page
Adobe Reader 9.5.5	9	opening of document
Pidgin 2.10.9	9	IRC chat
Gimp 2.8.2	9	simple drawing
VLC 2.0.8	11	playback of short OGG video
LibreOffice Calc 3.5.7.2	17	creation of simple spreadsheet

Analysis of Past and Future Indirect Branches Like kBouncer, ROPecker configures the CPU's LBR facility to only track indirect branches in user mode. Whenever execution reaches a page outside the sliding window, ROPecker first examines the thread's/process' *past* indirect branches for a chain of r-gadgets via the LBR stack: going backward from the most recent one, it is checked for each LBR entry (which necessarily ends in an indirect branch) if its branch destination is an r-gadget. The *past* detection stops with the first entry not matching this characteristic. After that, ROPecker simulates the thread's/process' *future* indirect branches using rather complex emulation techniques going forward from the most recent LBR entry's branch destination. As soon as a code sequence is encountered that does not qualify as r-gadget, the *future* detection stops. If the accumulated length of the *past* and the *future* gadget chains is above a certain threshold, an attack is assumed.

Gadget Chain Detection Threshold Cheng et al. suggest using a chain detection threshold between 11 and 16 r-gadgets where "an ideal threshold should be smaller than the minimum length min_{rop} of all ROP gadget chains, and at the same time, be larger than the maximum length max_{nor} of the gadget chains identified from normal execution flows" [8]. They report that various real world and artificial ROP chains analyzed by them consisted of 17 to 30 gadgets. Hence, they universally assume $min_{rop} = 17$. To assess max_{nor}, Cheng et al. examined a variety of applications (certain Linux coreutils, SPEC INT2006, ffmpeg, graphics-magick, and Apache web server) during runtime. For the code paths triggered in their experiments, they found max_{nor} overall to be 10 and for Apache even only 4; values well below their empirically determined $min_{rop} = 17$.

In practice, higher values for max_{nor} are not totally unlikely though. Consider for example again the simple recursive function `factorial()` from Listing 1.1 in § 3 whose epilogue qualifies as r-gadget. We used our experimental emulator to explore the range of max_{nor} for popular applications not covered by experiments conducted by Cheng et al. The results are listed in Table 1. The encountered chain of 17 r-gadgets for LibreOffice Calc resulted from a long chain of returns from nested function calls (similar to the `factorial()` example). We emphasize that our emulator with a sliding window size of only one page naturally catches more false positives and produces higher max_{nor} than configurations with larger sliding windows. However, these numbers suggest that ROPecker might not be equally well applicable to all kinds of applications, as in certain cases max_{nor} could be too high to allow for a reasonably low detection threshold min_{rop}.

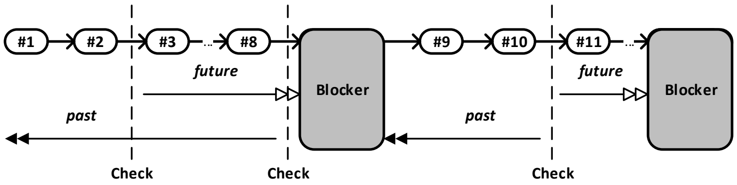


Fig. 5. Generic layout of a gadget chain bypassing ROPecker. Conventional gadgets (white) are interleaved with gadgets stopping the *past* and *future* detection logic (gray).

5.3 Circumvention

We now discuss methods for the generic circumvention of ROPecker. In general, we find that the narrow definition of r-gadgets makes ROPecker only a small hurdle for aware attackers.

Cheng et al. state that ROPecker's "[...] payload detection algorithm is designed based on the assumption that a gadget does not contain direct branch instructions, which is also used in the many previous work [...]. Therefore, the gadget chain detection stops when a direct branch instruction is encountered" [8]. They also acknowledge that an "[...] adversary may carefully insert long gadgets into consecutive short gadgets to make the length of each segmented gadget chain not exceed the gadget chain threshold [...]" to achieve the same. Note that these statements already describe all that is necessary in order to successfully bypass ROPecker in a generic manner. As depicted in Figure 5, attackers simply need to take care to periodically mix in a non-r-gadget (containing a branch or more than six instructions) into their gadget chains in order to stop ROPecker's *past* and *future* detection logic before the given detection threshold is reached. In the following, we refer to such a gadget as blocker-gadget.

Cheng et al. argue that to the best of their knowledge an attack using jump-containing gadgets "[...] has not been found in real-life.". We note that this observation does not necessarily imply that jump-containing (or long) gadgets are hard to use. Instead, it is in the uttermost cases trivial for an attacker to find and use such gadgets, as they do not need to be meaningful in any context. The only requirement is that they do not render the program state uncontrollable as already discussed in § 3 for kBouncer. Even entire regular functions as the ones discussed in § 3.3 can be misused by attackers here. In our example exploit against ROPecker (see § 5.4) we use for example the standard POSIX function `usleep()` as blocker-gadget.

5.4 Example Exploit

To demonstrate the applicability of the discussed ROPecker bypassing strategy, we created a ROP-based exploit for a stack buffer overflow vulnerability (CVE-2013-2028) [27] in the popular web server Nginx version 1.4.0. We inserted the function `usleep()` as blocker-gadget into the ROP chain after at least every seventh regular gadget. The entire resulting ROP payload is 107 dwords long—92 dwords are needed without ROPecker evasion—and creates a file on the target system using the `system()` function. Our ROPecker emulator detects a maximum chain length of nine for the exploit due to the epilogue of `usleep()` containing two chained r-gadgets. As this is below the default detection threshold of 11, the attack goes unnoticed.

5.5 Possible Improvements

We again briefly review potential improvements to address our bypasses and discuss their effectiveness.

Detection of Unaligned Gadgets. Cheng et al. propose that ROPecker could be improved by considering the execution of unaligned instructions as attack [8]. They note though, that it may not always be possible to decide if a given x86 instruction sequence is aligned or not. Attackers restricted to aligned gadgets would probably need longer gadget chains on average to achieve compromise. Also, finding suitable gadgets in general would be more complicated. The generic circumvention approach described in § 5.3 could though not be prevented.

Accumulation of Chain Lengths. To tackle attacks relying on blocker-gadgets, Cheng et al. suggest an extension to ROPecker that accumulates the detected chain lengths for multiple (e. g., three) consecutive sliding window updates. However, we find that an attacker could still generically avoid detection by using a (meaningless) function as blocker-gadget which updates the sliding window several times. When such a function returns to the next r-gadget, the accumulated chain length should in the uttermost cases be well below the detection threshold. We found for example the already mentioned `usleep()` to be a suitable function for this purpose. In our experiments, the function reliably switched pages several times before finally executing a system call.

Broadening of Gadget Definition. Lastly, Cheng et al. propose extending ROPecker in such a way that instruction sequences connected by direct jumps are also considered as gadgets, but also state that this might increase the number of false positives. In order to evaluate the practicality of such an extension, we experimentally modified our ROPecker emulator to consider kBouncer's k-gadgets (up to 20 instructions including direct jumps) instead of r-gadgets. With this hypothetical extension in place, we generally encountered high numbers of false positives often corresponding to astonishingly long benign chains of k-gadgets. For example, our emulator detected a chain of length 14 in libc for a small *hello world* application. While monitoring VLC during the playback of a short OGG video, the emulator even detected chains of lengths 77 and 82 in librsvg2 and libexpat respectively; the first being induced by a long static sequence of indirect calls to a very short function and the latter by a compact looped switch-case statement implemented using a central indirect jump. This hints at ROPecker possibly not being reasonably extendable to consider significantly more complex gadgets.

Checking for Illegal Returns. We believe that ROPecker's defensive strength could indeed be increased if it would consider returns to non call-preceded locations as indicator for an attack like kBouncer and ROPGuard do. Such an extension would effectively require attackers to largely resort to call-preceded gadgets or JOP-like concepts such as i-loop-gadgets (see § 3.4). While this would not prevent bypasses, it could significantly raise the bar. We would expect negligible overhead and close to zero additional false positives from such an extension as to the best of our knowledge returns to not call-preceded locations virtually never occur in benign control flows.

6 Related Work

To the best of our knowledge, the discussed ROP mitigation techniques have not been reviewed in other academic publications so far. Recently and concurrently to our work, Göktaş et al. demonstrate ways to bypass certain *control-flow-integrity* (CFI) systems for binary applications [12]. They show how certain types of gadgets still allow for ROP-like attacks in the presence of these systems. They mention that two of these gadget types could potentially be used to "call a function simply for tricking kBouncer" and refer to future work. We note that our described exploits would be prevented by these CFI systems. Our approaches could though be combined with the one presented by Göktaş et al.

Stephen Checkoway discusses in an article on the Internet, among others, kBouncer's first version [24] that "does not protect against return-oriented programming that doesn't use returns" [5]. This variant of kBouncer was meant to be invoked on the invocation of system calls instead of top-level WinAPI functions. Checkoway states that long enough regular code paths leading to system calls in an application could be used to erase traces of a ROP chain before kBouncer's detection logic becomes active.

The insights of both Göktaş et al. and Checkoway are similar to the foundation of our described attack techniques.

7 Conclusions

We examined the practical effectiveness of three recent approaches that attempt to prevent return-oriented programming. These are *kBouncer* [25], *ROPecker* [8], and *ROPGuard* [10]. All of them can reliably detect and prevent legacy exploits. We showed in turn that they can be bypassed in generic ways with little effort by *aware* adversaries. The basic problem is that the three approaches only analyze a limited number of recent (and upcoming) branches and an adversary can fool the employed heuristics. Both kBouncer and ROPecker rely on a custom kernel driver and employ complicated detection techniques build upon the LBR feature of modern processors. They though fall short to supply significantly higher protection levels than the much simpler ROPGuard. Our experimental results also hint at kBouncer and ROPecker being more prone to false positive attack detections than ROPGuard. We conclude that LBR, a feature that was originally designed for profiling and debugging purposes, is probably not particularly well suited for the implementation of strong defensive measures with reasonable runtime overhead.

Acknowledgments This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16BP12302; EUREKA-Project SASER.

References

1. Mplayer (r33064 lite) buffer overflow + ROP exploit. `http://www.exploit-db.com/exploits/17124/`, 2011.
2. Microsoft BlueHat Prize. `http://www.microsoft.com/security/bluehatprize/`, 2012.
3. Advanced Micro Devices. *AMD64 Architecture Programmers Manual Volume 2: System Programming.* December 2013. Publication no. 24593 Rev. 3.24.
4. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 30–40, New York, NY, USA, 2011. ACM.
5. S. Checkoway. Return-oriented programming's status is unchanged. `https://www.cs.jhu.edu/~s/musings/rop.html`, October 2013. Blog.
6. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 559–572, New York, NY, USA, 2010. ACM.
7. W. Chen. Here's that FBI Firefox exploit for you (CVE-2013-1690). `https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690`, August 2013.
8. Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.
9. T. Dullien, T. Kornau, and R.-P. Weinmann. A framework for automated architecture-independent gadget search. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2010.
10. I. Fratric. Runtime Prevention of Return-Oriented Programming Attacks. `http://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf`.
11. I. Fratric. My BlueHat prize entry: ROPGuard – runtime prevention of return-oriented programming attacks. `http://ifsec.blogspot.de/2012/08/my-bluehat-prize-entry-ropguard-runtime.html`, August 2012. Blog.
12. E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.
13. A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.
14. R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, 2009.

15. Intel. *Intel 64 and IA-32 architectures software developers manual*, volume 1, 2A, 2B, 2C, 3A, 3B and 3C. September 2013. 325462-048US.
16. N. Joly. Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013). http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php, 2013.
17. S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/~krahmer/no-nx.pdf, 2005.
18. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *SIGPLAN Not.*, volume 40.6, pages 190–200, New York, NY, USA, June 2005. ACM.
19. Microsoft Corporation. Enhanced mitigation experience toolkit 4.1—user guide, 2013.
20. Microsoft Developer Network. Argument passing and naming conventions. http://msdn.microsoft.com/en-us/library/984x0h58.aspx.
21. Microsoft Developer Network. C run-time library reference: _onexit. http://msdn.microsoft.com/en-us/library/zk17ww08.aspx, 2012.
22. Microsoft Security Research & Defense. Introducing enhanced mitigation experience toolkit (EMET) 4.1. http://www.microsoft.com/security/bluehatprize/, November 2013.
23. Nergal. The advanced return-into-lib(c) exploits: PaX case study. http://phrack.org/issues/58/4.html, 2001.
24. V. Pappas. kBouncer: Efficient and transparent ROP mitigation. http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf.
25. V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.
26. A. Portnoy. Bypassing all of the things. https://www.exodusintel.com/files/Aaron_Portnoy-Bypassing_All_Of_The_Things.pdf, 2013.
27. Rapid7 Vulnerability & Exploit Database. Nginx HTTP server 1.3.9–1.4.0 chunked encoding stack buffer overflow. http://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size, 2013.
28. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, march 2012.
29. M. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1.* Microsoft Press, 6th edition, 2012.
30. F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. Technical Report TR-HGI-2014-001, Ruhr-Universität Bochum, May 2014. http://syssec.rub.de/research/publications/Evaluating-Anti-ROP-Defenses/.
31. E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.
32. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.