VTPin: Practical VTable Hijacking Protection for Binaries

Pawel Sarbinowski
Vrije Universiteit Amsterdam
onexemailx@gmail.com

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

Elias Athanasopoulos
Vrije Universiteit Amsterdam
i.a.athanasopoulos@vu.nl

ABSTRACT

VTable hijacking has lately been promoted to the de facto technique for exploiting C++ applications, and in particular web browsers. VTables, however, can be manipulated without necessarily corrupting memory, simply by leveraging use-after-free bugs. In fact, in the recent Pwn2Own competitions all major web browsers were compromised with exploits that employed (among others) use-after-free vulnerabilities and VTable hijacking.

In this paper, we propose VTPin: a system to protect against VTable hijacking, via use-after-free vulnerabilities, in large C++ binaries that cannot be re-compiled or re-written. The main idea behind VTPin is to *pin* all the freed VTable pointers on a safe VTable under VTPin's control. Specifically, for every object deallocation, VTPin deallocates all space allocated, but preserves and updates the VTable pointer with the address of the safe VTable. Hence, any dereferenced dangling pointer can only invoke a method provided by VTPin's *safe* object. Subsequently, all virtual-method calls due to dangling pointers are not simply neutralized, but they can be logged, tracked, and patched.

Compared to other solutions that defend against VTable hijacking, VTPin exhibits certain characteristics that make it suitable for practical and instant deployment in production software. First, VTPin protects binaries, directly and transparently, without requiring source compilation or binary rewriting. Second, VTPin is not an allocator replacement, and thus it does not interfere with the allocation strategies and policies of the protected program; it intervenes in the deallocation process only when a virtual object is to be freed for preserving the VTable pointer. Third, VTPin is fast; Mozilla Firefox, protected with VTPin, experiences an average overhead of 1%–4.1% when running popular browser benchmarks.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Control-flow hijacking; use-after-free; VTable protection

ACSAC '16, December 05–09, 2016, Los Angeles, CA, USA

ISBN 978-1-4503-4771-6/16/12...$15.00

DOI: http://dx.doi.org/10.1145/2991079.2991121

1. INTRODUCTION

The recent advances in software hardening have undoubtedly made exploitation a challenging craft [45]. Yet, despite the plethora of defenses in place [37], attackers still find ways to compromise essential commodity software, like web browsers [5, 6]. Modern exploits are highly sophisticated and typically leverage a variety of different vulnerabilities to bypass established protections, such as address space layout randomization (ASLR) [42], non-executable memory [12], and sandboxing [18]. To this end, temporal safety errors, and particularly use-after-free vulnerabilities, are becoming a key component of the attackers' arsenal [1, 2, 7, 8]. Interestingly, exploiting use-after-free bugs does not require corrupting memory; instead, an attacker merely needs to utilize dangling pointers, still accessible by a process, for hijacking the control flow.

Temporal safety violations are extremely effective when (ab)used for compromising large C++ programs. Virtual objects contain (at least) one pointer towards a Virtual Table (VTable), which further contains (function) pointers to the implementation of the methods associated with the respective object(s). An attacker can *hijack* the VTable by forcing the vulnerable program to carefully allocate memory with attacker-controlled data; assuming there are still dangling pointers that (now) point to the hijacked VTable, the control flow of the program can be redirected according to the needs of the attacker. Notice that VTable hijacking, through use-after-free, can be combined with other attack vectors for delivering the end-to-end exploit. In fact, in recent Pwn2Own security contests, *all* major web browsers were compromised using exploits that contain a step where VTable hijacking was the *key* attack vector [5, 6].

In this paper, we propose VTPin for protecting software against VTable hijacking in the *least* intrusive way. VTPin works directly with C++ binaries that provide Run-time Type Information (RTTI), does not rely on complex binary analysis or rewriting (often hindering practical deployment [51]), does not interfere with the strategies and policies imposed by the allocator of the protected program, and exhibits low overhead (1%–4.1%). VTPin *pins* all freed VTable pointers on a safe VTable by instrumenting every `free` call of a running program. For every `free`, VTPin quickly identifies if it is associated with a virtual object; in case it is, VTPin handles the deallocation accordingly, otherwise the deallocation is forwarded to the program's allocator. VTPin deallocates all space allocated by the object, but preserves its VTable pointers. Additionally, the value of the contained VTable pointer(s) is replaced with the address of a special VTable that VTPin controls. Any dangling pointer, if triggered, can only invoke a method provided by the corresponding safe object. Subsequently, all virtual-method calls due to dangling pointers are not simply prevented, but can also be logged, tracked, and patched.

VTPin follows two strategies for pinning freed virtual objects. If the memory allocator provides reallocation of memory with particular placement, such as the standard GNU allocator, then VTPin pins just the VTable pointer(s) and *frees* the rest of the object. Otherwise, for slab allocators that do not support reallocation with placement, or for objects with multiple VTable pointers, VTPin pins all the VTable pointers and *maintains* all data associated with the freed virtual object. Notice that VTPin handles only the deallocation of objects; all other memory operations, including the allocation of virtual objects, are outsourced to the default allocator of the protected program. VTPin pins only virtual objects, and in most cases only a single pointer (*i.e.,* 8 bytes) survives. Hence, the induced memory overhead, as we demonstrate with extensive measurements that stress the memory footprint of Mozilla Firefox and Chromium, is low. In addition, VTPin employs a conservative garbage collector for periodically removing all pinned objects and reclaiming back their occupied memory.

Scope. VTPin protects VTable pointers, in C++ binaries, from being hijacked through use-after-free vulnerabilities. Although VTPin focuses only on a very specific class of attacks, we stress that VTable hijacking via use-after-free is a popular and effective exploitation vector—as demonstrated by recent security contests [1, 2, 7, 8]. Moreover, by limiting its scope to a specific, yet important, class of attacks, VTPin is able to not only complicate such attacks, but solve the problem entirely and in a practical way (*i.e.,* by just pre-loading a given binary with minimal impact on its execution). In Section 2, we show that more general binary-level solutions do exist, but they yield high overhead, intrusive deployment, or sufficient leeway for attacks, all factors which ultimately hinder their practical adoption in real-world scenarios.

Contributions. This paper makes the following contributions:

1. We design, implement, and evaluate VTPin: a system to protect VTable pointers from exploitation through use-after-free vulnerabilities. VTPin does not require access to the source code of the protected program, is not based on binary analysis or rewriting, is highly portable, and does not interfere with the semantics and policies used by standard allocators.
2. We evaluate VTPin with the C++ programs of SPEC CPU2006, Chromium, and Mozilla Firefox. Mozilla Firefox over VTPin experiences an average runtime overhead of 1%–4.1% when executing popular browser benchmarks.

2. BACKGROUND

2.1 VTable Hijacking

Use-after-free vulnerabilities are core assets in modern software exploitation. As they do not manifest by writing past memory bounds or overwriting the process' data structures (*i.e.,* memory corruption), existing countermeasures for protecting critical data, such as return addresses stored at the process stack, fail to protect against use-after-free abuses. Essentially, the exploitation of use-after-free vulnerabilities is based on performing a series of steps that can hardly be characterized as fraudulent. The key concept of use-after-free exploitation is that an adversary can leverage pointers that (still) point to deallocated memory. Once a chunk of memory is freed the system can reuse the released memory region as needed. If the attacker can place her data in the freed area, and cause the dereference of a pointer that still points to this space, then she can tamper-with the dereferenced data, and, consequently the data flow of the program. Although, any (dangling) pointer can be abused in this way, in the context of C++ programs, use-after-free vulnerabilities are often used to *hijack* VTable pointers. These pointers are abundant and they are used to trigger indirect branches at runtime; therefore they are extremely useful for diverting the control flow of a running process in arbitrary ways.

```
1  class Parent {
2  public:
3      virtual void talk();
4  };
5
6  class Boy: public Parent {
7  public:
8      void talk();
9  };
10
11 class Girl: public Parent {
12 public:
13     void talk();
14 };
15
16 int main(int argc, char *argv[]) {
17     Parent *p1, *p2;
18     ...
19     input == true ? p1 = new Boy() : p1 = new Girl();
20     p1->talk();
21     p2 = p1;
22     delete p1; /* Destructors of Boy/Parent called */
23     /* p2 is now dangling */
24     ...
25     /* use-after-free trigger */
26     p2->talk();
27     return 1;
28 }
```

Figure 1: Example program to demonstrate how use-after-free vulnerabilities can be (ab)used for VTable hijacking. In line 17 two pointers of type `Parent` are declared, namely `p1` and `p2`. We assume that `input` is `true`, and `p1` is instantiated as `Boy`. At line 21, `p2` and `p1` point to the same location, and at line 22 `p1` is deleted. The destructor of Boy and then Parent is called and then all space occupied by the `Boy` instance is released, but `p2` still points to the location that `p1` was pointing at. If `p2` is accessed (line 26) the program behavior is undefined.

Figure 1 illustrates the mechanics of a *VTable hijacking attack*. The program contains three virtual class definitions: `Parent` (base class, lines 1–4), `Boy` (inherits from `Parent`, lines 6–9), and `Girl` (inherits from `Parent`, lines 11–14). In line 17, two pointers of type `Parent` are declared, namely `p1` and `p2`. Depending on the value of `input`, which we assume to be influenced by user input, pointer `p1` can be either instantiated as `Boy` or `Girl` (line 19). Assuming that `input` is indeed `true`, `p1` will be instantiated as `Boy`. Hence, when the virtual method `talk` is invoked (line 20), the particular implementation, as provided by `Boy`, will be executed. The compiler emits the relevant code for performing this resolution (*i.e.,* calling the correct virtual method, depending on the object type); most compilers, implement this feature by using VTables.

In Figure 2(a), we illustrate the memory layout of the object that `p1` points at. The first 8 bytes are occupied by `vfPtr`, a pointer that contains the address of the VTable provided by `Boy`, followed by class data (*i.e.,* the internal variables of `Boy`). Notice that the VTable itself (of `Boy` class) is stored in read-only memory and it further contains pointers for resolving the implementations of the virtual methods provided by both `Boy` and `Parent`. Embedding a pointer that targets a VTable at the beginning of a virtual object is critical to resolve the method to be called at runtime. Unfortunately, the `vfPtr` pointer (Figure 2(a)) cannot be stored in read-only memory, since object instances are allocated on the stack, heap, or the global data section(s)—therefore all VTable pointers (but not the actual VTables) reside in writable memory.

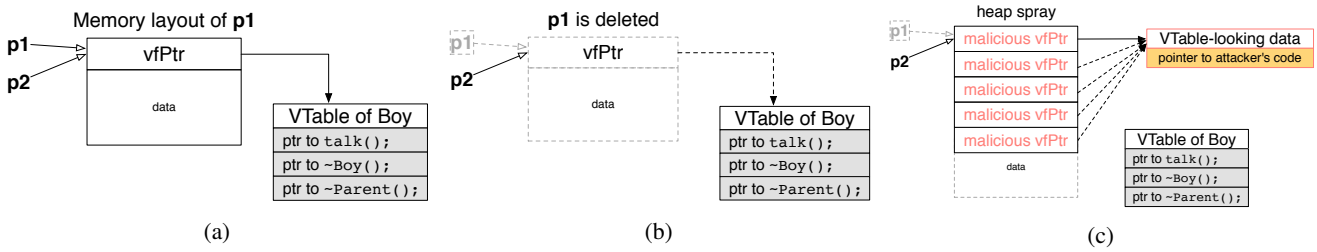

Figure 2: VTable hijacking mechanics. For the example program of Figure 1, we illustrate, in (a), the memory layout of the object that `p1` points to. The first 8 bytes are occupied by `vfPtr`, which points to the VTable provided by `Boy`, followed by the respective class data. The VTable of `Boy`, stored in read-only memory, contains pointers to the implementation of the methods `Boy` supports. In (b), we illustrate what happens if `p1` is deleted, and in (c), how the attacker can take advantage of the dangling pointer (`p2`). An adversary can *spray* memory with the contents of a phony `vfPtr` that points to attacker-controlled data. Such data can resemble a valid VTable: *i.e.,* arranged so that if dereferenced through a dangling pointer (Figure 1, line 26), then a forged *method* will be executed (*e.g.,* `mprotect`), instead of the one intended by the original program.

Protection	Source	Binary	Modified Allocator	Unmodified Allocator
Control-Flow Integrity				
BinCFI [61], CCFIR [60], PathArmor [54], TypeArmor [55]		✓		✓
VTable protection				
VTV [53], SafeDispatch [32]	✓			
vfGuard [43], VTint [59], T-VIP [24]		✓		
Use-after-free protection				
Undangle [15], FreeSentry [57], DANGNULL [33]	✓			
Conservative GC [14]		✓	✓	
Memory analyzers				
Purify [28], Valgrind [39]		✓		✓
Secure allocators				
Cling [11], DieHard [13, 40], CETS [38]			✓	
VTPin		✓		✓

Table 1: Existing mitigation mechanisms. Compared to competing solutions, VTPin does not require access to the source code of the protected program, and does not interfere with the strategies and policies of the allocator used.

Going back to Figure 1, in line 21, `p2 == p1`. Essentially, at this point the memory layout is the one presented in Figure 2(a): both pointers `p1` and `p2` point to the `vfPtr`, which points to the VTable provided by `Boy`. It line 22, `p1` is deleted. Technically, the destructor of `Parent` and `Boy` is called (in this exact order), and the space occupied by the `Boy` instance is marked as free. This space can be reused for future allocations, and, depending on the heap allocator of the system, the contents of the area can be zeroed or left as is. Finally, `p2` still points to the location that `p1` was pointing at: *i.e.,* at the `vfPtr` provided by `Boy`. At this point, the memory layout is as depicted in Figure 2(b) and `p2` is a dangling pointer. If `p2` is accessed, like for example in line 26 of Figure 1, where `talk` is called, the program behavior is undefined. There are basically three possible scenarios: (a) the program crashes, assuming the freed memory is zeroed; (b) the `talk` method of `Boy` is called, assuming the freed memory has not been touched by the heap allocator; or (c) arbitrary code gets executed, assuming the freed memory has been deliberately reused.

Scenario (c) is the one of interest for exploitation purposes. As illustrated in Figure 2(c), an adversary may *spray* [50] memory with the contents of a malicious `vfPtr` that points to attacker-controlled data. Such data can be VTable-looking: *i.e.,* arranged so that if dereferenced, a *forged* method will be invoked. Notice, that the attacker does not inject code, since we assume non-executable data [12], but rather memory addresses. For example, she can place the address of `mprotect` at the tampered-with VTable, which, once called, changes the permissions of the attacker-controlled memory and makes it executable. At this point (Figure 2(c)), if `p2` is used (Figure 1, line 26), the attacker can hijack the control flow of the vulnerable process.

2.2 Existing Mitigations

Given their practical relevance, it comes as no surprise that the research community has proposed many defenses to tackle VTable hijacking (and related) attacks. We group all these efforts in the following four categories and summarize our analysis in Table 1.

Control-flow Integrity. Ensuring that a program executes only control flows that are part of its original Control Flow Graph (CFG) is a core concept known as Control-flow Integrity (CFI) [10]. CFI can protect software against arbitrary control-flow hijacking attacks, despite the existence of vulnerabilities of any type (including use-after-free). However, CFI, though being a strong defense, can be hardly realized without approximating the ideal CFG [54, 55, 60, 61]. This approximation has security consequences: practically any coarse-grained CFI scheme can be defeated [17, 19, 26, 27, 48] and even fine-grained schemes are prone to attacks [16, 21, 35, 47].

VTable protection. Proposals for protecting VTable pointers can be either applied to source code [32, 53] or directly at the binary level [24, 43, 59]. In contrast to generic indirect branches targeted by CFI solutions, VTable pointers should be contained in a well defined C++ class hierarchy, which is likely to be unknown if source code is not available [47]. Generic binary-only solutions that significantly raise the bar against advanced code-reuse attacks do exist [55], however their effectiveness in the presence of complex class hierarchies is questionable [35].

Use-after-free protection. Several proposals aim solely at preventing use-after-free vulnerabilities by carefully updating all pointers of a program so that they do not point to memory areas that can be reused [15, 33, 57]. Such approaches are very effective against use-after-free exploitation and typically experience moderate overhead. However, they all require analysis and restructuring of the program source code, thus failing to protect binaries. Alternatively, conservative garbage collectors [14] could be used to mitigate use-after-free vulnerabilities at the binary level [11], but they typically mandate custom allocators and their full-coverage application incurs nontrivial performance and memory impact (*e.g.,* due to the frequent garbage collection cycles required [29], as well as because of accuracy problems [30]).

Memory analyzers. Another approach is based on tracking all memory operations of a program for detecting safety errors [28, 39]. Such tools can accurately detect memory-related bugs, but they incur overheads that prevent them from protecting deployed software. Nevertheless, they are suitable for debugging software that is under development.

Secure allocators. Secure allocators [11, 13, 38, 40] provide drop-in replacements for the standard memory allocator, with allocation strategies that take into account security vulnerabilities. Custom allocators can thwart use-after-free attacks, but, unfortunately, it is common for industrial software to already employ and heavily rely on an embedded allocator for better memory management. For instance, Google Chrome employs `tcmalloc` [34], while Mozilla Firefox uses `jemalloc` [22]. A secure allocator can protect Mozilla Firefox or Google Chrome only if their embedded allocator(s) are disabled (often infeasible in practice).

The type-safe memory reuse strategy of Cling [11] is the closest, in terms of scope, to VTPin. Yet, unlike VTPin, Cling yields important limitations in practice. First and foremost, Cling cannot co-exist with custom allocators [11], while VTPin integrates nicely with them. In addition, Cling attempts to infer allocation wrappers for identifying the type of a given object. However, accurate type identification is challenging at the binary level, especially on C++ binaries with complex design patterns (*e.g.,* the *factory pattern*) [23], which yield not a single, but many levels of allocation wrappers. This results in type over-approximation and ultimately may allow sufficient leeway for attacks.

Summary. Using any of the mitigations in Table 1 for protecting against VTable hijacking attacks has one of the following limitations: (a) source code is needed and the solution does not apply to binaries [15, 32, 33, 38, 53, 57]; (b) the solution applies to binaries [24, 43, 54, 55, 59–61], but it is ineffective [16, 21, 26, 35, 47]; (c) the solution applies to binaries but the memory allocator must be replaced [11, 13, 14, 40]; (d) the solution applies to binaries without replacing the memory allocator, but the overhead is high [28, 39].

3. SYSTEM OVERVIEW

VTPin aims at protecting binary-only software against VTable hijacking, through use-after-free vulnerabilities, by instrumenting programs in the least intrusive way and with low overhead. Our *least intrusive* requirement entails the following:

- No access to source code or debugging symbols.
- No binary analysis, disassembling, or patching.
- No changes to memory allocation strategies (*e.g.,* via drop-in allocator replacements).

The core idea behind VTPin is simple, but the mechanics for realizing such a system can be complicated (see Section 4). Fortunately, our techniques can be implemented in a highly portable way, promoting VTPin to a generic solution for mitigating VTable hijacking through use-after-free vulnerabilities.

VTPin is based on the observation that the majority of use-after-free (ab)uses are capitalized by hijacking VTable pointers through strategic re-allocation of memory. Therefore, instead of resolving and protecting dangling pointers, VTPin ensures that all VTable pointers are always valid, and thus cannot be maliciously (ab)used via future re-allocations. Whenever a virtual object is freed, VTPin handles the deallocation and ensures that the VTable pointer is preserved in a new and safe form. Specifically, VTPin releases the memory occupied by the virtual object, but preserves the space taken by all VTable pointers, and overwrites them so that they point to a (read-only) VTable provided by VTPin itself. Essentially, VTPin *pins* all freed VTable pointers on a *safe* VTable. Dangling pointers clearly survive, but they are effectively neutralized, and, if triggered, they can be accurately detected—this also makes VTPin a practical tool for identifying use-after-free vulnerabilities. If a dangling pointer is triggered when VTPin is in place, the program will not crash, but control will be transferred to a virtual method that the system controls. Therefore, the dangling pointer access can be logged, traced, and eventually patched.

We demonstrate VTPin works in Figure 3, using (as a reference) the vulnerable program presented in Figure 1. Initially, the two pointers, `p1` and `p2`, point to the VTable pointer of a `Boy` instance (Figure 1, line 17), and the memory layout is the one depicted in Figure 3(a). Once `p1` is freed (Figure 1, line 22), as illustrated in Figure 3(b), VTPin takes control. The system frees all space occupied by the `Boy` instance, but preserves the `vfPtr` pointer. Next, it replaces the value of `vfPtr` with the address of a safe object controlled by VTPin itself. An adversary may further spray memory with forged pointers, but `vfPtr` cannot be hijacked, as shown in Figure 3(c). If a method is invoked through the dangling pointer `p2` (Figure 1, line 26), a safe method of the special VTPin virtual object will be executed, and the call will be contained and logged.

The protection offered by VTPin is not provided without a cost. Preserving all VTable pointers introduces memory overhead. If the protected program uses the system allocator, then for each deallocated virtual object only the VTable pointer (`vfPtr`; 8 bytes) survives. Otherwise, if a custom, slab-like allocator is used (*e.g.,* `jemalloc`, `tcmalloc`), virtual objects are preserved in whole. As we demonstrate later in Section 5, where we precisely estimate the memory overhead of VTPin, virtual objects account for a small fraction of all memory objects, and even complicated benchmark suites that heavily stress the capabilities of web browsers leak only a few MBs of data. Moreover, many applications are based on short-lived process models. For example, popular web servers fork a new process, occasionally, for serving a new client, and shortly afterwards the process terminates. In such cases, VTPin can offer protection with practically no memory overhead. On the other hand, for long-lived processes, VTPin employs a second garbage collection (GC) step, where memory is periodically scanned for potential dangling pointers. VTPin releases any preserved VTable pointer that has no reference to it, and retains all other pointers that are (possibly) still referenced from other memory locations.

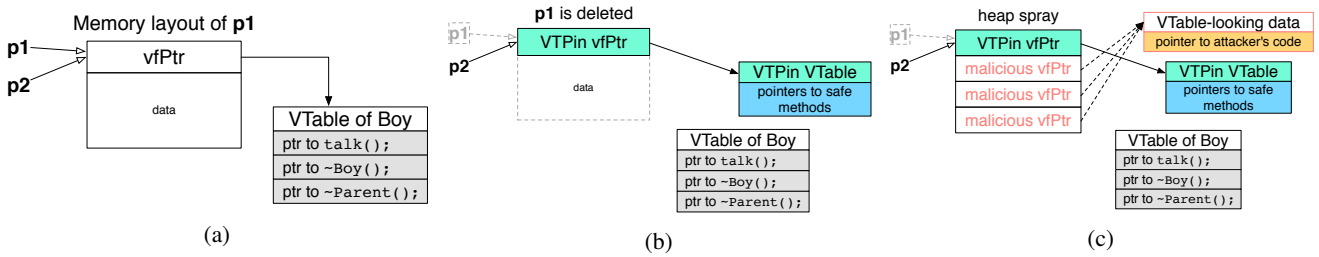

Figure 3: VTable hijacking prevention when VTPin is in place. Initially, in (a), the two pointers, `p1` and `p2`, point to a `Boy` instance (Figure 1, line 17). Once `p1` is freed (Figure 1, line 22), in (b), VTPin frees all space occupied by the `Boy` instance, but preserves the `vfPtr` pointer. (If the memory allocator used does not allow re-allocating the object at the same address, then VTPin will fully preserve the memory used by the object.) Additionally, the system replaces the old `vfPtr` with the address of a safe virtual object controlled by VTPin itself. An adversary may further spray memory with forged pointers, as in (c), but the VTable pointer cannot be hijacked. If a method is invoked through the dangling pointer `p2` (Figure 1, line 26), a safe method of the special VTPin virtual object will be executed, and the call will be contained and logged.

Notice, that VTPin can be extremely conservative at this stage, since the majority of the preserved pointers will not be referenced (*i.e.,* we assume that protected programs do not intentionally contain a vast amount of dangling pointers). This additional GC step is a costly operation. However, we stress that, since it is only used to reclaim *dead* VTable pointers (unlike traditional conservative garbage collectors [14]), it can be carried out infrequently with very low memory and performance impact on the running program. We further elaborate on this aspect in Section 5.5, where we evaluate the time needed to scan the memory of a process and free any remaining non-referenced VTable pointers.

3.1 Virtual Objects

VTPin protects software without requiring access to code (source or binary), or modifying the memory allocator used. To this end, it instruments all `free` calls issued by a program, and intervenes only when a `free` operation is called on a virtual object. Hence, VTPin has to infer if a pointer is allocated to a virtual object or not. In the following, we discuss the basic algorithm for resolving pointers to virtual objects, and in Section 4 we provide the technical details for implementing the algorithm in the VTPin prototype.

Given a pointer, `ptr`, VTPin first checks if `ptr` is valid (*i.e.,* non-null); if it is, then VTPin extracts the first 8 bytes of the memory block pointed by `ptr`, and treats them as a VTable pointer (`vfPtr`). Next, it examines all the read-only memory regions of the running process to check if the address `vfPtr` points to belongs to them. If the address is not found, then it is certain that `ptr` is not associated with a virtual object, since VTables are stored in read-only memory regions. Examining every read-only memory range is easy and fast, as all shared objects are linearly mapped and ASLR, or fine-grained randomization [25, 31, 41, 56], has no impact.

If `vfPtr` is not found in read-only memory, VTPin outsources the deallocation operation to the underlying program allocator. Otherwise, `ptr` is consider *potentially* associated with a virtual object. To ultimately infer whether `ptr` is indeed associated with a virtual object, VTPin uses the RTTI mechanism provided by C++ [4]. Specifically, it assumes the `ptr` is pointing to a virtual object, and traverses memory to discover its class type. If the class type is successfully found during this traversal, then `ptr` is *certainly* associated with a virtual object. Notice, that there is high probability, during a memory traversal, for VTPin to touch unmapped memory and cause a fault. Such faults are handled by VTPin without impacting the running program, as we further discuss in Section 4.

3.2 Multiple Inheritance

C++ supports multiple inheritance, and the ABI [4] suggests how VTables of virtual objects that inherit from several classes should be laid out in memory. In particular, two different layouts are possible. The first layout applies to cases where the class of the object instance inherits from none or a single class. Notice that whether the base class also inherits from other classes does not affect the layout. In this case, the object contains a single VTable pointer. This pointer points to a "merged" VTable that has been created by merging the VTable of the derived class with the VTable of the base class. For this layout, VTPin can easily pin the single VTable pointer and release the rest of the virtual object. However, in cases of multiple inheritance, where a class inherits from two, or more, classes directly, the layout is fairly different. The object now contains as many virtual pointers as the number of classes from which it inherits. For each class there is a VTable pointer, at a fixed offset from the start of the object, pointing to the corresponding VTable of that class. In this case, VTPin cannot just pin several VTable pointers and release the rest of the object. Being overly conservative, it pins all the VTable pointers and preserves the memory of the object. In Section 4.5, we discuss how to distinguish virtual objects with multiple from single inheritance, and in Section 5.2 we evaluate the frequency of multiple inheritance objects.

4. IMPLEMENTATION

Our prototype implementation of VTPin is written in C/C++, consists of ∼2000 LOC, and targets (C++) Linux binaries on x86-64 systems—VTPin itself is compiled with GCC (`g++`). As stressed in Section 3, VTPin is highly portable. In this section, not only we provide the implementation details of VTPin, but we also expand on various technical aspects that need adjustment when VTPin is meant to offer protection to binaries on other platforms.

4.1 Portability Requirements

Implementing VTPin on any platform is possible as long as the following requirements are met:

1. *Hooking* `free`. VTPin needs to intercept each `free` call for pinning VTable pointers. In particular, VTPin must be able to hook `free` directly, if the system allocator is in place, or place hooks in a custom allocator otherwise [3].

2. *Allocation with placement.* VTPin preserves VTable pointers, while deallocating the rest of a virtual object. To enforce

this property, VTPin needs to manage memory in a very precise way: *i.e.,* reserve 8 bytes at the exact address that the original virtual object was allocated at. The implemented prototype is based on the standard `glibc` allocator, which offers in-place `realloc` functionalities at a specific memory address. Once `realloc` is provided with a smaller size, it simply keeps all contents up to the new size and discards the rest [9]. Other allocators [22, 34] implement `realloc` in a different way; the reallocated memory block is always placed at an address that is different from the one that it was initially allocated at. If no such `realloc` functionality is available, VTPin resorts to keeping the entire virtual object in memory, overwriting all the contained VTable pointers, and garbage-collecting the object at a later point in time.

3. *RTTI support.* As C++ offers support for Run-time Type Information, it is possible to infer the type of a particular pointer at runtime. Compilers are free to implement RTTI differently [46,52], but they all export the same API (`type_info` and similar) for handling RTTI requests [4]. VTPin is based on the RTTI implementation of GCC and LLVM.

4. *Handling invalid memory accesses.* VTPin uses the RTTI functionality provided by the compiler for inferring, at runtime, if a pointer is associated with a virtual object (Section 3.1). However, accessing the RTTI properties of a non-virtual object, by traversing memory, may result in touching unmapped memory. Therefore, VTPin needs to either handle and recover from a `SIGSEGV` signal or probe memory without causing a `segfault`. Most platforms support handling `SIGSEGV` in user space. For example, in Microsoft Windows programs can use Structured Exception Handling [36], and in Linux programs compiled with GCC can transparently convert signals to C++ exceptions (`-fnon-call-exceptions`). Probing memory without causing a `segfault` is also possible on Linux (and other Unix-like OSes) via system calls like `mincore`, which fail when the respective memory address (*i.e.,* the first argument of `mincore`) is not mapped. VTPin implements both techniques.

4.2 Basic Components

VTPin is implemented as a shared library that can be preloaded using `LD_PRELOAD` on Linux (`DYLD_INSERT_LIBRARIES` on OSX, *etc.*) for instrumenting all `free` calls of a running binary. The system consists of three components: (a) a memory map that contains all read-only memory regions; (b) a safe VTable where all virtual objects are pinned when deallocated; and (c) a secondary thread that scans memory at configurable intervals and reclaims *pinned* pointers that no longer have dangling references.

4.2.1 Memory Map

VTPin maintains a memory map with all allocated and read-only memory pages. As compilers place VTables on read-only pages, VTable pointers should point to such pages. This property is used for quickly checking if a freed pointer is associated with a virtual object (Section 3.1). VTPin collects these pages during bootstrap by reading the `/proc/self/maps` file of the running binary, and, successively, by hooking `dlopen` for updating the map with shared objects that are mapped at runtime. All read-only memory is spread in the virtual address space in non-overlapping regions, which can be sorted. Checking if an address belongs to these regions can be done efficiently by maintaining a splay tree [20].

4.2.2 Safe VTable

VTPin, early at its initialization phase, allocates a special virtual object, which contains the implementation of several virtual methods. Every time a virtual object is deallocated, its VTable pointer is preserved and its value is swapped to point to the VTable of the safe object (Figure 3). Any dangling pointer related to the deallocated object, if triggered, invokes a method implemented by the safe object, and thus exploitation can be easily contained. Each method of the safe object can be arbitrarily implemented according to the needs of the administrator (or developer). For the prototype discussed here, each method logs the address of the instance that called the method. Program execution should not be terminated, since the dangling pointer is no longer dangerous. Hence, use-after-free attacks that aim at a denial of service, which otherwise could be still dangerous—for example when (dangling) pointers are nullified after deallocation—are also alleviated.

4.2.3 Garbage collection

VTPin pins the VTable pointers of virtual objects once they are deallocated, and adds their addresses to a special data structure, called `VTPit`. Next, it periodically scans the stack, heap, and global data section(s) of the process, and checks for addresses contained in `VTPit`. If such addresses exist, then (possible) dangling pointers exist as well, and the object is marked accordingly (in `VTPit`). After the scan completes, the `VTPit` data structure is fully examined again, and every object that was not marked as possibly having a dangling pointer, is finally freed. This process can be parallelized, aggressively, by scanning different memory regions with different CPU cores. The regions to be scanned are determined by parsing `/proc/self/maps`. By default, VTPin opts for a simple nonparallel GC strategy, triggering a complete memory sweep every time the overall size of memory occupied by pinned objects exceeds a particular threshold (100 MB by default). However, as we discuss in Section 5.5, additional configurations are possible.

4.3 Virtual Object Resolution

A high-level overview of how VTPin works is sketched in Figure 4. Given a pointer to be freed, VTPin infers if the pointer is associated with a virtual object using the algorithm outlined in Section 3.1. First, the memory map is checked to see if the (to-be-pinned) VTable pointer points to a read-only region. Then, RTTI is used for verifying that the pointer is indeed associated with a virtual object. VTPin incorporates two methods for RTTI resolution.

The first method employs `SIGSEGV` signals. The pointer for the expected RTTI (`type_info`) structure is resolved based on the Itanium ABI that both LLVM and GCC implement [4]. Figure 5 shows the structure of the RTTI information in memory for a given pointer (`ptr`). In case there is no RTTI information associated with the given pointer, this operation fails by emitting a `SIGSEGV` signal for the calling thread. As VTPin is compiled using GCC (`g++`), `-fnon-call-exceptions` is used in compilation for translating signals to C++ exceptions. RTTI resolution is performed in a `try/catch` block, and, in case unmapped memory is accessed, a custom exception handler is called for handling `SIGSEGV`. Notice, that once the exception is raised, and once the handler is finished, an additional `free` is performed for deallocating the object that handled the exception. This additional `free` is carefully issued by VTPin to avoid infinite loops.

The second method tries to validate the `type_info` structure by probing the memory locations, shown in Figure 5, one by one. If at any step the memory is unmapped, or if a VTable pointer points to writable memory (even though it should point to read-only memory), it can be deduced that the RTTI structure is invalid and the

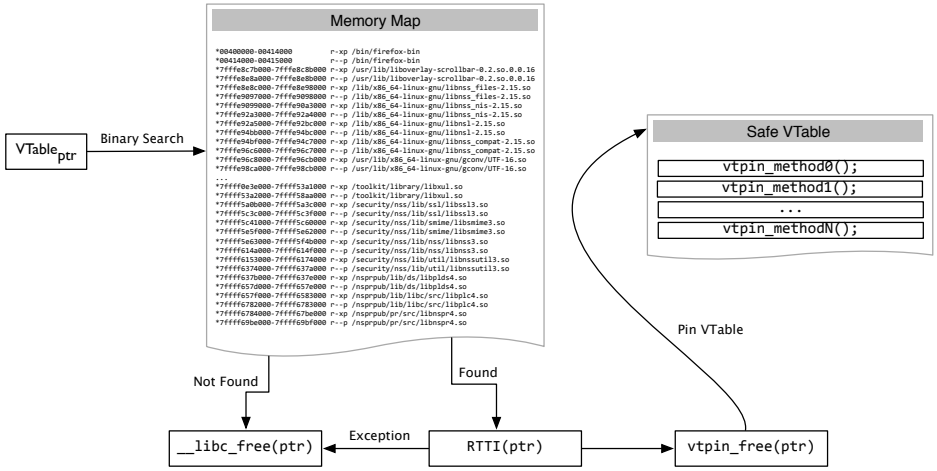

Figure 4: VTPin prototype. Given a pointer to be freed, VTPin infers whether the pointer is associated with a virtual object or not (Section 3.1). First, the memory map is searched for locating a possible VTable, and then RTTI resolution is used. Once VTPin identifies that a pointer is associated with a virtual object, it *pins* the VTable pointer of that object to a safe VTable. VTPin uses `realloc` for shrinking the object to 8 bytes and discarding the rest of it. Finally, VTPin copies the VTable address of the safe object to the preserved 8 bytes containing the VTable pointer. VTable hijacking through dangling pointers is not possible anymore, since triggering such a pointer will eventually call a virtual method implemented by the safe object.

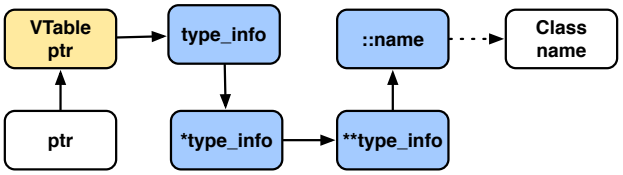

Figure 5: Memory layout expected during RTTI resolution. Each shaded block denotes a memory location probed with the `mincore` syscall to check whether it is mapped, or not, before accessing it. `ptr` is the pointer intercepted by our `free` hook.

original object is not virtual. To probe the memory without causing a `SIGSEGV`, VTPin uses the `mincore` system call, which fails (*i.e.,* `ENOMEM` in Linux) when the memory is unmapped. By default, VTPin uses this second method to probe memory locations, since, on preliminary tests, `SIGSEGV` handling exhibited worse performance (*i.e.,* extra 2–9%). Note that while `mincore`-based probing is fairly efficient, we believe that platform-specific, user-mode-only probing methods may further improve performance. For example, the recent Intel transactional synchronization extensions (TSX) allow a user thread to efficiently infer unmapped memory locations from transaction aborts without involving the kernel [44]. However, due to its portability limitations (TSX is only available in recent Intel CPUs and often disabled due to implementation bugs), we decided against incorporating such a method in our current prototype.

4.4 VTable Pinning

Once VTPin identifies that a pointer to be freed is associated with a virtual object, it proceeds and *pins* the respective VTable pointer(s) to a safe VTable (Figure 4). In cases of single inheriting objects, VTPin uses `realloc` as provided by `glibc` for shrinking the object to 8 bytes. This action preserves (at the original memory address) the first 8 bytes, where the (still valid) VTable pointer is stored, and deallocates the rest of the object [9]. Finally, VTPin copies the VTable address of the safe object to the preserved 8 bytes of the object. This action essentially *pins* the old VTable pointer to a safe VTable controlled by VTPin. VTable hijacking through dangling pointers is not possible anymore, as triggering such a pointer will invoke a virtual method implemented by the safe object.

Notice that not all memory allocators follow the semantics of `glibc` when implementing `realloc`. Many implementations [22, 34] return a fresh pointer located at a different memory address, no matter if the new object occupies less or more space than the originally-allocated one. VTPin favors transparency and does not require changes in the semantics followed by the memory allocator. VTPin needs only a custom function for reallocating an object at a particular memory address. This function can be any private function (not `realloc`), which is meant to be used only by VTPin for offering protection to the running process. Thus, all policies related to the memory allocation of the protected program are preserved. If by no means such a function can be made available, then VTPin resorts to preserving the entire virtual object instead of freeing it—and releasing it at a later point in time, once no dangling pointers to it are found by the memory-scan thread (Section 4.2.3). The entire virtual object is also preserved in cases of multiple inheritance. Since the objects in those cases contain more than one virtual pointer, VTPin cannot reallocate memory at their specific addresses (`malloc` does not provide that functionality).

4.5 Handling Multiple Inheritance

VTPin needs to determine whether to preserve only the VTable pointer or the entire object. The entire object is preserved in cases of multiple inheritance, as several VTable pointers are contained in it. VTPin gets the size of the object with `malloc_usable_size`[1], and then traverses the memory occupied by the object to find VTable pointers that point to a nearby (in memory) VTable compared to the

[1] `malloc_usable_size` is a non-standard interface provided by all allocators tested with VTPin (*i.e.,* `ptmalloc2/glibc`, `tcmalloc`, `jemalloc`) for introspection purposes. However, notice that `free` does not require a size argument; every allocator has a way of knowing the *exact* size of the memory block that is to be reclaimed.

first VTable pointer of the object. In case VTPin finds more than one VTable pointer contained in the virtual object, then the class is considered to make use of multiple inheritance and the complete object is preserved. Notice that all contained VTable pointers are pinned and therefore neutralized.

5. EVALUATION

In this section, we assess the security effectiveness of VTPin, and evaluate its associated memory and runtime overheads. We test our prototype with the C++ benchmarks (all but `471.omnetpp` due to limitations of our prototype) from the SPEC CPU2006 suite, Mozilla Firefox v47, and Chromium v50. All experiments were carried out on a host running Ubuntu Linux v14.04 (64-bit), armed with a 2GHz quad-core Intel Core i5-3320M CPU and 8GB RAM.

5.1 Effectiveness

We evaluated the effectiveness of VTPin by employing the same three (publicly available) exploits used by the authors of VTrust [58]. All three exploits target Firefox and rely on use-after-free vulnerabilities and VTable hijacking. The tests were performed on an Ubuntu Linux v14.04 (64-bit) virtual machine, with the latest version of Metasploit framework running on the host machine. Each corresponding Firefox version was compiled with GCC and the `'ac_add_options --enable-cpp-rtti'` flag was added to the default configuration. The original exploits target Windows XP/SP3, so we had to port them to Linux; they match the following CVEs: CVE-2013-1690 (Firefox v17.0), CVE-2011-0065 (Firefox v3.5) and CVE-2013-0753 (Firefox v17.0.1). All exploits successfully triggered the respective vulnerabilities, which we cross-checked by inspecting their stacktraces, and they were all thwarted by VTPin.

5.2 Deallocation Calls

Table 2 summarizes the distribution of the instrumented `free` calls, as recorded by VTPin, for the C++ benchmarks of the SPEC CPU2006 suite, as well as for Mozilla Firefox and Chromium. For SPEC CPU2006 we run the respective benchmarks until completion. Each recorded `free` call may correspond to deallocating: (a) a *definitely* non-virtual object; or (b) a *possibly* virtual object. The former means that the pointer about to be freed does not point to a VTable—in practice it points to a writeable memory block (recall that VTables are always kept in read-only pages). For the latter scenario(s), we further resolve object types using RTTI. If the resolution is not successful, we surmise that the deallocation corresponds to a non-virtual object as well (as mentioned in Section 3.1).

Given the above, Table 2 should be interpreted as follows (using `xalanc` as an example). After running `xalanc` to completion, VTPin recorded a total of 126,929,346 `free` invocations. 1,594 calls were not handled by VTPin, since the system had not bootstrapped at that time; these calls are associated with loading long-lived dynamic shared objects during startup. Furthermore, there were 81,211 `free` invocations with `null` as argument (in that case VTPin returns immediately), and 126,780,565 calls regarding non-virtual objects—126,780,487 calls with a `vfPtr` that does not point to a VTable (*i.e.,* scenario (a) above), and 78 calls (with a seemingly valid `vfPtr`) for which RTTI resolution failed. All in all, VTPin resolved 65,898 virtual objects, while 27,481 (41.7%) of them were virtual objects belonging to a class with multiple inheritance.

Several observations can be made based on Table 2. First, the deallocation of virtual objects is only a small percentage (in most cases less than 1%) of the total deallocations. The highest percentage is recorded when Firefox is running the SunSpider benchmark, where the deallocations of virtual objects account for $\sim$14% of all deallocations. Second, most deallocations that are not related with a virtual object are cleared fast, without relying on RTTI resolution; VTPin quickly infers that the pointer to be freed is not associated with any VTable (column *invalid vfPtr*). Lastly, whenever VTPin decides to take the slow path, *i.e.,* to perform a resolution based on RTTI, the decision is correct in most of the cases; only a small fraction of all RTTI resolutions fail (column *RTTI failed*).

5.3 Memory Overhead

VTPin preserves VTable pointers or (whole) virtual objects. Although the memory occupied by pinned objects is periodically reclaimed, it is interesting to explore its volume. For this, we hook `malloc` and record the size of the requested memory block, and the pointer returned, on each invocation. In parallel, we trace all the `free` calls. The pointer that is freed is associated to the respective `malloc` invocation, giving us the actual size for each virtual object detected by VTPin. For each benchmark, we maintain the cumulative allocation size, increased/decreased by `S` when an allocation/deallocation of size `S` takes place. We then calculate the *maximum* cumulative allocation size, as observed during the lifecycle of each benchmark, at the end of a given run. In parallel, we account for any operations that affect virtual objects, and measure the extra memory required by VTPin when using a slab-like allocator or the standard `glibc` allocator.

Table 3 summarizes our findings. For brevity, we focus our browser memory analysis on Firefox, which maintains many more virtual objects that yield significantly higher pressure on the memory footprint. Notice that, when the `glibc` allocator is used, VTPin exhibits negligible memory overhead across all the benchmarks. When a slab allocator is used (and VTPin needs to retain entire virtual objects), the memory overhead is, as expected, more prominent. Nonetheless, only two benchmarks, Peacekeeper and `xalanc`, force VTPin's default configuration to garbage collect dead objects after hitting the 100 MB threshold. Our worst-case memory overhead with a slab allocator (less than 30%) is comparable to that of existing solutions. For example, Cling [11] reports a memory overhead of 40% for `xalanc`. Again, if the `glibc` allocator is in place, our worst-case memory overhead drops to less than 20%.

5.4 Performance Overhead

In Table 4, we summarize the results of the runtime overhead imposed by VTPin. Again, the SPEC CPU2006 benchmarks run to completion and each experiment was repeated 10 times. For Firefox and Chromium, we use standard JavaScript and HTML5 benchmarks, since they are more accurate for measuring the performance of web browsers. In the following, we discuss each benchmark category separately.

SPEC CPU2006. `namd` and `soplex` do not (heavily) use virtual objects (Table 2), and thus the overhead imposed by VTPin is practically zero. `dealII` involves virtual objects, but the exhibited slowdown is negligible ($\sim$1.8%). Finally, `xalanc`, an XML-based benchmark, which massively allocates and deallocates memory incurs a relatively higher overhead (4.9%).

Mozilla Firefox/Chromium. Mozilla Firefox and Chromium were both tested with all popular browser benchmarks: *i.e.,* SunSpider, Kraken, Peacekeeper, and Octane. Notice that each benchmark reports scores based on different metrics. For SunSpider and Kraken, which mainly stress JavaScript operations, VTPin imposes an overhead of 4.1% and 1.2%, respectively. The rest of the benchmarks incur less overhead, practically less than 1%. Similar overheads are reported for Chromium, as well. We also observed our default garbage collection strategy to yield a very low impact on our benchmarks. Even for Firefox/Peacekeeper, which reported the highest number of garbage collection sweeps (*i.e.,*, 3), we observed a GC-

Application	Calls	Unhandled	null	Non-virtual (invalid vfPtr)	Non-virtual (RTTI failed)	Virtual (all)	Virtual (multi. inheritance)
483.xalanc	126,929,346	1,594	81,211	126,780,487	78	65,898	27.481 (41.70%)
447.dealII	12,173,483	1,594	857,082	11,313,200	11	1,596	1263 (71.90%)
444.namd	2,944	1,594	28	1,322	0	0	0
450.soplex/1	2,816	1,594	6	1,213	0	3	0
450.soplex/2	191,361	1,594	12	189,751	0	4	0
453.povray	2,569,867	1,594	151,060	2,415,154	1,990	69	0
462.libquantum	2063	178	0	0	0	0	0
473.astar	1116981	181	0	0	0	0	0
Firefox/SunSpider	2,102,173	1,747	226,331	1,562,756	7,613	303,726	8,462 (2.78%)
Firefox/Kraken	1,708,089	1,600	157,846	1,329,169	7,673	211,507	5,939 (2.80%)
Firefox/Peacekeeper	33,478,893	178	15,820,851	14,614,465	119,131	2,924,268	17,752 (0.60%)
Firefox/Octane	2,328,826	178	607,237	1,591,611	7,082	122,718	4,353 (3.54%)
Chromium/SunSpider	473,217	16	30,603	437,928	3,804	867	308 (35.5%)
Chromium/Kraken	340,390	15	25,222	310,511	3,535	1,107	468 (42.2%)
Chromium/Peacekeeper	467,963	15	37,848	425,051	4,208	841	358 (42.5%)
Chromium/Octane	270,036	15	24,731	238,534	4,759	1,997	828 (41.4%)

Table 2: Distribution of `free` calls for the C++ benchmarks of the SPEC CPU2006 suite, as well as for Mozilla Firefox and Chromium.

Benchmark	Max memory	VTPin/norealloc (%)	VTPin/realloc (%)
Firefox			
SunSpider	131,309 KB	38,616 KB (29.4%)	3,462 KB (2.63%)
Octane	321,166 KB	16,740 KB (5.21%)	1,549 KB (0.48%)
Peacekeeper	624,546 KB	102,400 KB (16.4%)	21,632 KB (3.46%)
Kraken	1,240,534 KB	28,674 KB (2.31%)	2,559 KB (0.20%)
SPEC CPU2006			
483.xalanc	373,889 KB	102,400 KB (27.4%)	68,350 KB (18.2%)
447.dealII	107,035 KB	372 KB (0.34%)	272 KB (0.25%)
450.soplex/1	16,231 KB	496 B (0.00%)	24 B (0.00%)
450.soplex/2	15,758 KB	608 B (0.00%)	32 B (0.00%)
453.povray	3,278 KB	12 KB (0.36%)	552 B (0.01%)

Table 3: Memory footprint of virtual objects for Firefox (when running several benchmarks) and SPEC CPU2006 (only benchmarks that contain virtual objects are included). *Max memory* refers to the maximum cumulative allocation size observed during the execution of each benchmark. The *VTPin/norealloc* column (default on Firefox) depicts the amount of memory used by VTPin in absence of adequate `realloc` support from the underlying allocator (*e.g.*,, via a slab allocator). The *VTPin/realloc* column (default on SPEC) lists the amount of memory used by VTPin when adequate `realloc` support is available (*e.g.,* via the `glibc` allocator). Note that VTPin's default configuration bounds memory leakage to 100 MB.

induced overhead impact of only +1.7%. Our results on Firefox and Chromium suggest that VTPin can secure real-world software with negligible (less than 5%) overhead.

5.5 Collecting Pinned VTables

To further investigate our garbage collection performance, we evaluated the time for VTPin to scan memory and collect pinned pointers. For this purpose, Table 5 lists the time needed for scanning the heap (which dominates the scanning time) of Mozilla Firefox for collecting 10K, 100K and 1M pinned pointers. All pinned pointers are stored in one of two C++ `unordered_maps`, as provided by STL, and when the heap scan runs one buffer is freed while the other continues to fill via new `free` calls. This strategy eliminates the need to block other threads while the heap is scanned. Our results show that, even for large heap sizes (of several MBs), such as the one of Mozilla Firefox, the scanning process can complete in a few seconds (up to 10 seconds for scanning 10M of pinned pointers on a single thread). In addition, our results confirm that our garbage collection strategy is heavily parallelizable, with a scan being up to 3 times faster with 4 threads running on different cores compared to the single thread scenario. Since such scan takes place infrequently (every 100 MB of pinned pointers by default), the overall performance of the protected program is not affected in practice. This is also reflected in the performance overhead results presented for our benchmarks earlier.

6. RELATED WORK

Many techniques have been proposed for defending against various exploitation approaches. Control-flow Integrity (CFI) [10] is a generic concept for ensuring that an attacker cannot tamper-with the control flow of a running process. In the case of VTable hijacking, CFI can protect the vulnerable program, as VTable pointers are constrained and thus unable to point-to any foreign VTable introduced at runtime [53]. Alas, CFI requires perfect knowledge of the Control Flow Graph (CFG), and, in practice, can be only realized as an approximation. Specifically, all coarse-grained forms of CFI [60, 61] suffer from inherent limitations and it has been shown that they can be bypassed [26].

Benchmark	Vanilla	VTPin (overhead)
SPEC CPU2006		
483.xalanc	183.42	192.5 (**1.049x**)
447.dealII	23.3	23.73 (**1.018x**)
450.soplex/1	32.06	32.564 (**1.015x**)
450.soplex/2	4.33	4.39 (**1.013x**)
462.libquantum	74.66	75.19 (**1.007x**)
444.namd	43.802	44.859 (**1.024x**)
453.povray	174.51	175.2 (**1.004x**)
473.astar	319.3	321.76 (**1.007x**)
Firefox		
SunSpider	400.3ms ± 2.0%	416.7ms ± 3.3% (**1.041x**)
Kraken	1,653.3 ± 1.1%	1,674.4 ± 1.2% (**1.012x**)
Peacekeeper	2,843 ± 0.1%	2,919 ± 0.4% (**1.027x**)
Octane	18,320 ± 1.5%	18,378 ± 1.9% (**1.003x**)
Chromium		
SunSpider	3,616ms ±3.1 %	3,668ms ± 1.4% (**1.014x**)
Kraken	12,945 ±1.8 %	13,137 ± 1.9% (**1.015x**)
Peacekeeper	166 ±1.4 %	160 ± 0.7% (**1.036x**)
Octane	3,636 ± 1.3%	3,670 ± 1.2% (**1.009x**)

Table 4: Performance overhead of SPEC CPU2006, and Firefox and Chromium when running popular web browser benchmarks. For SunSpider and Kraken, which mainly stress JavaScript operations, VTPin imposes an overhead of 4.1% and 1.2%, respectively on Firefox. The overhead imposed to benchmarks that do not heavily use virtual objects is practically zero. On the other hand, `xalanc`, an XML-based benchmark, which massively allocates and deallocates memory has an overhead of 4.9%. Notice, also, that for Firefox/Peacekeeper the overhead includes the garbage-collection phase, which is applied when the amount of pinned virtual objects exceeds the size of 100MB.

Heap size	Objects	GC (1 thread)	GC (4 threads)
64,675 KB	10K	1.30sec	0.60sec
64,675 KB	100K	1.41sec	0.71sec
64,675 KB	1M	1.73sec	0.87sec
128,675 KB	10K	1.59sec	0.68sec
128,675 KB	100K	2.59sec	0.77sec
128,705 KB	1M	2.77sec	1.76sec
512,675 KB	10K	6.17sec	2.85sec
512,675 KB	100K	9.67sec	2.94sec
512,675 KB	1M	10.87sec	5.64sec

Table 5: Time needed for scanning the heap (which dominates the scanning time) of Mozilla Firefox for collecting 10K, 100K, and 1M of pinned pointers, respectively. Since this is easily parallelizable, duration was also measured when running garbage collection on 4 CPU cores.

Since VTables are a primary asset for attackers, the research community has focused on applying narrow-scoped CFI just for protecting VTables [24, 32, 43, 53, 59]. All techniques that work with binaries [24, 43, 59] have been demonstrated imperfect, since recovering correctly all the semantics related to the C++ class hierarchy (without access to source code) is still an open problem [47]. For techniques that work at the source level [32, 53] it is still questionable if they are indeed bullet-proof. Compared to all CFI-based solutions for protecting VTables, VTPin can work directly with binaries, without suffering from problems related to the C++ class hierarchy [47], and can offer a sound solution.

VTPin protects VTables only from use-after-free vulnerabilities. This particular type of vulnerability has been addressed by many studies [15, 33, 57]. However, in contrast to VTPin, for all these proposals access to source code is required. Another option is to provide a custom memory allocator that carefully re-uses memory [11, 13, 38], but this option needs the program allocator to be replaced. On the other hand, VTPin aims at being as transparent as possible, and for that reason it is not offered as an allocator replacement. In fact, VTPin can be freely used in cooperation with any custom allocator, even in combination with allocators that protect against use-after-free bugs, as it handles specially only deallocations associated with virtual objects. For example, Cling [11] protects against use-after-free exploitation, but it is possible for an object of the same type to be allocated at a memory area previously occupied. This particular instance of use-after-free vulnerability can be prevented by VTPin at a low cost, without disabling Cling.

Finally, there are memory analyzers [28, 39, 49], which offer managed memory allocation, but their imposed overhead is dramatically high, making them suitable only for debugging. In contrast, VTPin experiences low overheads: 1%–4.1% when running popular web browser benchmarks, and 0.4%–4.9% when running the SPEC CPU2006 suite.

7. CONCLUSION

In this paper we proposed VTPin: a system for protecting C++ binaries from VTable hijacking. Compared to existing protection mechanisms, VTPin exhibits certain characteristics that make it suitable for practical and instant deployment in production software. First, VTPin protects binaries directly without requiring access to the source code or relying on complex binary analysis and rewriting techniques. Second, VTPin is not an allocator replacement, and as such, it does not interfere with the allocator's strategies and policies; VTPin intervenes in the deallocation process only when a virtual object is to be freed, so as to preserve the VTable pointer. Third, VTPin is fast. Mozilla Firefox experiences an overhead ranging from 1% to 4.1%, on popular browser benchmarks, while the overhead of SPEC CPU2006 ranges from 0.4% to 4.9%.

Availability

Our prototype implementation of VTPin is freely available at: https://github.com/uberspot/VTPin

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported by the European Commission through project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571.

8. REFERENCES

[1] Advanced Exploitation of Mozilla Firefox Use-after-free Vulnerability (MFSA 2012-22). http://www.vupen.com/blog/20120625.Advanced_Exploitation_of_Mozilla_Firefox_UaF_CVE-2012-0469.php.

[2] Advanced Exploitation of Mozilla Firefox Use-After-Free Vulnerability (Pwn2Own 2014). http://www.vupen.com/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php.

[3] Hooking the memory allocator in Firefox. https://glandium.org/blog/?p=2848.

[4] Itanium C++ ABI. https://mentorembedded.github.io/cxx-abi/abi.html.

[5] Pwn2Own 2015: The year every web browser went down. http://www.zdnet.com/article/pwn2own-2015-the-year-every-browser-went-down/.

[6] Pwn2Own 2016: Hackers Earn $460,000 for 21 New Flaws. http://www.securityweek.com/pwn2own-2016-hackers-earn-460000-21-new-flaws.

[7] (Pwn2Own) Adobe Flash Player AS3 ConvolutionFilter Use-After-Free Remote Code Execution Vulnerability. http://www.zerodayinitiative.com/advisories/ZDI-15-134/.

[8] (Pwn2Own) Google Chrome Blink Use-After-Free Remote Code Execution Vulnerability. http://www.zerodayinitiative.com/advisories/ZDI-14-086/.

[9] `realloc()` – GNU C Library. http://bazaar.launchpad.net/~vcs-imports/glibc/master/view/head:/malloc/malloc.c#L4235.

[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proc. of ACM CCS*, pages 340–353, 2005.

[11] P. Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proc. of USENIX SEC*, pages 177–192, 2010.

[12] S. Andersen and V. Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[13] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proc. of ACM PLDI*, pages 158–168, 2006.

[14] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proc. of ACM PLDI*, pages 157–164, 1991.

[15] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *Proc. of ISSTA*, pages 133–143, 2012.

[16] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proc. of USENIX SEC*, pages 161–176, 2015.

[17] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proc. of USENIX SEC*, pages 385–399, 2014.

[18] Chromium OS. Sandbox. https://www.chromium.org/developers/design-documents/sandbox.

[19] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proc. of USENIX SEC*, pages 401–416, 2014.

[20] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proc. of ICSE*, pages 162–171, 2006.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proc. of ACM CCS*, pages 901–913, 2015.

[22] J. Evans. A Scalable Concurrent `malloc(3)` Implementation for FreeBSD. In *Proc. of BSDCan*, 2006.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[24] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proc. of ACSAC*, pages 396–405, 2014.

[25] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proc. of USENIX SEC*, pages 475–490, 2012.

[26] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 575–589, 2014.

[27] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *Proc. of USENIX SEC*, pages 417–432, 2014.

[28] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of USENIX ATC*, pages 125–136, 1992.

[29] M. Hirzel and A. Diwan. On the Type Accuracy of Garbage Collection. In *Proc. of ISMM*, pages 1–11, 2000.

[30] M. Hirzel, A. Diwan, and J. Henkel. On the Usefulness of Type and Liveness Accuracy for Garbage Collection and Leak Detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, Nov. 2002.

[31] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *Proc. of IEEE S&P*, pages 571–585, 2012.

[32] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proc. of NDSS*, 2014.

[33] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proc. of NDSS*, 2015.

[34] S. Lee, T. Johnson, and E. Raman. Feedback directed optimization of TCMalloc. In *Proc. of MSPC*, 2014.

[35] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In *Proc. of USENIX ATC*, pages 209–221, 2016.

[36] Matt Pietrek. A Crash Course on the Depths of Win32 Structured Exception Handling. January 1997. https://www.microsoft.com/msj/0197/exception/exception.aspx.

[37] Microsoft. Enhanced Mitigation Experience Toolkit, 2016. http://www.microsoft.com/emet.

[38] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: Compiler-Enforced Temporal Safety for C. In *Proc. of ISMM*, pages 31–40, 2010.

[39] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of ACM PLDI*, pages 89–100, 2007.

[40] G. Novark and E. D. Berger. DieHarder: Securing the Heap. In *Proc. of ACM CCS*, pages 573–584, 2010.

[41] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proc. of IEEE S&P*, pages 601–615, 2012.

[42] PaX Team. Address Space Layout Randomization (ASLR), 2003. http://pax.grsecurity.net/docs/aslr.txt.

[43] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proc. of NDSS*, 2015.

[44] Rafal Wojtczuk. TSX improves timing attacks against KASLR. https://labs.bromium.com/2014/10/27/

tsx-improves-timing-attacks-against-kaslr/.

[45] T. Rains, M. Miller, and D. Weston. Exploitation Trends: From Potential Risk to Actual Risk. In *RSA Conference*, 2015.

[46] P. V. Sabanal and M. V. Yason. Reversing C++. In *BlackHat*, 2007.

[47] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proc. of IEEE S&P*, 2015.

[48] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proc. of RAID*, pages 88–108, 2014.

[49] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of USENIX ATC*, pages 309–318, 2012.

[50] A. Sotirov. Heap Feng Shui in JavaScript. In *Blackhat 2007*, 2007.

[51] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 256–267, New York, NY, USA, 2015. ACM.

[52] `shifan@freecity.cn`. C++ Object Model. http://lifegoo.pluskid.org/upload/doc/object_models/C++%20Object%20Model.pdf.

[53] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proc. of USENIX SEC*, pages 941–955, 2014.

[54] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In *Proc. of ACM CCS*, pages 927–940, 2015.

[55] V. van der Veen, E. Göktaş, M. Contag, A. Pawloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A Tough `call`: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proc. of IEEE S&P*, pages 934–953, May 2016.

[56] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of ACM CCS*, pages 157–168, 2012.

[57] Y. Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proc. of NDSS*, 2015.

[58] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining Trust on Virtual Calls. In *Proc. of NDSS*, 2016.

[59] C. Zhang, C. Songz, K. Z. Chen, Z. Cheny, and D. Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proc. of NDSS*, 2015.

[60] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proc. of IEEE S&P*, pages 559–573, 2013.

[61] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX SEC*, pages 337–352, 2013.