Dynamic Reconstruction of Relocation Information for Stripped Binaries

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis

Columbia University
{vpappas,mikepo,angelos}@cs.columbia.edu

Abstract. Address Space Layout Randomization (ASLR) is a widely used technique for the prevention of code reuse attacks. The basic concept of ASLR is to randomize the base address of executable modules at load time. Changing the load address of modules is also often needed for resolving conflicts among shared libraries with the same preferred base address. In Windows, loading a module at an arbitrary address depends on compiler-generated *relocation* information, which specifies the absolute code or data addresses in the module that must be adjusted due to the module's relocation at a non-preferred base address. Relocation information, however, is often stripped from production builds of legacy software, making it more susceptible to code-reuse attacks, as ASLR is not an option.

In this paper, we introduce a technique to enable ASLR for executables with stripped relocation information by incrementally adjusting stale absolute addresses at runtime. The technique relies on runtime monitoring of memory accesses and control flow transfers to the original location of a relocated module using page table manipulation techniques. Depending on the instruction and memory access type, the system identifies stale offsets, reconstructs their relocation information, and adjusts them so that subsequent accesses to the same locations proceed directly, without any intervention. To improve performance further, the reconstructed relocation information is preserved across subsequent runs of the same program. We have implemented a prototype of the proposed technique for Windows XP, which is transparently applicable to third-party stripped binaries, and have experimentally evaluated its performance and effectiveness. Our results demonstrate that incremental runtime relocation patching is practical, incurs modest runtime overhead for initial runs of protected programs, and has negligible overhead on subsequent runs.

1 Introduction

Keeping systems up-to-date with the latest patches, updates, and operating system versions, is a good practice for eliminating the threat of exploits that rely on previously disclosed vulnerabilities. Major updates or newer versions of operating systems and applications also typically come with additional or improved security protection and exploit mitigation technologies, such as the stack buffer overrun detection (/GS), data execution prevention (DEP), address space layout

randomization (ASLR), and many other protections of Windows [27], which help in defending against future exploits.

At the same time, however, updates and patches often result in compatibility issues, reliability problems, and rising deployment costs. Administrators are usually reluctant to roll out new patches and updates before conducting extensive testing and cost-benefit analysis [34], while old, legacy applications may simply not be compatible with newer OS versions. It is indicative that although Windows XP SP3 went out of support on April 8th, 2014 [7], many home users, organizations, and systems still rely on it, including the majority of ATMs [1]. In fact, the UK and Dutch governments we forced to negotiate support for Windows XP past the cutoff date, to allow public-sector organizations to continue receiving critical security updates for one more year [6].

As a step towards enhancing the security of legacy programs and operating systems that do not support the most recent exploit mitigation technologies, application hardening tools such as Microsoft's EMET (Enhanced Mitigation Experience Toolkit) [25] can be used to retrofit these and even newer (sometimes more experimental) protections on third-party legacy applications. An important such protection is address space layout randomization, which aims to defend against exploitation techniques based on code reuse, such as return-to-libc [15] and return-oriented programming (ROP) [36].

ASLR randomizes the load address of executables and DLLs to prevent attackers from using data or code residing at predictable locations. In Windows, though, this is only possible for binaries that have been compiled with *relocation* information. In contrast to Linux shared libraries and PIC executables, which contain position-independent code and can be easily loaded at arbitrary locations, Windows portable executable (PE) files contain absolute addresses, e.g., immediate instruction operands or initialized data pointers, that are valid only if an executable has been loaded at its preferred base address. If the actual load address is different, e.g., because another DLL is already loaded at the preferred address or due to ASLR, the loader adjusts all fixed addresses appropriately based on the relocation information included in the binary.

Unfortunately, PE files that do not carry relocation information cannot be loaded at any address other than their preferred base address, which is specified at link time. Relocation information is often stripped from release builds, especially in legacy applications, to save space or hinder reverse engineering. Furthermore, in 32-bit Windows, it is not mandatory for EXE files to carry relocation information, as they are loaded first, and thus their preferred base address is always available in the virtual address space of the newly created process. For these reasons, tools like EMET unavoidably fail to enforce ASLR for executables with stripped relocation information. Consequently, applications with stripped relocation information may remain vulnerable to code reuse attacks, as DEP alone can protect only against code injection attacks. Furthermore, recently proposed protection mechanisms for Windows applications rely on accurate code disassembly, which depends on the availability of relocation information, to apply control flow integrity [45] or code randomization [28].

In this work, we present a technique for reconstructing the missing relocation information from stripped binaries, and enabling safe address space layout randomization for executables which are currently incompatible with forced ASLR. The technique is based on discovering at runtime any stale absolute addresses that need to be modified according to the newly chosen load address, and applying the necessary fixups, replicating in essence the work that the loader would perform if relocation information were present. As transparency is a key requirement for the practical applicability of protections tailored to third-party applications, the proposed approach relies only on existing operating system facilities (mainly page table manipulation) to monitor and intercept memory accesses to locations that need fixup.

We have evaluated the performance and effectiveness of our prototype implementation using the SPEC benchmark suite, as well as several Windows applications. Based on our results, incremental runtime relocation patching is practical, incurs modest runtime overhead for initial runs of protected programs, and has negligible overhead on subsequent runs, as the reconstructed relocation information is preserved. Besides forced ASLR, the proposed technique can also be used to resolve conflicts between stripped binaries with overlapping load addresses, a problem that occasionally occurs when running legacy applications, and to significantly improve code disassembly.

The main contributions of this work are:

- We present a technique for dynamically reconstructing missing relocation information from stripped binaries. Our technique can be used to enable forced ASLR or or resolve base address conflicts for third-party non-relocatable binaries.
- We have implemented the proposed approach as a self-contained software hardening tool for Windows applications, and describe in detail its design and implementation.
- We have experimentally evaluated the performance and correctness of our approach using standard benchmarks and popular applications, and demonstrate its effectiveness.

2 Background

The wide support for non-executable memory page protections [27, 30] in recent operating systems and processors has given rise to code reuse attacks, such as return-to-libc [15] and return-oriented programming (ROP) [36], which allow the exploitation of memory corruption vulnerabilities by transferring control to code that already exists in the address space of the vulnerable process. Return-oriented programming, in particular, has become the primary exploitation technique for achieving arbitrary code execution against Windows applications. In contrast to return-to-libc, the reused code in ROP exploits consists of small instruction sequences, called *gadgets*, scattered throughout the executable segments of the targeted process.

To reuse code that already exists in the address space of a vulnerable process, an attacker needs to rely on a priori knowledge of its exact location (although in some cases the location of code can be inferred dynamically during exploitation [8, 10, 20, 23, 35, 42, 43]). Address space layout randomization (ASLR) [11, 27, 29] protects against code reuse attacks by randomizing the location of loaded executable modules, breaking the assumptions of the attacker about the location of any code of interest. Besides address space randomization, process diversity [13, 16] can also be increased by randomizing the code of executable segments, e.g., by permuting the order of functions [2, 11, 12, 22] and basic blocks [3, 5], or by randomizing the code itself [19, 28, 44].

In Windows, which is the main focus of this work, ASLR support was introduced in Windows Vista. By default, it is enabled only for core operating system binaries and programs that have been configured to use it through the `/DYNAMICBASE` linker switch. For legacy applications, not compiled with ASLR support and other protection features, Microsoft has released the Enhanced Mitigation Experience Toolkit (EMET) [25], which can be used to retrofit ASLR and other exploit mitigation technologies on third-party applications. A core feature of EMET is Mandatory ASLR, which randomizes the load address of modules even if they have not been compiled with the `/DYNAMICBASE` switch, but do include relocation information. This is particularly important for applications that even though have opted for ASLR, may include some DLLs that remain in static locations, which are often enough for mounting code reuse attacks [17, 21, 47]. EMET's ASLR implementation also provides higher randomization entropy through additional small memory allocations at the beginning of a module's base address. Many of the advanced ASLR features of EMET have been incorporated as native functionality in Windows 8, including forced ASLR.

The above recent developments, however, are not always applicable on legacy executables. Typically, when creating a PE file, the linker assumes that it will be loaded to a specific memory location, known as its *preferred base address*. To support loading of modules at addresses other than their preferred base address, PE files may contain a special `.reloc` section, which contains a list of offsets (relative to each PE section) known as "fixups" [38]. The `.reloc` section contains a fixup for each absolute addresses at which a delta value needs to be added to maintain the correctness of the code in case the actual load address is different [32]. Although DLLs typically contain relocation information, release builds of legacy applications often strip `.reloc` sections to save space or hinder reverse engineering. This can be achieved by providing the `/FIXED` switch at link time. Furthermore, in older versions of Visual Studio, the linker by default omits relocation information for EXEs when performing release builds, as the main executable is the first module to be loaded into the virtual address space, and thus its preferred base address is always expected to be available.

As modules (either EXEs or DLLs) with stripped relocation information cannot be loaded at arbitrary addresses, the OS or tools like EMET cannot protect them using ASLR. Legacy applications may also occasionally encounter address conflicts due to different modules that attempt to use the same preferred

base address. Our system aims to enable the randomization of the load address of modules with stripped relocation information by incrementally adjusting stale absolute addresses at runtime.

3 Approach

Our approach to the problem of relocating stripped binaries relies on reconstructing the missing relocation info by discovering such relocatable offsets at runtime. We note here that a static approach, i.e., using disassembly to find all the relocatable offsets, would be much more difficult, if not infeasible in many cases—the reason being that stripped binaries also lack debugging symbols, so complete disassembly coverage would be impossible in most cases.

3.1 Overview

The basic idea of our approach is to load the stripped binary at a random location and monitor any data accesses or control transfers to its original location. Any such access to the original location is either a result of using a relocatable offset or an attack attempt (the attacker might try to reuse parts of the original code, not knowing that the binary was relocated). The next step is to identify the source of the access by checking whether it was indeed caused by a relocatable offset. In this case, the offset it located, its value is fixed to the new random base, and the relocation info is reconstructed so as next time the same program is executed a fixup for that address can be automatically applied.

Although there are a few different ways to monitor memory access and control transfers at runtime, we followed an approach that minimizes its effects and dependencies on third-party components. For instance, instruction-level dynamic binary instrumentation was not considered for this reason, as it requires the installation of third-party dynamic binary instrumentation frameworks (and typically incurs a prohibitively high runtime overhead). Our monitoring facility is built around basic operating system functionality, mostly memory protection mechanisms. More precisely, after a binary is loaded to a random location, we change the permissions of its original location to inaccessible, so as each time a memory access or control transfer happens to one of the original locations, a memory violation exception is raised. This type of exception usually contains the location of the instruction that caused it, the faulting address (can be the same as the instruction location), and the type of access (read or write).

The main challenge of our approach now becomes to identify whether an access to the original binary location is caused by a relocatable offset and how to trace it back to that offset. To better explain this issue, consider the following example. Assume that an instruction updates the contents of a global variable using its absolute address (e.g., `0x1000`). When the instruction is executed from the new, randomly chosen location of the binary, an exception will be raised. At this point, we know the location of the instruction and the faulting address (`0x1000`). After analyzing the faulting instruction, we see that one of its

operands is actually the faulting address. In this case, we have to fix the operand by adjusting it to the new random base, and also reconstruct the relocation info of this offset.

The example above is the most straightforward case of identifying a relocatable offset. In practice, in most cases the relocatable offset is not part of the faulting instruction. For example, consider the case of dereferencing a global pointer. There is an instruction to load the value of the pointer, probably in a register, and another instruction to read the contents of the memory location stored in the register. In this case, the faulting address is not directly related with the faulting instruction. Even worse, there are cases in which the relocatable offset has been changed before it is used. For example, accessing a field from a structure in a global array would only require a single relocatable address (the location of the array) and would result in many runtime accesses within the range of the array. It is very difficult to trace such an access reliably back to its source relocatable offset.

However, code-reuse attacks rely solely on the knowledge of the code's location, regardless of the location of data. Based on this observation, and due to the problematic nature of data pointer tracing, we focus on randomizing the load address of code segments only. Code pointers are usually guaranteed not to support any arithmetic—it would be difficult to imagine code that depends on expressions such as adding a few bytes to the location of a function start, at least for compiler-generated code. An exception to this is jump tables that contain relative offsets, but this is a case that can be easily covered, as we will see later on. This simplifies the overall approach, without sacrificing any of the security guarantees.

Figure 1 shows a high-level overview of our approach. When a stripped binary is loaded for execution (left side), its code segment is moved to a random location, while the original location becomes inaccessible (right side). Then, whenever there is a memory access or control transfer to the original location (solid arrow), the faulting address along with the instruction that caused it are analyzed. Based on this analysis, the source relocatable offset is pinpointed, gets fixed, and its relocation information is reconstructed. In the following, we describe in more detail how this analysis is being performed.

3.2 Access Analysis

The series of steps performed after a memory access violation exception is raised due to a memory access in the original code location is depicted in Figure 2. Broadly speaking, access violations are grouped into two categories based on their root cause: (i) reading the contents of the original code segment, and (ii) control transfers to the original code segment. To distinguish between the two, the system checks whether the value of the instruction pointer is within the original code segment.

In practice, the first case corresponds mostly to indirect jump instructions that read their target from the code segment. These are typically part of jump tables, which are used for the implementation of switch statements in C. In the

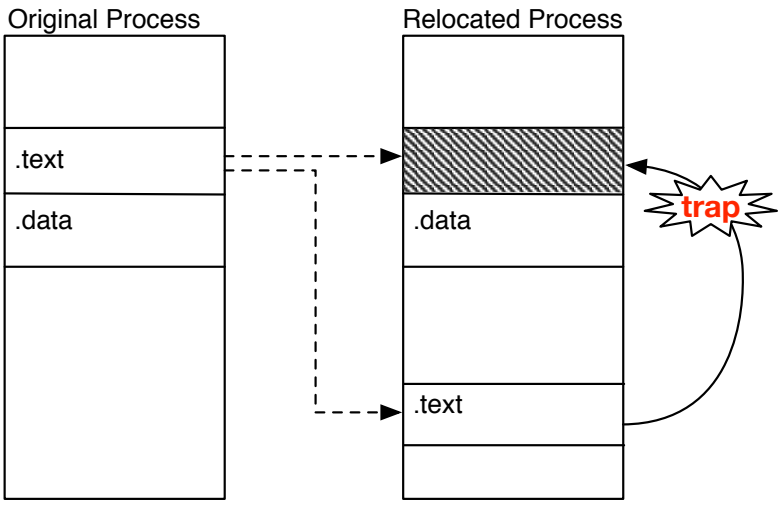

Fig. 1. High-level overview of runtime relocation fixup. The code segment of a stripped binary is loaded to a randomly chosen location, and its original memory area is marked as inaccessible. Memory accesses and control transfers to any of the original locations are trapped. Relocation information is then reconstructed by analyzing the faulting instruction.

second case, control is transfered to the original code segment because a code pointer that has not been relocated is used. This could be a simple function pointer, part of a C++ virtual table (vtable), or a static one, represented as an immediate value in an instruction. In the following subsections we describe in detail how each of these cases is handled.

When control is transferred to locations in the original code segments for which there is no code pointer, or when we can not verify it as a legitimate code pointer, these transfers are flagged as code-reuse attempts (see Fig. 2). This effectively allows attackers to reuse code paths for which there are legitimate code pointers (e.g., function entries or jump table targets), given that they have not been reconstructed yet. Arguably, this leaves a very limited set of gadgets for the attacker, which quickly shrinks further as relocatable code pointers are identified.

3.3 Jump Tables

A jump table is an array of code targets that is usually accessed through an indirect jump. The following is an example of such a jump table in x86 assembly (taken from `gcc`'s binary):

```
.text:004D5CCE     jmp  ds:off_4D6864[eax*4] ; switch jump
...
; DATA XREF: _main+2CE ; jump table for switch statement
```

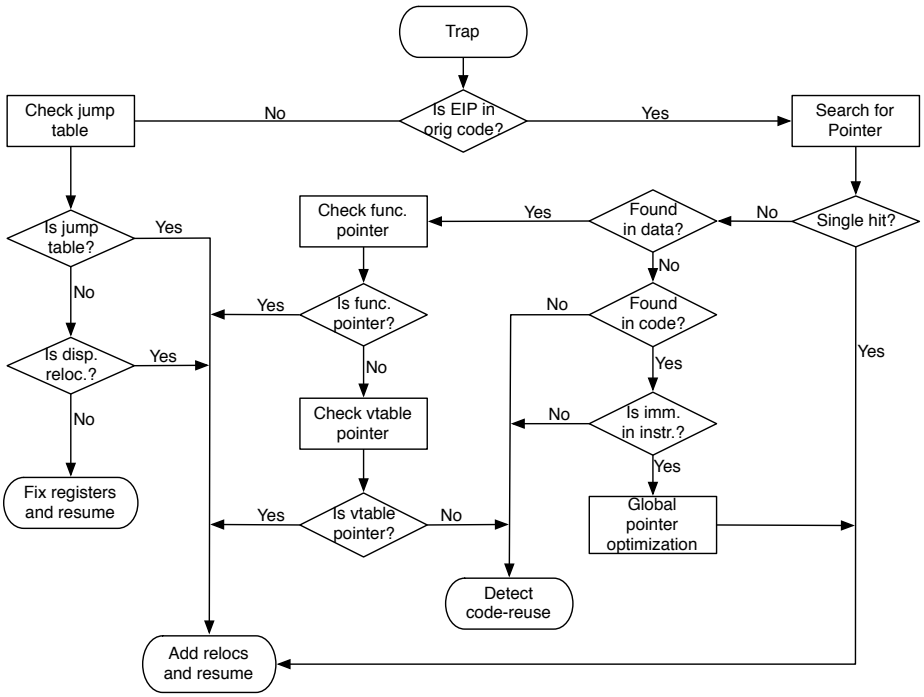

Fig. 2. Flow graph of the procedure followed after a memory access exception (trap) is generated. If the instruction pointer (EIP register) at the time of the exception is within the original code segment, the system performs pointer verification, otherwise the faulting instruction is fixed.

```
.text:004D6864 off_4D6864    dd offset loc_4D5D53
.text:004D6868               dd offset loc_4D5D63
.text:004D686C               dd offset loc_4D5D93
.text:004D6870               dd offset loc_4D5D8B
```

When the `jmp` instruction is executed from the new random location, an exception is going to be raised, with the faulting address being `(0x4D6864 + eax * 4)`. This is handled as follows: i) starting from the location pointed to by the faulting address, we scan the bytes before and after that location for more addresses and fix them, and ii) we also fix the relocatable offset in the address operand of the indirect jump instruction. In case of jump tables with relative offsets, we just skip the first step.

3.4 Pointer Verification

After jump tables are covered, we only expect to see control flow transfers to the locations of the original code. In these cases, the location of the faulting instruction is also the faulting address—there is no information about the source

instruction. Given a faulting address, the whole code segment and initialized data are scanned for all its occurrences. If there is a single occurrence, we assume that it is a relocatable offset, which is handled appropriately. Otherwise, for each occurrence in the code segment, we verify that it is indeed part of a valid instruction—more precisely, an immediate operand.

Occurrences found in the initialized data segments are a bit more complicate to cover. Usually, for such a hit to be indeed a relocatable offset, it has to be a variable holding a function pointer, so there should be a way of accessing that variable. To verify this, we just need to find a data reference to that variable. In addition, function pointers can be parts of structures, arrays, or a combination of both. In general, we verify that an occurrence of the faulting address in the data segment is a relocatable offset that needs to be fixed if we can find a reference to or near its location (given as a parameter).

The following example illustrates the function pointer verification process. Assume there is a global variable that is statically initialized with the address of a function. Also, there is an indirect call instruction that reads the value of the global variable and transfers control to its value. At runtime, the value is going to be read (because the data segment is not relocated) and an exception is going to be raised when control is transfered to the function. Both the faulting address and the faulting instruction will correspond the beginning of the target function. At this point, we find an occurrence in the code segment and verify that it belongs to an instruction—which is the indirect call in this case.

Another use of function pointers is in C++ virtual tables, which is how dynamic class methods are represented. These pointers are handled a bit differently than simple function pointers, and, for this reason, we have introduced special checking rules. We first verify that there is a move instruction that copies the head of the table to a newly created class instance, by finding a move instruction that references a memory location close to the place where the code pointer was found. We then also verify that the control was transferred by an indirect call through a register, by reading the current value at the top of the runtime stack (return address) and disassembling the instruction right before the location it points to. Bellow is a real example taken from the `eon` binary of the SPEC benchmarks suite:

```
;; function call
.text:004017F9    mov   eax, [ecx]   ; ecx is this ptr
.text:004017FB    mov   eax, [eax+24h]
.text:004017FE    push  edx
.text:004017FF    mov   edx, [ebp+arg_4]
.text:00401802    push  edx
.text:00401803    mov   edx, [ebp+arg_0]
.text:00401806    push  edx
.text:00401807    call  eax
.....
```

```
;; vtable (the static part)
; DATA XREF: sub_409B40+8o  ; sub_40B0E0+2Fo
.rdata:00461D24 off_461D24    dd offset sub_40AAD0
.rdata:00461D28               dd offset sub_409BB0
.rdata:00461D2C               dd offset sub_409BC0
.....
;; copying the head of the table
.text:0040B10C    lea   ecx, [esi+4]    ; this
.text:0040B10F    mov   dword ptr [esi], offset off_461D24
```

The top part of the example shows the code that loads the function pointer from the vtable to the `eax` register and then transfers control there by calling it. The call instruction at the end will actually going to raise an exception. While handling the exception, we check (i) the table that contains the faulting address at `0x461D24` (middle part) is referenced by a move instruction at `0x40B10F` (bottom part), and (ii) the instruction before the return address is a call instruction with a register operand (at `0x401807`).

3.5 Dynamic Data

Although in order to reconstruct the missing relocation information we need to locate relocatable offsets within the image of the executable module, copies of such values also appear in dynamic data (e.g., in the stack or heap). This is the result, for example, of a global pointer being copied in a structure field that was dynamically allocated. In this case, an exception is going to be raised when the copy of the pointer (in the structure) is used. As described before, our technique is going to trace the original relocatable offset. This is sufficient for reconstructing the relocation information for this pointer, and avoid dealing with the same problem next time the same program is executed. However, we do not take any further actions to deal with copies in dynamic data. Thus, we might have to handle more than one exceptions for the same relocatable value during the same run in which it was first discovered. This, of course, does not affect the correctness and robustness of the technique in any way, but can affect overall performance.

To avoid the performance penalty under some cases, while not weakening our original approach, we added a simple optimization for global pointers. Each time a relocatable offset is fixed, and it is found to be the source operand of an instruction that copies it over to a global data location, we check whether the destination memory location contains the same value and relocate that copy, too. Below is an example of a few such instructions (taken from `gcc`'s binary):

```
.text:004D5A69    mov   dword_550968, offset loc_4D1F10
.text:004D5A73    mov   dword_550AAC, offset loc_4D1C20
.text:004D5A7D    mov   dword_5509C4, offset nullsub_1
```

The first `mov` instruction in the above example copies the (relocatable) offset `loc_4D1F10` to the global data memory location `0x550968`. At the time an

exception is raised because control was transfered to address `0x4D1F10`, the source operand of the first `mov` instruction will be fixed, and, if the same value is found at address `0x550968`, that will be fixed as well. In this way, future copies of the relocatable offset will point to the new code location, and no more exceptions will be raised for this instance.

In general, when this optimization is not applicable and there are many copies of relocatable offsets being repeatedly used, we have the option to set an access threshold, beyond which the system can inform the user that restarting the program would greatly increase its performance. Still, we believe that this is a minor issue, as it might occur only in the first few times a program is executed. After that, the relocation information of the majority of the relocatable offsets will have been reconstructed.

4 Implementation

We built a prototype of the described technique for the Windows platform. Most of the development of the tool was done on Windows XP. However, as the APIs we use have not changed in more recent versions of the operating system, our prototype supports even the latest version, which is Windows 8.1 at the time of writing.

The most significant part of the implementation is built on top of the Windows Debugging API [26], with the addition of some other standard functions (e.g., `CreateProcess`). This API is designed to work between two processes: the parent process is responsible for spawning a child process, and then capture and analyze any debug events the child generates. Debug events include memory access violation exceptions, process/thread startup/termination, and so on. Our implementation is bundled as a single application (about 1.5 KLOC) which can be executed from the command prompt, and receives the path of the target program to be protected as a command-line argument.

At a higher level, there are two phases of operation: initialization and runtime. We discuss both in sufficient detail in the rest of this section.

4.1 Initialization

The first step during the initialization phase is to spawn the process, while passing the appropriate arguments in order to enable debugging. The very first debug event generated by the child process is a process creation event, which is handled by the parent by performing the following tasks before resuming the execution of the child process. Initially, the Portable Executable (PE) headers are parsed. These headers include information such as the boundaries of each section (data, code, etc.) and the entry point of the code. Given that information, we proceed by copying the code section to a new, randomly chosen location using the `ReadProcessMemory` and `WriteProcessMemory` API functions, while changing the memory protections of the original code segment to inaccessible using `VirtualProtectEx`.

In order to improve the performance of certain runtime operations, a hash table of all possible code pointer values is built. This is done by scanning all sections and inserting any four-byte values (assuming 32-bit processes) that fall into the address range of the original code segment. Finally, we check whether there is a file that contains relocation information that was discovered as part of previous runs, and apply them.

4.2 Runtime

After initialization is completed and control is given back to the child process, the parent blocks while waiting for the next debugging event. Usually, we expect memory access violation exceptions to be generated after this stage. New DLL loaded events might happen as well, but rarely. Whenever a new DLL is loaded in the address space of the child process, the system checks whether it contains relocations. In case it does not, the same initialization steps that were previously described are performed.

As described in Section 3, the core of our technique is implemented as part of the handling mechanism of memory access violation exceptions. Each exception record contains information about the location of the instruction that caused it, along with the faulting address. Based on this information, we distinguish between two main cases: i) the instruction pointer falls within the address range of the original (inaccessible) code segment (instruction address and faulting address are the same), and ii) an illegal memory access was made by an instruction located in the relocated code segment (instruction address and faulting address are different).

If the instruction pointer after a memory exception is received falls within the original code segment, this means that the control flow was transfered there and the program failed when it tried to execute the next instruction. In this case, the faulting address corresponds to the location of the instruction in the exception record. The exception is handled by first looking up the faulting address in the hash table—which is constructed during the initialization phase. A single hit is the simplest case, because it means that this is the source of the exception. If there are more than one hits, each one is verified using the rules described in Section 3 for immediate values or function pointers.

Alternatively, if the faulting instruction belongs to the relocated code segment, this means that one of its operands caused the fault. This happens under two circumstances: the instruction is an indirect jump, reading a jump table target from the original code location, or an instruction that uses a copy of a relocatable value from dynamic data.

5 Evaluation

In this section we present the results of the experimental evaluation of our prototype in terms of correctness and performance overhead. For the largest part of our evaluation, we used benchmarks from SPEC CPU2006 [4], as well as some

Program	Possible Pointers	Jump Tables	Verified Pointers	Single Hit	Dynamic Data	Global Opt.	Reconst. Reloc.
perlbench	31,260	118	633	83.0%	43M	41	2,614
bzip2	2,147	4	11	84.6%	25	4	76
gcc	98,955	510	1,008	65.2%	73M	269	7,849
mcf	1,875	1	13	100.0%	19	-	22
gobmk	69,852	21	968	63.5%	4M	54	1,270
hmmer	4,798	15	17	94.4%	42	2	152
sjeng	8,460	12	17	100.0%	18	-	135
h264ref	17,526	17	27	71.0%	320K	61	209
omnetpp	24,861	13	1,509	90.6%	269K	8	1,669
astar	2,690	2	20	100.0%	31	-	42
xalancbmk	141,246	54	4,402	84.2%	9M	24	5,392

Table 1. Statistics from running the SPEC benchmarks using the reference input data (largest dataset).

real-world applications, such as Internet Explorer and Adobe Reader. All the experiments were performed on a computer with the following specifications: Intel Core i7 2.00GHz CPU, 8GB RAM, 256GB SSD with 64-bit Windows 8.1 Pro.

5.1 Statistics

We started our evaluation with the goal of getting a better feeling on the differences of applying our technique to programs with distinct characteristics. First, we selected all the test programs in the integer suite that come with the SPEC benchmark and stripped the relocation information from the compiled binaries. Out of the twelve programs in that set, only `libquantum` had to be left out because it uses some C99 features that are not supported by Visual C++ (as noted in the SPEC configuration file Example-windows-ia32-visualstudio.cfg). Then, we executed each one using our prototype and gathered some valuable statistics that provide insights about the runtime behaviour of our technique. At the same time, we checked that the output of the benchmark test runs was correct, which in turn verified the correctness of our implementation under these cases.

Table 1 shows the results of this run. The first column contains the name of each SPEC test program, followed by the number of possible pointers that we identified for each during the initialization phase. The next three columns show the number of identified jump tables and the number of verified pointers along with the percentage of them that had a single hit in the possible pointers set. Next, we have the number of times that an already fixed relocatable offset reappeared at runtime because of copies of it in dynamic data, followed by the number of global pointer copies that we were able to apply the optimization described in the last part of Section 3. Finally, the number of actual relocatable

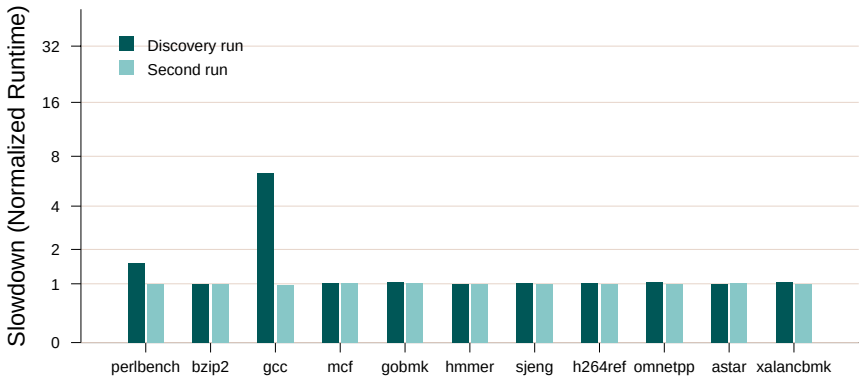

Fig. 3. Normalized slowdown compared to normal execution (no relocation). Dark-colored bars show the slowdown during the first run, where most of the relocations are discovered and there are still copies of them in dynamic data. Light-colored bars show the slowdown during the second run (and any subsequent runs) where most of the relocations have already been discovered.

offsets that we were able to reconstruct their relocation information in shown in the last column.

An interesting observation is that most of the times we have a single hit during the verification of a code pointer, which simplifies the overall procedure. Another interesting thing to note is that there is a very high variation in the number of times that a copy of an already fixed relocatable offset in dynamic data is used. This ranges from a few tens to tens of millions using these test cases. At the same time, we note that there does not seem to be any significant correlation of this number and the actual number of the reconstructed relocatable offsets.

5.2 Performance Overhead

Next, we focus on evaluating the performance overhead. As already mentioned, the only case where we expect our technique to affect the performance of a target application is during the first (or, few first) times we execute it, where most of the relocations are being discovered. Any consecutive execution should have a minimal runtime overhead impact.

Figure 3 shows the normalized slowdown for the first execution of the SPEC programs under our prototype (Discovery run) and another execution after the relocations have been discovered (Second run). In both cases, the slowdown is compared to a normal execution without relocating the program (baseline). Also, the input data used for this experiment was the reference dataset (i.e., the largest dataset), where the average completion time for each test program is a couple of minutes. As expected, we see that the overhead of the second run is minimal (less than 5% on average) and mostly attributed to the unoptimized way of applying the discovered relocation information. Currently, in our prototype

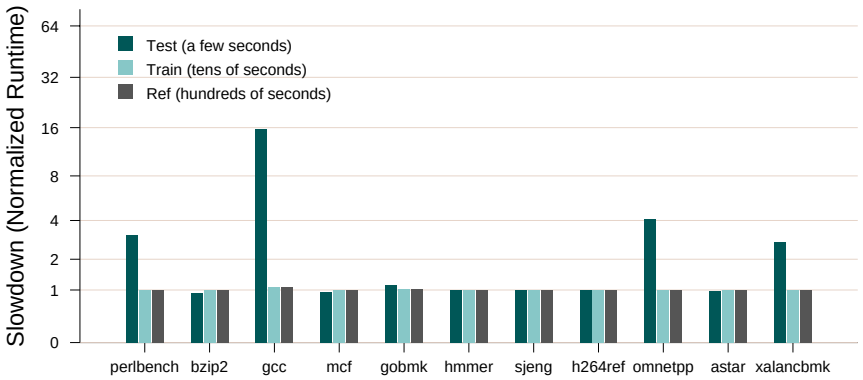

Fig. 4. Avoiding the performance hit during the dynamic relocation discovery phase (first run) by gradually increasing the input size on each execution. The overall time in this case is much less compared to running a program using large input the first time.

implementation we relocate every offset separately. For each of them, we read its value, change the memory permissions, update its value and restore the memory permissions. The unusually high performance overhead that we observed when executing `gcc` is due to the fact that it contains a high number of relocatable offset copies in dynamic data (see Table 1). Although, that overhead does disappear in any consecutive execution, there is not much we can do at this point, except asking the user to restart the execution of the program in order to take advantage of the already discovered relocatable offsets. An alternative strategy is to ask the user to start with a very small input and progressively increase the workload of the program during the first few executions, until the majority of the relocations are discovered.

To demonstrate the effectiveness of that strategy, we applied it on the SPEC CPU2006 benchmarks. These test programs come with three different inputs: a very small test dataset used for verifying the functionality of the programs, a medium-sized train set used for feedback-directed optimizations and the reference dataset, which is much larger that the other two. For all the results up to this point, we have used the reference dataset. Figure 4 shows the normalized slowdown of applying our technique to the same SPEC programs, but while increasing the workload (from test, to train and reference) this time. Also, during each execution, we allow our prototype to use any reconstructed relocation information that has been discovered from previous executions. The slowdown of the reference dataset is much less compared to the one reported in Figure 3. Moreover, the overall discovery phase (which is now broken down to three executions) is much quicker compared to Figure 3, in absolute numbers. Even though `gcc` seems to have a larger slowdown with the test dataset than before, this accounts for 22 seconds, plus a few minutes for the next two executions, compared to 48 minutes when using the large reference dataset during the first execution.

5.3 Use Cases

The final part of our evaluation focuses on the feasibility of applying our technique on popular, real-world applications. For this purpose, we installed older versions of both Internet Explorer and Adobe Reader, where the relocation info of their EXE files was stripped. The exact versions we used were 6.0.2900.5512 and 8.1.2, respectively. In both cases, the code size of the non-relocatable EXE was relative small, approximately 10KB. Using our prototype implementation of our technique we were able to successfully relocate the code segments to a new and random location, while not breaking the functionality of the applications. The number of relocatable offsets for which we reconstructed their relocation information was 18 for Internet Explorer and 3 for Adobe Reader. Although it is just a small number of relocations, reconstructing this information is crucial in protecting these applications.

6 Related Work

We divide the related work into two parts. First, we review work that is related to address space layout randomization. Reconstructing relocation information enables or improves the accuracy of these proposals. Second, we review work from the field of dynamic data structure excavation, where similar techniques to ours are used.

6.1 Code Randomization and Disassembly

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 [13, 16]. Address space layout randomization [27, 29] is probably one of the most widely deployed countermeasures against code-reuse attacks. The problem of randomizing non-relocatable executable files was identified early on, with the first ASLR implementations for Linux by the PaX project, and an approach based on the interception of page faults to the original locations was proposed [31]. Our work is based on the same core idea, but our implementation focuses on Windows executables, we extend it with patching support to reduce runtime overhead, and experimentally evaluate it.

In practice, however, the effectiveness of ASLR is hindered by code segments left in static locations [17,21,47], while, depending on the randomization entropy, it might be possible to circumvent it using brute-force guessing [37]. 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 [8, 10, 20, 23, 35, 42, 43].

To overcome the limitations of the original design, more fine-grained forms of randomization [19,28,44] have been proposed. These can be statically applied on stripped binaries and randomize code at the instruction level (instead of randomizing the base address only). Their accuracy and correctness, however, heavily

depends on the accuracy of disassembly and control flow graph extraction, which is improved when relocation information is available.

Control flow integrity [9] is another protection scheme that confines program execution within the bounds of a precomputed profile of allowed control flow paths. Although its original implementation depends on debug symbols for the complete extraction of the control flow graph, recent proposals have demonstrated how more relaxed forms of the same technique can be applied on stripped binaries [45, 46]. Again, for legacy applications, these techniques would benefit from the improved control flow extraction based on the availability of relocations.

Finally, although binary rewriting is still possible in the absence of relocation information, it relies on dynamic instrumentation for indirect calls/jumps [41]. This makes the overall runtime overhead of the technique to depend on the number of executed indirect calls/jumps, which are very frequent in C++ applications.

6.2 Dynamic Data Structure Excavation

Another body of work that uses related techniques to ours is dynamic data structure excavation [14, 18, 24, 39, 40]. By looking at memory access patters dynamically at runtime, these techniques are able to infer the type of binary data, such as data structures and arrays.

Laika [14] employs Bayesian unsupervised learning to detect data structures. Possible object positions and sizes are identified by using potential pointers in the process' memory image. Although sufficient for cases like evaluating the similarity between malware samples, Laika's output is not precise enough for debugging or reverse engineering. Similar to Laika, Rewards [24] reconstructs type information dynamically, based on abstract structure identification [33]. A fundamental limitation of this approach is that it is not capable of identifying data structures that are internal to a module. Howard [40] improves on the precision of data structure excavation by applying a set of specific rules to identify data structures dynamically. Arrays, structure fields, etc. are recognized based on runtime memory access patterns.

7 Conclusion

Address Space Layout Randomization (ASLR) has proven to be a very effective mitigation against code reuse attacks, making successful exploitation much harder. Unfortunately, ASLR depends on some information that is often stripped from executable files.

As a step towards addressing this limitation, we designed and implemented a technique to dynamically reconstruct this missing information, which effectively enables ASLR even on programs that are otherwise incompatible. The results of our experimental evaluation focusing on performance measurements and use cases with real-world applications clearly show the practicality of the proposed approach.

Acknowledgements. This work was supported by the US Air Force, the Office of Naval Research, and DARPA through Contracts AFRL-FA8650-10-C-7024, N00014-12-1-0166 and FA8750-10-2-0253, respectively, with additional support by Intel Corp. This material is based upon work supported by (while author Keromytis was serving at) the National Science Foundation. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, the Air Force, ONR, DARPA, NSF, or Intel.

References

1. ATMs Face Deadline to Upgrade From Windows XP. http://www.businessweek.com/articles/2014-01-16/atms-face-deadline-to-upgrade-from-windows-xp.
2. /ORDER (put functions in order). http://msdn.microsoft.com/en-us/library/00kh39zz.aspx.
3. Profile-guided optimizations. http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx.
4. SPEC CPU2006 Benchmark. http://www.spec.org/cpu2006.
5. Syzygy - profile guided, post-link executable reordering. http://code.google.com/p/sawbuck/wiki/SyzygyDesign.
6. UK government pays Microsoft £5.5m to extend Windows XP support. http://www.theguardian.com/technology/2014/apr/07/uk-government-microsoft-windows-xp-public-sector.
7. Windows XP SP3 and Office 2003 Support Ends April 8th, 2014. http://www.microsoft.com/en-us/windows/enterprise/endofsupport.aspx.
8. 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/.
9. 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.
10. 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.
11. 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.
12. 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.
13. Frederick B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12:565–584, October 1993.
14. Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, OSDI'08, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.
15. Solar Designer. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63.

16. 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.
17. 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.
18. Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA)*, pages 255–265, Portland, ME, USA, July 18–20, 2006.
19. 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.
20. 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.
21. Richard Johnson. A castle made of sand: Adobe Reader X sandbox. CanSecWest, 2011.
22. 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.
23. Haifei Li. Understanding and exploiting Flash ActionScript vulnerabilities. CanSecWest, 2011.
24. Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.
25. Microsoft. Enhanced Mitigation Experience Toolkit. `http://www.microsoft.com/emet`.
26. Microsoft. Windows Debugging API. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms679303(v=vs.85).aspx`.
27. Matt Miller, Tim Burrell, and Michael Howard. Mitigating software vulnerabilities, July 2011. `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788`.
28. 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.
29. PaX Team. Address space layout randomization, 2003. `http://pax.grsecurity.net/docs/aslr.txt`.
30. PaX Team. Non-executable pages design & implementation, 2003. `http://pax.grsecurity.net/docs/noexec.txt`.
31. PaX Team. Non-relocatable executable file randomization, 2003. `http://pax.grsecurity.net/docs/randexec.txt`.
32. Matt Pietrek. An in-depth look into the Win32 portable executable file format, part 2. `http://msdn.microsoft.com/en-us/magazine/cc301808.aspx`.
33. G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *In Symposium on Principles of Programming Languages (POPL)*, pages 119–132, 1999.

34. Eric Rescorla. Security holes... Who cares? In *Proceedings of the 12th USENIX Security Symposium*, pages 75–90, August 2003.
35. 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`.
36. 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.
37. 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.
38. Skape. Locreate: An anagram for relocate. *Uninformed*, 6, 2007.
39. Asia Slowinska, Traian Stancescu, and Herbert Bos. Dde: dynamic data structure excavation. In *Proceedings of the 1st ACM SIGCOMM Asia-Pacific Workshop on Systems (ApSys)*, pages 13–18, 2010.
40. Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.
41. Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. Binary rewriting without relocation information. *University of Maryland, Tech. Rep*, 2010.
42. 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.
43. Peter Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. `http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf`.
44. 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.
45. 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.
46. Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium*, pages 337–352, Berkeley, CA, 2013. USENIX.
47. Dino A. Dai Zovi. Practical return-oriented programming. SOURCE Boston, 2010.