VTint: Protecting Virtual Function Tables’ Integrity

Chao Zhang*, Chengyu Song‡, Kevin Zhijie Chen*, Zhaofeng Chen†, Dawn Song*

*University of California, Berkeley
†Peking University
‡Georgia Institute of Technology
{chaoz, kevinchn, dawnsong}@cs.berkeley.edu chenzhaofeng@pku.edu.cn csong84@gatech.edu

Abstract—In the recent past, a number of approaches have been proposed to protect certain types of control data in a program, such as return addresses saved on the stack, rendering most traditional control flow hijacking attacks ineffective. Attackers, however, can bypass these defenses by launching advanced attacks that corrupt other data, e.g., pointers indirectly used to access code. One of the most popular targets is virtual table pointers (vfptr), which point to virtual function tables (vtable) consisting of virtual function pointers. Attackers can exploit vulnerabilities, such as use-after-free and heap overflow, to overwrite the vtable or vfptr, causing further virtual function calls to be hijacked (vtable hijacking). In this paper we propose a lightweight defense solution VTint to protect binary executables against vtable hijacking attacks. It uses binary rewriting to instrument security checks before virtual function dispatches to validate vtables’ integrity. Experiments show that it only introduces a low performance overhead (less than 2%), and it can effectively protect real-world vtable hijacking attacks.

I. INTRODUCTION

Memory corruption bugs are still one of the most critical problems in computer security. Attackers can use these bugs to gain unauthorized access to the program state (e.g., the code and data in memory), or even corrupt the state, to indirectly control the program counter and hijack the control-flow to execute malicious code.

In the last decades, pointers to code (i.e., control data), such as return addresses, exception handlers and function pointers, are the most common targets to corrupt. Many defense solutions are proposed and deployed to defeat these control data corruption attacks, e.g., StackGuard [8] for return address, SafeSEH [27] for exception handlers. More generally, DEP (Data Execution Prevention [2]) or W+X prevents writable memory from being executed, and prevents executable memory from being written, making code corruption and code injection attacks ineffective. ASLR (Address Space Layout Randomization [37]) randomizes the locations of code and data, raising the bar of code reuse attacks such as return-to-libc [46] and ROP (Return Oriented Programming [45]).

The vtable hijacking attacks In addition to corrupting traditional control data, modern advanced attacks turn to corrupting other data, e.g., pointers to control data, including the virtual table pointers (vfptr). The vfptr is a hidden field of C++ objects that have virtual functions, generated by all major compilers (e.g., GCC, Visual C++ and LLVM). The vfptr points to a table called virtual function table (vtable), consisting of virtual function pointers associated with the object’s class.

By overwriting a vtable’s contents (i.e., vtable corruption), or overwriting a vfptr to point to an attacker-crafted vtable (i.e., vtable injection), or overwriting a vfptr to point to some existing data (i.e., vtable reuse), attackers can hijack the control flow later when a virtual function is invoked. These three types of attacks form the well-known vtable hijacking attacks.

The vtable hijacking attacks have been increasing in popularity. Attackers can exploit vulnerabilities that can lead to arbitrary or specific memory write, e.g., traditional buffer overflows [42], type confusion [9, 30] and use-after-free [15], to corrupt the vfptr or vtable, and launch the vtable hijacking attacks. Among these vulnerabilities, the use-after-free vulnerability has become a major threat to applications. It accounts for over 80% attacks against the Chrome browser [49], and more than 50% known attacks targeted Windows 7 [25]. As far as we know, most public known real-world exploits against use-after-free vulnerabilities are vtable injection attacks, as this type of attack is more reliable than the other two types.

Researchers have proposed several defenses against the vtable hijacking attacks, such as SafeDispatch [20], a candidate GCC extension [48, 49], and VTGuard [28] (see Section III-C for more detail). These solutions, however, all need accurate type and class inheritance information of target applications, which are not available in binary executables. And thus, they cannot protect the massive legacy applications and closed-source applications. Moreover, these solutions either do not offer sufficient protection against vtable hijacking attacks or incur a high performance overhead.

The approach In this paper, we propose a novel solution VTint to protect binary executables against vtable hijacking attacks with a low performance overhead. In particular, we notice that most legitimate vtables are in read-only sections because no vtables are ever modified during runtime in legitimate programs. On the other hand, the vtable corruption attacks require vtables to be writable, and vtable injection attacks will inject writable vtables into applications at runtime. Based on this observation, we check whether vtables are read-only to validate their integrity at runtime. In this way, our solution VTint prevents writable vtables from being used in virtual calls, and can defeat all vtable corruption and vtable injection attacks. This solution is simple but effective, similar to the DEP solution that prevents writable memory from being executed.
In summary, our VTint protection approach has the following key advantages:

- Easy to enforce. The security policy of VTint is simple and easy to enforce. Similarly to DEP, it blocks all writable memory from being vtables, whereas DEP forbids writable memory from being executed.

- Binary compatible. By using binary analysis and rewriting, VTint is able to protect programs without any source code or debug information. It also offers modularity and backward compatibility support.

- Efficient. VTint’s performance overhead is negligible: less than 0.4% for SPEC benchmark, and less than 2% for real-world browsers.

- Effective. VTint can defend all vtable corruption and vtable injection attacks by enforcing the vtables’ integrity. It also defeats most vtable reuse attacks by checking the validity of vtables. Moreover, it is resilient to information leakage attacks.

The remainder of this paper is organized as follows: We describe the background and problem definition in Section II and III. We then give an overview of our approach and the implementation details in Section IV and V. Section VI evaluates the performance and security of VTint. Section VII discusses the future work and Section VIII compares our approach with the related work. Finally we conclude in Section IX.

II. BACKGROUND

vtable is a general mechanism used in programming languages to support dynamic dispatch (or runtime method binding) [22], e.g., C++ virtual function calls. In this section, we briefly describe the background on how compilers use vtable to implement C++ virtual function calls, the related compiler calling conventions, and an important characteristic of Windows Portable Executable (PE) file format.
A. Virtual Function Calls

Virtual function is a key mechanism to support polymorphism in C++. Figure 1 shows a sample class inheritance, the associated object layout, and the final executable code regarding to the virtual function dispatches.

For each class with virtual functions, depending on the class inheritance hierarchy, the compiler will create one or more associated virtual function tables (vf table), as shown in Figure 1(c). In each instance of the class (i.e., object), there will be a vfptr pointing to the corresponding vf table (Figure 1(b)). If the sub-class has multiple base-classes, its object may have multiple vfptr. Moreover, if virtual inheritance is involved, the layout of objects is much more complex [22]. Unlike solutions that depend on class inheritance information, the VTint solution is independent to the layout of objects.

Figure 1(d) shows the sample executable code of virtual function dispatches. In general, vfptr is read out from the object first; then the target function pointer is read out from the vtable pointed by vfptr; and finally the target virtual function is called.

B. Calling Conventions

When compiling a function, a compiler needs to determine the interface to this function’s caller, such as how caller functions pass arguments to this callee, who is responsible to clean the stack. There are many choices available for compilers. Each choice is called a calling convention.

For virtual function calls, a hidden argument is always passed to the callee, i.e., the pointer to the object instance (a.k.a. this pointer). On x86-32 processors, there are two calling conventions related to the this pointer: thiscall and stdcall, which are adopted by major compilers including Microsoft Visual C++ and GCC. In the thiscall convention, the this pointer is passed in the ECX register. In the stdcall convention, the this pointer is the first argument of the callee, passed by pushing on the stack. On x86-64 or ARM processors, the this pointer is always passed in registers.

C. Relocation Tables

To support dynamic linking and ASLR, the loader may load (i.e., relocate) an executable module’s code and data to a different address other than the address assumed by the compiler. As a result, the loader should update all absolute addresses in the module at load time. Here we call the memory blocks whose contents need to be updated at load time as relocation slots, and call the contents as relocation entries.

The loader needs extra metadata to identify these relocation slots. In PE format files used by Microsoft Windows, the relocation table provides such metadata. The relocation table records both the offsets (e.g., 0x1000 and 0x6000 in Figure 2) of all memory pages that contain relocation slots, and the offsets (e.g., 0x102 and 0x20A) of all relocation slots within each page. The address of the relocation slot can be computed by the base address of this executable image, the page offset, and the slot offset. For example, the first relocation slot in this figure is 0x401102 (= 0x400000 + 0x1000 + 0x102).

Relocation entries are usually absolute addresses of code (e.g., functions) or data (e.g., global variables). In this example,

![Fig. 2: A sample relocation table of Windows PE executables. The relocation entry 0x406500 at slot 0x401102 points to a vf table; the entry 0x402250 at slot 0x405508 points to a virtual function.](image-url)

the relocation entry 0x406500 is actually a pointer to a virtual function table. Entries in this table are all function pointers. And thus they are all relocation entries (e.g., 0x402250) that are indexed by the relocation table too.

With relocation table, we can identify all absolute addresses in binary executables, including addresses of virtual function tables and functions whose addresses have been taken.

Note that we focus on the PE format in this paper, however our approach can be adapted to the Executable and Linkable Format (ELF) files used by Unix and Unix-like systems, since they also have a similar mechanism for relocation.

III. Problem Definition

In this section, we first describe our threat model, and then describe the attacks that VTint aims to defeat, and finally discuss some related defense solutions.

A. Threat Model

We assume attackers have the following capabilities: (1) attackers can read arbitrary readable memory, thus they can launch information leakage attacks and bypass secret-based defenses such as ASLR and VTGuard; (2) attackers can write to any writable memory, thus they can corrupt control data such as return addresses, and pointers to control data such as vfptr; (3) attackers cannot directly read from or write to registers, they can only achieve this indirectly by using existing instructions that propagate data between registers and memory. We further assume (4) attackers can leverage multiple concurrent threads to launch time-of-check-to-time-of-use attacks. Therefore, any writable memory content is untrusted even if it has been checked before it is read out again.

This assumption is strong enough because it covers most capabilities an attacker can retrieve in practice. This assumption is also realistic, because real-world attackers can exploit certain vulnerabilities to obtain these capabilities. Especially for browsers or other applications with script engines, the input data (e.g., JavaScript scripts) will be interpreted as code, or it will be Just-In-Time (JIT) compiled to native code and executed. By feeding these applications with scripts, it is easier for attackers to trigger vulnerabilities, read and write the program state, or create multiple threads to launch attacks.

On the defense side, we assume that (1) popular defenses such as ASLR and DEP are deployed; (2) no legitimate control flows in target applications will allow attackers to change the memory protection of a specified memory region, or to map an
attacker controlled file as read-only memory. If such control flows exist, the target applications should be re-designed to eliminate them, otherwise none of the current defenses can protect them from being exploited. Fortunately, we have not found any of the applications we have tested (including the Firefox and Chrome browsers) contain such behaviors. We also assume that (3) other control flow transfers except for the virtual function calls are well-protected, so they cannot be exploited to hijack the control flow before virtual function calls; (4) non-control-data attacks [6] that may lead to control-flow hijacking are out of the scope of this paper. Given these assumptions, the attackers cannot change any memory’s protection attribute before any virtual function invocation.

On the defense side, we also assume that (1) the binary programs are not obfuscated; (2) the programs are compiled by major compilers that follow the traditional calling conventions, such as GCC and Visual C++. Given these assumptions, it is feasible to perform the needed binary analysis to recover the necessary information for binary instrumentation, by leveraging certain patterns and compiler conventions. Otherwise, it is still an open challenge to enable the needed binary analysis.

B. VTable Hijacking Attacks

The traditional control-flow hijacking attacks can be classified into three categories: code corruption attacks, code injection attacks and code reuse attacks. Similarly, vtable related attacks (denoted as vtable hijacking attacks) could also be classified into three categories: vtable corruption attacks, vtable injection attacks and vtable reuse attacks.

1) Vtable Corruption Attacks: This type of attacks directly corrupts existing vtables’ contents (i.e., virtual function pointers). Once a vtable is corrupted, any further virtual function call related to this vtable will be hijacked. This attack can succeed if and only if the vtables are writable.

2) Vtable Injection Attacks: This type of attacks injects fake vtables into the applications, and overwrite vfptr to point to the fake vtables. These injected vtables are in writable sections, and attackers can control their contents. Among all three types of vtable hijacking attacks, the vtable injection attack is the most popular due to its high reliability.

Figure 1(b) shows a typical vtable injection attack. Attackers first build a fake vtable filled with crafted function pointers; then overwrite the vfptr in an object to point to this fake vtable by exploiting certain vulnerabilities, e.g., use-after-free. As a result, any further virtual function invocation that accesses this vfptr will be hijacked and the function specified in the fake vtable will get executed. Advanced attack techniques such as ROP could be used here to launch reliable attacks.

For example, attackers may first fill the fake vtable with pointers to some ROP gadgets, such as a gadget (xchg eax, esp; ret;). This gadget changes the stack to a memory controlled by attackers (i.e., pointed by eax), and guides further ROP gadgets to get executed. Then, if there is a use-after-free vulnerability in the program, attackers may try to allocate a memory block taking the exact address as the original freed object (e.g., object s in Figure 1(b)), in order to overwrite the vfptr. Then, attackers can hijack the control flow when this freed object’s virtual function is called.

![Figure 3: The VTGuard defense solution. (a) is a sample code snippet using a subclass derived from the base-class. The code in bold font is instrumented by VTGuard to match a secret cookie (i.e., ID) in target vtable. (b) shows three objects of type Base and Sub. (c) shows the associated vtables instrumented with the secret cookies.](image)

3) VTable Reuse Attacks: This type of attacks also overwrites vfptr like the vtable injection attacks. The corrupted vfptr, however, do not point to attacker-crafted vtables, but point to existing vtables, or other existing code or data in memory which are not controlled by attackers.

Unlike the vtable injection attack, it is hard to reliably launch the vtable reuse attack in practice, because there are few existing vtables or data that can be used for attack purposes. As far as we know, there are no publicly known vtable reuse attacks.

C. Existing Solutions

In this section, we introduce three state-of-the-art defenses against vtable hijacking attacks and compare them with our solution. The comparison is summarized in Table I.

1) VTGuard: VTGuard [28] is a defense solution deployed in Internet Explorer (IE) browsers. As shown in Figure 3, the basic idea is to insert a secret cookie into each vtable. Before each virtual function dispatch, VTGuard inserts a security check to examine the corresponding secret cookie. If the cookie in the vtable does not match the expected secret, a security violation is detected and the program is terminated. This solution can effectively mitigate certain vtable reuse attacks, but not vtable corruption and vtable injection attacks.

![Table I: Comparison between defense solutions against vtable hijacking attacks, including whether they can detect vtable hijacking attacks, whether they are resilient to information leakage attacks, whether they support binary programs, and the performance overhead comparison. The abbreviation SD here stands for SafeDispatch.](table)

---

**Table I:** Comparison between defense solutions against vtable hijacking attacks, including whether they can detect vtable hijacking attacks, whether they are resilient to information leakage attacks, whether they support binary programs, and the performance overhead comparison. The abbreviation SD here stands for SafeDispatch.

<table>
<thead>
<tr>
<th>defense solution</th>
<th>vtable hijacking</th>
<th>info leakage</th>
<th>binary support</th>
<th>perf. overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTGuard</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SD-vtable</td>
<td>N</td>
<td>Y</td>
<td>N/A</td>
<td>N</td>
</tr>
<tr>
<td>SD-method</td>
<td>Y</td>
<td>Y</td>
<td>N/A</td>
<td>N</td>
</tr>
<tr>
<td>DieHard</td>
<td>partial</td>
<td>partial</td>
<td>N/A</td>
<td>N</td>
</tr>
<tr>
<td>VTint</td>
<td>Y</td>
<td>Y</td>
<td>partial</td>
<td>Y</td>
</tr>
</tbody>
</table>

---

4
For example, by launching an information leakage attack, attackers can read the expected secret cookie from the security check instruction before the virtual function dispatch. Then, it can forge a *vtable* and put the secret cookie at the proper location inside the fake *vtable*. In this way, the fake *vtable* can pass the security check and the control-flow will be hijacked.

2) **SafeDispatch:** SafeDispatch [20] is a compiler-based defense solution against *vtable hijacking* attacks. It first makes a whole program Class Hierarchy Analysis, to infer the valid *vtable* set and valid virtual function set that may be used for each virtual function dispatch. Then it inserts security checks before each virtual function dispatch.

There are two types of security checks that may be inserted by SafeDispatch: *method checking* and *vtable checking*. The first approach will check whether the target virtual function is in a valid method set, and the second approach will check whether the target *vtable* is in a valid table set.

Without runtime profile information, the average overhead of the method checking solution is about 7%, and the *vtable* checking solution’s overhead is about 30%. In addition, the later solution cannot defeat *vtable corruption* attacks.

3) **DieHard:** DieHard [4, 34] provides a custom memory allocator to randomize and separate the memory allocation. In this way, it can provide a probabilistic guarantee against vulnerabilities such as heap overflow and use-after-free, and thus can protect *vtable* and *vfptr* from being overwritten in some cases. Its average overhead is about 8%.

In comparison, our solution VTint directly protects binaries with a low performance overhead. It can defeat all *vtable corruption* and *vtable injection* attacks, and most *vtable reuse* attacks. It is also resilient to information leakage attacks.

### IV. THE VTINT DESIGN

In this section, we describe the design of VTINT: starting with the security policy we deploy to defeat *vtable hijacking* attacks, followed by an overview of the VTINT workflow, and finally the design of each component.

#### A. Security Policy

The basic security policy enforced by VTINT is simple and effective. Similar to the widely deployed DEP, it enforces that all *vtables* are read-only, prohibiting *vtable injection* and *vtable corruption* attacks. It, however, also faces a similar challenge as DEP. Attacker can launch *vtable reuse* attacks, e.g., by overwriting the *vfptr* to point to existing read-only *vtables* or other data or code in read-only sections, although there are no publicly known exploits that work in this way.

Therefore VTINT also enforces another security policy. It separates *vtables* from other data or code in read-only sections, and enforces that all virtual function calls’ targets are legitimate *vtables*, not other data. In particular, VTINT moves *vtables* to a read-only section, and instruments them with a special ID. At runtime, the virtual function dispatches’ target *vtables* are validated against this ID. As a result, attackers can only reuse existing *vtables*, not any other data in read-only sections. Most *vtable reuse* attacks are thus defeated.

#### B. Overview of VTINT

To deploy this security policy on binary executables, VTINT uses the binary rewriting technique. In general, it parses and disassembles the target binary first, and then identifies all *vtables* and virtual function dispatches in this binary, and finally instruments these *vtables* and virtual function dispatches with security checks to enforce aforementioned security policy. Figure 4 describes the overall workflow.

PEParser first parses the target executable. All basic information, including sections, the export table and relocation table, are retrieved from the binary. The executable binary’s EntryPoint, relocation entries and entries in export table, together form the set of candidate function entries.

Given the candidate function entries, our custom disassembler BitCover conducts a recursive disassembly and identifies all functions, instructions and control tables such as jump tables. This disassembler is based on our previous work [52]. We have introduced some new forward and backward data-flow analysis to BitCover, to identify *vtables* and virtual function dispatches, in order to support VTINT.

Given the *vtables* and virtual function dispatches, the last component VRewriter rewrites the original binary program. More specifically, it moves all identified *vtables* to a read-only section, and instruments them with a special ID to differentiate them from other code or data in read-only sections. Then it inserts security checks before each virtual function call to enforce the aforementioned security policies, i.e., to check whether it is read-only and whether its ID is correct.

Finally, a hardened PE format binary executable is generated. This hardened executable is immune to *vtable corruption*, *vtable injection* attacks, and most *vtable reuse* attacks. Furthermore, it is resilient to information leakage attacks.

#### C. Binary Parsing

As described earlier, PEPARSER will parse the target PE binary, retrieve all basic information, and build a set of candidate function entries to guide further recursive disassembling.
Moreover, it will also identify all candidate vtables. Addresses of vtables will be used in classes’ constructor functions, and thus they must be relocation entries. Furthermore, entries in vtables are all virtual function pointers, and thus they are also relocation entries. So, by examining all relocation entries via the relocation table, we can identify a candidate vtable, if and only if we find a relocation entry which points to an array, and this array is composed of (maybe only one) other relocation entries which point into code sections. In this way, PEParser is able to find out all candidate vtables.

D. Disassembling and Identification

BitCover disassembles the given binary and identifies all vtables and virtual function call sites in it.

1) Disassembling: BitCover takes the original PE executable file as input, as well as the candidate function entries produced by the previous component PEParser, and generates the target binary’s disassembly information.

In general, the workflow of BitCover consists of two phases. First, it greedily explores all the code and data in the program, looking for function entries. For each candidate function entry, BitCover starts a recursive disassembling from it until a stop condition is met. For example, it will stop when an invalid instruction is met, or the new instruction overlaps with some previous resolved instructions. Once BitCover meets an invalid byte sequence, it will mark the code entry from where it starts disassembling as invalid, and continues disassembling the next candidate function entry. In the second phase, BitCover refines the disassembling result. In particular, it will mark some code entries as reliable code entries based on some rules. Then, it propagates the reliability across code entries based on the call graph. Finally it removes unreachable code entries. More details of the disassembler are discussed in our previous work [52], including other stop conditions, and other control data tables’ identification.

2) Identifying VTables: After disassembling, we are able to filter out all vtables from the candidate vtables provided by PEParser. For each vtable, it will be used in the associated class’s constructor function, and its pointer will be assigned to the generated object’s vfptr. Based on this observation, we propose a new forward data-flow analysis to identify these vtable assignment operations, and filter real vtables. More implementation details are discussed in Section V-B.

3) Identifying Virtual Function Call Sites: As Section II-A describes, to dispatch a virtual function call, vtables will be indirectly accessed to get the address of a target virtual function. The workflow is that, a vfptr is read out from an object first (i.e., a memory read operation), and then a function pointer is read out from the table pointed by the vfptr (i.e., another memory read operation), finally the function pointer is indirectly jumped to or called.

Based on this general pattern, we can identify all candidate virtual function call sites. Combining with other rules, such as the propagation of the object pointer (i.e., this pointer) from caller to callee functions, we can further filter out all valid virtual function call sites. In this process, we use several backward data-flow analyses to help identify the propagation of pointers to objects and pointers to vtables. More implementation details are discussed in Section V-C.

E. Rewriting and Instrumentation

Based on the output of BitCover, the next component VRewriter will rewrite the original binary to enforce our security policies. First, it creates a read-only section, and moves all identified vtables to this section, and instruments IDs for vtables in this section. Then, all references to vtables will be updated to the vtables’ new addresses. Finally, VRewriter will instrument security checks for each virtual function dispatch to validate the target vtable’s integrity (i.e., whether it is read-only and contains a correct ID). The instrumented security enforcement code will be put into a new code section, leaving most of the original code section intact.

1) VTable Instrumentation: First, we will move all identified vtables to a new read-only section, in case that some legitimate vtables locate in writable sections. To move a vtable, we need to know its size. However, it is challenging to identify the exact size of a vtable in a binary. We conservatively move several more bytes around vtables to the new section. More implementation details are discussed in Section V-D1.

Then we instrument an identical ID at the beginning of each page in the new vtable section. This ID is randomly selected, and is different from any words at the beginning of any page in any exiting read-only sections. In this way, we only need to instrument an identical ID for a small number of pages, and do not need to modify vtable’s layout. More implementation details are discussed in Section V-D2.

2) Virtual Call Instrumentation: As Figure 1 shows, when a virtual function is invoked, the corresponding vtable will be implicitly accessed to get the target virtual function. VTint will validate the property of the vtable before it is accessed, to enforce the aforementioned security policy.

In particular, before the vtable is accessed, VTint will check whether it is read-only, and whether there is a legitimate ID associated with this table. If one condition is not satisfied, the target vtable is a vulnerable table that can be exploited, and thus will be rejected by the security check. Once a vtable is rejected, a warning can be raised and the program can be blocked. More implementation details are discussed in Section V-E.

F. Modularity and Compatibility Support

For any single module, VTint can identify all of its vtables and virtual function dispatches, only using information collected from the module itself. In addition, the security checks instrumented for this module also work independently. As a result, VTint can be applied to a single module without any problem, i.e., it has a perfect modularity support.

Further, if VTint hardens all modules, there will be no compatibility issues. For legitimate virtual function calls, the instrumented security checks should not fail, because all legitimate vtables have been put in read-only sections and instrumented with an identical ID.

In some cases, however, we cannot harden all modules at the same time. For example, some modules belonging to the operating system cannot be altered by user space applications. For these unharden modules, there are no IDs instrumented for their vtables. As a result, when these vtables are accessed in
the hardened modules, the security checks instrumented before virtual function dispatches will fail and cause false positives.

To solve this problem, VTint also deploys a fail-safe check to eliminate these false positives. In this fail-safe check, only the basic security policy will be enforced, i.e., only read-only property of vtables will be checked. In this way, it is still able to defeat all vtable corruption and vtable injection attacks, and will not generate false positives even for unhardened modules that do not have IDs.

Note, in some very rare cases, the unhardened modules’ vtables are writable, and cannot pass our fail-safe check. We believe this type of modules is extremely dangerous because attackers can directly corrupt these vtables. So we do not handle this type of incompatibility, and will advise the administrator/user to harden such modules immediately.

V. Implementation

In this section, we describe the detail of VTint’s prototype implementation, including the identification of some key structures (e.g., vtables, constructor functions, and virtual function dispatches) in binary executables, and the implementation of the security checks. It is worth noting that, in our current prototype, we focused on Windows PE binaries on x86 platforms compiled from C++ language; but the techniques we developed for VTint is general, and can be extended to handle other executables on other platform.

A. Identifying Constructor Functions

Every C++ object will be initialized explicitly or implicitly by a constructor function. The general workflow of a constructor function is shown in Figure 5. The object to be initialized can be passed in from the constructor’s caller, or be directly allocated in the constructor itself.

In the first case, the constructor’s caller has already prepared the object’s memory, and passes in the this pointer. As described in Section II-B, for stdcall, the this pointer is pushed on stack (Figure 5(a)); and for thiscall, the this pointer is passed through the ECX register (Figure 5(b)).

In the second case, the constructor will allocate the object directly, by calling memory allocation function, such as malloc or new (Figure 5(c)), or by reserving the space on stack (Figure 5(d)).

During the initialization, if the object’s class contains virtual function(s), the constructor function will assign associated vtables to proper locations inside the object’s memory. In particular, the first word of this object’s memory must be a vfptr. If this class has multiple base classes, then there may be other vtable assignments to this memory block.

Based on this observation, we are able to identify constructor functions as follows. For each candidate vtable (i.e., candidate_vtable in Figure 5) identified by the PEParser component of VTint, we locate all assignment operations that assign the candidate vtable to registers or memory (i.e., tmp_reg in the figure).

Then, for each of such assignments, VTint uses a forward data-flow analysis to find out the final target memory of this assignment. If a target memory of form [register+offset] is found, and no other accesses to the candidate vtable (e.g., reading an entry of this table) are made before assigning it to this memory, we mark the register (i.e., obj_reg in the figure) used in this memory as a constructor pointer to an object.

Finally, VTint makes a backward data-flow analysis to locate the source of this object register. If the source of this object register is (1) the first argument of the current function, e.g., Figure 5(a), (2) the register ECX that is passed from the caller function, e.g., Figure 5(b), (3) the return value of memory allocation function such as malloc, new or user-defined functions, e.g., Figure 5(c), or (4) an address on the stack, e.g., Figure 5(d), then this candidate object register very likely points to an object, and the current function is a constructor function.

B. Identifying vtables

After identifying all constructor functions, real vtables can be filtered out from the candidate vtables identified by PEParse. In particular, for each candidate vtable, it is a real vtable if and only if all references to it are assignment operations in constructor functions that assign them to vfptr.

For each candidate vtable, we first find out all its references by examining all relocation entries. Then, for each reference, we apply a static forward data-flow analysis to recognize the propagation of this pointer. If and only if the pointer is assigned to a memory (e.g., vfptr) before all other accesses to this pointer (e.g., read an entry from the candidate table) are made, this reference is marked as a vtable assignment. Finally, if and only if all of these references are vtable assignments, this candidate vtable is marked as an actual vtable.

For these real vtables, their sizes are still not clear. Since it is challenging to recover the exact size of a vtable from a binary, we conservatively compute an upper bound of the size. We scan the words (4-byte) from the beginning of the vtable and increase the size one by one, and stop counting the size if one of the following conditions is met. First, if a word
Fig. 6: Virtual function dispatches. In general, the vfptr is first read from the object; and then the function pointer is read out from the vtable.

The object instance is either passed to the callee (a) explicitly through the first argument, or implicitly by ECX register. When using ECX, there are three cases: (b) the ECX is from the object register, (c) the object register is from ECX, or (d) they are from a same memory.

C. Identifying Virtual Function Dispatches

As described in Section II-A, a virtual function dispatch has at least three steps. First, the vfptr that points to a vtable is read out from the target object. Then, the target function pointer is read out from the vtable. Finally the control-flow transfers to the target function by an indirect call or jump instruction. In most cases, the last two steps can be done in one instruction because the call and jump instructions on x86 platforms support memory operands.

Before introducing the process of identifying virtual function dispatches, we first define a notion definition point. For a register, its definition point is a memory access operation that retrieves the value from the memory and finally propagates it to this register. A special case is that, if the target register is EAX, its definition point could also be any function invocation instruction, because for Intel x86 ABI, EAX always holds the return value of a function invocation.

By analyzing the definition points of registers used in instructions, especially in call/jump instructions, we are able to identify candidate virtual function calls. First, all indirect call/jump instructions (e.g., call fp_reg in Figure 6) are extracted from the disassembling result. Then, for each indirect call/jump instruction, VTint makes a backward data-flow analysis to locate the definition point of the target function (i.e., fp_reg in Figure 6) of this call/jump. If the definition point is found, and there is another register used in this memory access, this new register is thus a candidate vfptr (i.e., vfptr_reg in Figure 6). Next, VTint makes another backward data-flow analysis to locate the definition point of the vfptr_reg register. If a definition point is found, and there is a register used in this memory access, this register is thus a candidate this pointer register (i.e., obj_reg in Figure 6). Now, a candidate virtual function dispatch is identified.

Nevertheless, not all function calls that match the above pattern are real virtual function dispatches. For example, if a structure member is a pointer to a function pointer, the corresponding code to invoke the target function will also match this pattern. Fortunately, in addition to this basic pattern, the callers of virtual function calls need to pass a special argument, i.e., the this pointer, to the callees.

As described in Section II-B, there are two calling conventions adopted by major compilers for passing this pointers, i.e., thiscall and stdcall. For thiscall, the this pointer is passed to the callee implicitly through ECX register; and for stdcall it passed explicitly through the first argument. So there must be a data-flow between the candidate this pointer and ECX register, or the first argument.

Considering this data-flow, there are four general cases of a virtual function dispatch, as shown in Figure 6.

- The first argument pushed for the callee is from the candidate this_reg in Figure 6(a). It is worth noting that, push operations are not the only way to pass arguments. For example, the instruction mov [esp], arg also prepares the argument for the callee. VTint also covers this case.
- At the call site of the indirect call, the value of ECX register is from the candidate this pointer (obj_reg in Figure 6(b)).
- The this pointer (obj_reg in Figure 6(c)) is from ECX register, and the value of ECX has not changed between the load of vfptr and the call site.
- The candidate this pointer (obj_reg in Figure 6(d)) and ECX both come from the same memory.

D. Instrumenting vtables

To enforce the security policy, we need to instrument vtables first, i.e., put them in read-only sections, and differentiate them with other code or data in read-only sections.

1) Moving vtables: We will first create a read-only section (denoted as vtsec) and copy all identified vtables to this section. So, even if the original vtable is writable, the new copy will be set to read-only. During this copy operation, we will conservatively copy more bytes, to make sure all vtable related accesses can work without any problem.

As far as we know, there are two other types of vtable related metadata that may be accessed by the program. First, there may be a RTTI (RunTime Type Information) pointer for some C++ objects (e.g., operands of dynamic_cast<>).
This pointer usually locates right before the `vtable`, and is used for runtime introspection. Second, there may be some offset adjusters before the `vtables` used for adjusting this pointer before invoking a virtual function. For each virtual function in this `vtable`, there is at most one this adjustor. So, for a `vtable` of size \( N \), there may be at most \( 4+4+N \) bytes metadata before the `vtable`. It is worth noting that, all this adjustors are not relocation entries, so we can reduce the size of the metadata if a relocation entry is found within this metadata region.

As mentioned in Section \ref{sec:vt Guardian}, PEParse is able to get an upper bound of the count of virtual functions in a `vtable`. Suppose this upper bound is \( N \), we will copy this `vtable` and its candidate metadata (at most \( 4+4+N \) bytes) before the `vtable` to the new section `vtsec`.

2) **Instrument IDs**: Then we will instrument `vtables` with a special ID, to differentiate them from other data or code in read-only sections. To instrument IDs, we need to answer two questions: where to instrument, and what IDs can we use?

As shown in Figure \ref{fig:vt Guardian}, the VTGuard solution instruments the secret cookies (i.e., IDs) at the end of `vtables`. It will thus change the layout of `vtables`. For binary programs, we can hardly get the class hierarchy information, and we cannot instrument IDs in this way. Similarly, we cannot instrument IDs before `vtables`, as there may be some metadata. Instead, we instrument an ID at the beginning of each page in the new `vtable` section (i.e., `vtsec`). In this way, we do not modify the layout of `vtables`, introducing no compatibility issues.

We propose a novel ID selection solution. More specifically, we randomly select an ID that is different from any words at the beginning of any page of any existing read-only sections. Because the count of such words is limited (i.e., smaller than page count), there are many candidate IDs for use. Unlike other secret-based solutions, this solution is resilient to information leakage attacks. Even if attackers know the ID we are using, they cannot forge a read-only page with this ID, and can only reuse pages in our new `vtable` section.

### E. Instrumenting Security Checks

After identifying all `vtables` and virtual function dispatches, VTInt inserts security checks before the virtual function calls to validate the `vtable`’s integrity. As shown in Figure \ref{fig:vt Guardian}, the security checks will first match the ID of the target `vtable`. Then, it will test whether the `vtable` is readable-only. If any one check fails, the control flow will transfer to pre-defined error handlers, blocking the program from further execution.

It is worth noting that, there is no efficient solution available to check whether a memory location is writable (or readable, executable) on x86 platforms. In Windows, a program is able to get all its loaded modules’ memory map information by traversing a special data structure in the Process Environment Block (PEB). In Linux, it is possible to get the memory map information from the `/proc` file system. In practice, however, both solutions are too slow.

Some APIs provided by the operating system can also be used to test whether a memory is writable. For instance, the `IsBadWritePtr` (Figure \ref{fig:vt Guardian}(b)) in Windows is a qualified API. But this function is inefficient, and will break the program state in some cases. The `read()` function in Linux, which reads file contents to the target memory, can also be used to test whether the target memory is writable. The performance overhead, however, is also high.

VTInt deploys another solution to check whether the target memory is read-only, by utilizing the Windows’ Structured Exception Handling (SEH) mechanism. The function `try_write_vtable` in Figure \ref{fig:vt Guardian}(c) will check if the `vtable` is writable. In particular, this function first registers a special SEH exception handler on the stack. Then, it tries to write the target `vtable` with the ID (i.e., `trigger_seh` in the figure).

If the `vtable` is writable, this write operation will succeed, and the following instruction `jmp ERROR2` will be executed. This error handler terminates the application and never returns to the original virtual function call that is vulnerable.

Otherwise, this write operation will trigger an exception, and the exception handler (i.e., `SEH_handler` in Figure \ref{fig:vt Guardian}) registered in this function will be automatically invoked by the operating system. In this exception handler, we only increase the EIP register in the program state record (i.e., a CONTEXT structure) by 5 and then return. The return value of this handler is set to 0, instructing the operating system to restore the program state and transfer the control flow to the user application.
As a result, the instruction `jmp error2` (5 bytes) will be skipped, and the testing function `try_write_vtable` will return to the original virtual function call site.

### F. Fail-Safe Check

As discussed earlier, in case that some modules are not hardened by VTint, the instrumented security check may fail and cause false positives, because there is no ID instrumented before the `vtables` in unhardened modules.

To handle such cases, we deploy a fail-safe solution to provide better compatibility. This fail-safe solution will only enforce the `vtable` to be read-only, but not enforce the presence of the ID if the current module is not hardened. If VTint is not deployed on all modules, the error handler for ID mismatching (i.e., `ERROR1` in Figure 7) will first check whether current module is hardened by VTint. If yes, it blocks the control flow, and reports an error. Otherwise, it will not block the control flow, but just return back to the original code to further check whether the `vtable` is read-only. In this way, VTint can still defeat all `vtable corruption` and `vtable injection` attacks, and also provides better compatibility.

To tell whether a module is hardened by VTint, we will traverse the Process Environment Block (PEB) of the current executable at runtime, to resolve the section information of the current module. If a read-only section named `vtsec` is found, and the first 4-bytes of this section is the matching ID we used for `vtables`, we can conclude that it is hardened by VTint.

### VI. Evaluation

We have implemented a prototype of VTint for x86 PE executables on the Windows platform. In this C++ implementation prototype, our custom disassembler BitCover uses an open source disassembler library Udis86 [47] to decode x86 instructions. In addition to the 8K LOC of Udis86, BitCover takes about 5k LOC, while VRewriter takes another 3k LOC and PEPARSER takes 2k LOC.

We have tested VTint with the SPEC2000 [16] and SPEC2006 [17] benchmark binaries, and some real world browsers including Firefox, Chrome and Internet Explorer 6 and 8 (denoted as IE6 and IE8),\(^1\) to evaluate our defense solution’s performance and protection effectiveness.

#### A. Performance of Static Analysis

In this section, we first describe our experiment setup, and then show the analysis time of VTint, and the file size overhead brought by VTint.

1) Experiment Setup: The SPEC2000 and SPEC2006 benchmarks are composed of applications written in C/C++ and Fortran. To harden these benchmarks with VTint, we first compile them to PE binaries supporting relocation table. For those applications written in C/C++, we compile them using the Microsoft Visual Studio 2010 compiler. For those written in Fortran, we use the Intel Fortran Compiler to compile them. For each application, all modules are statically linked together. This experiment is performed on a Windows 7 32bit system, with an Intel Core2 Duo CPU at 3.00GHz.

We use VTint to automatically disassemble and rewrite all these benchmark applications’ binaries. The functions identified by VTint are compared with the symbol information from the source code. The result shows that there is no false positives or false negatives when parsing these binaries. We then replace the original SPEC binaries with the hardened binaries generated by VTint, and then run the performance test by using the original SPEC harness scripts. The performance test scripts also check the behavior of the benchmark applications by matching the outputs of applications with expected outputs. Results show that the behavior of hardened applications is same as the original ones, indicating that the disassembling and rewriting of VTint is correct and does not break target applications.

For real world browsers including Firefox and Chrome, we harden each executable module (i.e., *.dll and *.exe) of the browser using VTint. We then replace the original modules with the hardened ones to test the performance and protection effectiveness. Results show that VTint is able to disassemble

---

\(^1\) Newer browsers often require newer OS, causing them hard to be replaced by VTint due to the integrity protection provided by the OS. In addition, there are fewer public exploits available for defense evaluation. Hereby, we chose these two old version browsers for testing.
**TABLE III:** Analysis results of Firefox modules, including the analysis time each module takes, the file size information of the original binaries and the hardened binaries, and the statistics of the VTable related information.

<table>
<thead>
<tr>
<th>App</th>
<th>analysis time (sec)</th>
<th>file size (KB)</th>
<th>VTable info</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig</td>
<td>new</td>
<td>overhead</td>
</tr>
<tr>
<td>crashreporter.exe</td>
<td>1.8</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td>updater.exe</td>
<td>3.7</td>
<td>271</td>
<td>276</td>
</tr>
<tr>
<td>webapprt-stub.exe</td>
<td>1.6</td>
<td>96</td>
<td>97</td>
</tr>
<tr>
<td>D3DCompiler_43.dll</td>
<td>74.3</td>
<td>2,106</td>
<td>2,202</td>
</tr>
<tr>
<td>d3dx9_43.dll</td>
<td>36.9</td>
<td>1,998</td>
<td>2,184</td>
</tr>
<tr>
<td>gkm edias.dll</td>
<td>84.9</td>
<td>4,221</td>
<td>4,493</td>
</tr>
<tr>
<td>libEGL.dll</td>
<td>0.99</td>
<td>59</td>
<td>64</td>
</tr>
<tr>
<td>libGLESv2.dll</td>
<td>23.7</td>
<td>473</td>
<td>519</td>
</tr>
<tr>
<td>mozjs.dll</td>
<td>123.6</td>
<td>2,397</td>
<td>2,444</td>
</tr>
<tr>
<td>msvcp100.dll</td>
<td>5.0</td>
<td>421</td>
<td>450</td>
</tr>
<tr>
<td>msvcr100.dll</td>
<td>13.2</td>
<td>770</td>
<td>778</td>
</tr>
<tr>
<td>xul.dll</td>
<td>328.9</td>
<td>15,112</td>
<td>17,768</td>
</tr>
</tbody>
</table>

**TABLE IV:** Analysis results of Chrome modules, including the analysis time each module takes, the file size information of the original binaries and the hardened binaries, and the statistics of the VTable related information.

<table>
<thead>
<tr>
<th>App</th>
<th>analysis time (sec)</th>
<th>file size (KB)</th>
<th>VTable info</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig</td>
<td>new</td>
<td>overhead</td>
</tr>
<tr>
<td>delegate_execute.exe</td>
<td>21.9</td>
<td>2,105</td>
<td>2,162</td>
</tr>
<tr>
<td>chrome_elf.dll</td>
<td>1.9</td>
<td>132</td>
<td>131</td>
</tr>
<tr>
<td>d3dcompiler_43.dll</td>
<td>59.4</td>
<td>2,106</td>
<td>2,199</td>
</tr>
<tr>
<td>d3dcompiler_46.dll</td>
<td>106.8</td>
<td>3,231</td>
<td>3,489</td>
</tr>
<tr>
<td>libEGL.dll</td>
<td>1.8</td>
<td>174</td>
<td>174</td>
</tr>
<tr>
<td>libGLESv2.dll</td>
<td>31</td>
<td>1,097</td>
<td>1,157</td>
</tr>
<tr>
<td>libpeconnection.dll</td>
<td>23.6</td>
<td>2,461</td>
<td>2,668</td>
</tr>
<tr>
<td>metro_driver.dll</td>
<td>6</td>
<td>504</td>
<td>534</td>
</tr>
<tr>
<td>pdf.dll</td>
<td>70.3</td>
<td>8,577</td>
<td>8,868</td>
</tr>
<tr>
<td>ppgooglenaclplugin.chrome.dll</td>
<td>3.7</td>
<td>331</td>
<td>334</td>
</tr>
<tr>
<td>widevinecdmadapter.dll</td>
<td>1.5</td>
<td>137</td>
<td>138</td>
</tr>
<tr>
<td>xinput1_3.dll</td>
<td>1.3</td>
<td>81</td>
<td>79</td>
</tr>
<tr>
<td>chrome.exe</td>
<td>8.1</td>
<td>852</td>
<td>865</td>
</tr>
</tbody>
</table>

and rewrite real world binaries. For example, the module xul.dll in Firefox (version 17.0.1) has a size of more than 15MB, containing about 6 millions instructions and more than 70,000 functions. VTint can automatically handle these binaries without causing problems. These experiments are also performed on a Windows 7 32bit system.

We also harden some modules of the IE browser (e.g., the core module mshhtml.dll). These modules are extracted from a virtual machine running Windows XP sp3. After hardening, they are copied back to replace the original ones.²

2) Analysis Time: The disassembling and rewriting process of VTint is quite fast. In general, it only takes several seconds to harden a single module.

As shown in Table II, for most SPEC applications that have virtual functions, the analysis time is less than 10 seconds. The application 483.xalancbmk from the SPEC2006int subset takes the longest time, about 212 seconds. Actually it is also the largest one among these applications, with 267K lines of source code and about 900K machine code instructions.

Table III and Table IV show the analysis time of VTint when handling the executable modules of Firefox and Chrome respectively. Most of these modules take less than one minute. The module xul.dll in Firefox takes the longest time (about 5 minutes), due to its size. This module is more than 15MB and has about 6 million instructions.

We also tested the IE 6 browser, the results are similar and we omit the detailed data. For example, the core module mshhtml.dll is about 4MB and has about 1 million instructions. It only takes 51 seconds for VTint to disassemble and rewrite this module.

3) File Size Overhead: VTint needs to copy the original _vtabs_ to a new data section, instrument IDs in this section, and also instruments virtual function dispatches with security checks to validate the target _vtab’s_ integrity. In general, VTint will allocate new data and code sections to put the new _vtabs_ and the security check instructions in.

Moreover, entries in the _vtabs_ are all function pointers, and need to be relocated when the binary module is loaded into memory. As a result, we need to update the executable binary’s relocation table, to record all entries in the new _vtabs_. In addition, the security checks may introduce new relocation entries too. These entries also need to be recorded in the relocation table. Therefore, VTint also creates a new relocation table. All these newly generated relocation table and sections contribute to a file size increment to target binaries.

As shown in Table II, Table III and Table IV, the overall file size overhead brought by VTint is about 5%. For example, the file size overhead of the SPEC2006 application 483.xalancbmk is about 6.97%. The file size overhead of

²These modules are in the system directory, protected by the operating system. And thus, special efforts are needed to replace these modules.
the xul.dll module of Firefox is about 17.57%, largest in all these tested modules. Meanwhile, the file size overhead of the chrome.exe is about 1.47%.

It is worth noting that, for some binaries, their hardened version may have a smaller file size. For example, the libegl.dll module of Chrome becomes smaller after hardening. This is a normal phenomenon, because some binary files include extra padding bytes at the end of the files. VTint will remove these padding bytes, and output a smaller file.

For the IE browser which is not listed here, its file size overhead is also low. For example, the file size overhead of mshtml.dll is about 9%.

Furthermore, in the current implementation of VTint, the security checks are all instrumented in a new code section because there may not be enough space around the virtual function dispatch instructions. But in fact, compilers usually instrument several padding bytes around functions. With an accurate disassembling, it is possible to identify all these padding bytes in the original code section, and then reuse these bytes to instrument security checks. Our future work will apply this optimization to get a smaller file size overhead.

**B. VTable Statistics**

VTint first needs to identify some key structures in the binary, including vtables and virtual function dispatches, before deploying the security policy. In this section, we give the statistics of these structures.

In the Table II, columns in the VTable info group show the statistics related to vtables. For example, among the 872K instructions of 483.xalancbmk, there are 4248 virtual function dispatches. These virtual function dispatches may use virtual function pointers coming from 29 vtables. There is a special case in this table, the application 473.astar has two vtables, but has no virtual function dispatches. After manual analysis of the source code, we confirm that, there are two classes containing virtual functions. But any objects of these two classes or their sub-classes never invoke these virtual functions. In other words, the vtable information of 473.astar identified by VTint is correct.

Table III and Table IV shows the statistics of vtable related information of the modules from Firefox and Chrome. For example, in the Firefox’s module xul.dll, there are 6548 vtables, and 54743 virtual function dispatches. As there are about 5.8 millions of instructions in total in this module, we find that, in every 100 instructions, there is about one virtual function dispatch. This ratio is quite high, indicating that the attack surface of vtable hijacking attacks is large.

**C. Runtime Performance**

The runtime performance is a key factor affects the adoption of a defense solution. In this section, we test the overhead brought by VTint by comparing the performance of original binaries and the hardened binaries of several applications.

1) Runtime Overhead on SPEC benchmarks: First, we test the SPEC benchmark applications’ performance. Table II shows the performance data of the original binaries and the hardened binaries. The average performance overhead is about 0.37% for these applications. The highest performance overhead is the one of 252.eon, about 2.96%. Some applications even get faster after hardening, such as the 444.namd and 453.povray, maybe due to some experimental errors.

It is worth noting that, because there is no virtual function dispatches in 473.astar (although it has 2 vtables), VTint does not instrument any security checks for this application. On the other hand, the performance overhead of 473.astar is zero, also respects this fact.

2) Runtime Overhead on Real World Browsers: Then we test the performance of two real-world browsers: Firefox and Chrome. We use six popular browser benchmarks to evaluate these two browsers’ performance on JavaScript, HTML rendering and HTML5 support, including Google’s Octane [14], Mozilla’s Kraken [29], Apple’s Sunspider [3], Microsoft’s LiteBrite [26], RightWare’s BrowserMark [41] and FutureMark’s PeaceKeeper [12].

As shown in Figure 8 and Figure 9, the overall performance overhead for all browsers is about 1.6%, whereas the average overhead of Chrome and Firefox is about 1.37% and 1.84%.

3) Performance Analysis: The runtime performance overhead brought by VTint is lower than most of previous solutions. The SafeDispatch solution [20] introduces a performance overhead of 7% (using method checking) and 30% (using vtable checking), when no runtime profile information is provided. The memory allocation solution DieHard [4] also introduces an overhead of 8%.

The major performance overhead of VTint comes from the instrumented security checks that test the target vtable’s read-only property. As described in Section V-E, the current
solution used by VTint is based on Windows’ Structured Exception Handling (SEH) mechanism. When testing legitimate read-only vtables, the security checks will trigger memory write violation exceptions. The registered SEH handler will then catch these exceptions, and finally redirect the control flow to the application’s original code. When testing illegal vtables that are writable, the security checks will not trigger exceptions, but flow to a predefined error handler.

This testing works fine, but its performance overhead is still high. Most (even all) target vtables at runtime are legitimate and thus are read-only, and cause many memory write exceptions, leading to a higher-than-desired performance overhead. As far as we know, there is no other alternate solution that is more efficient to check whether a memory is writable. We hope the hardware or the operating system can provide supports to do a quick memory property check in the future, similar to the hardware support for DEP. In that case, the performance overhead of VTint will become much lower.

D. Protection against Real World Exploits

To evaluate the effectiveness of VTint, we choose 6 publicly available vtable hijacking exploits from the Internet, including security research blogs such as Vupen’s blog, the penetration testing framework Metasploit [24] and the exploit database exploit-db [35], as shown in Table V. These exploits all target real-world browser applications, e.g., Internet Explorer and Firefox, by exploiting vulnerabilities such as use-after-free and heap overflow. They all inject fake vtables into the memory and finally launch the vtable injection attacks. For the Chrome browser, there are very few public available exploits. So, it is not tested here.

Table V also lists other detailed information of the 6 exploits we used. For each exploit, we list the vulnerabilities’ CVE-ID, application version, and type of the vulnerabilities and the reference of a detailed POC description.

We carry out these experiments in a virtual machine running Windows XP SP3. Core modules of target applications are extracted from the virtual machine first. Then VTint disassembles and rewrites these modules in our analysis platform. Finally, these hardened modules are copied back to the virtual machine to replace the original ones. After deploying the hardened modules, we drive target applications to access the malicious URLs that contain these exploits. Results show that, the security checks instrumented by VTint will block all these exploits. In other words, VTint is able to protect real world browsers from vtable hijacking attacks.

VII. Security Analysis and Discussion

By checking the memory property of vtables, VTint can defeat all vtable corruption and vtable injection attacks. The only way to bypass VTint is to reuse fake vtables in the read-only memory, like using code reuse attacks (e.g., return-to-libc or ROP attacks) to bypass DEP.

In addition, VTint will instrument a special ID for vtables, and match this ID before virtual function dispatches. This solution is resilient to information leakage attacks. As a result, attackers can only reuse existing vtables in the read-only memory to bypass VTint. As there are not many vtables in an application, the attack surface is small. It is also hard to find an existing vtable that is useful to launch further attacks.

Further, a fine-grained vtable checking policy is able to defeat this type of vtable reuse attack. A fine-grained vtable checking policy, however, requires the type information and class hierarchy information of target applications, which is hard to retrieve using binary analysis. Our future work will focus on identifying this information from binaries, and extend VTint to defeat all vtable hijacking attacks.

Currently, to deploy VTint in practice, we can combine it with other lightweight solution to provide better security. For example, the solution VTGuard is a perfect choice. VTGuard also instruments IDs for each vtable. It can defeat vtable reuse attacks, but is vulnerable to vtable corruption and vtable injection attacks. It also has a negligible performance overhead (less than 0.5%).

Combining with the VTGuard solution, VTint can defeat all vtable hijacking attacks. The overall performance overhead is still acceptable (it should be less than 2% together). It is worth noting that, after combining, the VTGuard solution is also resilient to information leakage attacks, because the target vtables must be read-only and cannot be forged or corrupted by attackers even if VTGuard’s IDs have been leaked. Moreover, currently VTint implements a similar ID checking to defeat some vtable reuse attacks. When combining with VTGuard, this ID checking can be merged with VTGuard’s. As a result, the overall performance overhead can be further reduced.

VTint is a pure binary instrumentation solution. It identifies vtable-related components (e.g., virtual function tables and virtual function call sites) from the binary, by leveraging certain patterns and calling conventions implemented by major compilers. VTint also cannot handle binaries that are obfuscated. For binaries which are obfuscated or do not follow these patterns, it is still an open challenge to enable the needed binary analysis.

VIII. Related Work

In this section, we discuss some representative works on the defense of vtable hijacking attacks and other control flow hijacking attacks, as well as the binary analysis work.

Memory Safety. The memory safety policy ensures that no out-of-bounds or dangling pointers can be exploited for unauthorized read or write of memory. As a result, attackers cannot tamper target applications’ program states, or launch control-flow hijacking attacks, including the vtable hijacking attacks. There are many memory safety solutions proposed.
by researchers, e.g., [10, 36, 44, 51]. These solutions usually instrument pointers with extra metadata when they are created, track the metadata during the program execution, and check the metadata when these pointers are used to access memory.

The most representative solutions are the spatial memory safety solution SoftBound [31] and the temporal memory safety solution CETS [32]. These solutions, however, all introduce a high performance overhead, prohibiting their adoption. For example, the combination of SoftBound and CETS will enforce a complete memory safety at the cost of 2x or more performance overhead.

Code Pointer Integrity (CPI) [21] proposed a lightweight memory safety solution, protecting only sensitive pointers including code pointers. The performance overhead of CPI is about 8.4%. It also needs the source code to enable the code pointer analysis and instrumentation.

Control-Flow Integrity. The CFI solution provides a strong guarantee that all control flow transfers must comply with programmers’ intentions, i.e., they must respect the program’s compile-time Control-Flow Graph (CFG). It can stop many different types of control flow hijacking attacks including ROP [45], return-to-libc [46] and vtable hijacking attacks. The original CFI solution was proposed in 2005 [1]. It is not adopted by the industry, however, because it requires source code of target applications and introduces a high overhead.

Recently proposed coarse-grained CFI solutions [52, 53] deploy CFI directly on binary executables. As there is no type information in binaries, only a coarse-grained CFI policy is thus enforced. As a result, attackers can bypass the protection in some cases [13].

Some other CFI solutions [33, 50] enforce a fine-grained CFI policy on target applications. These solutions depend on the information collected by compilers at compile-time or by virtual machines. The dependency on source code and the high performance overhead, however, also restrict their adoption.

VTable Hijacking Defense. SafeDispatch [20] hardens the program at compile time to defeat vtable hijacking attacks. It utilizes the LLVM compiler infrastructure to perform a whole program Class Hierarchy Analysis, and compute the valid method set and the valid vtable set for each virtual function dispatch, and then validate them at runtime. Researchers from Google also proposed a similar approach [48]. These solutions, however, need source code of all modules and require recompiling target applications to deploy the security policy. In addition, their runtime overhead is still high.

VTGuard [28] is another solution to protect applications from vtable hijacking attacks, deployed in the modern Internet Explorer browsers. This solution instruments secret cookies in vtables and match them at runtime. It has a negligible performance overhead and can defeat vtable reuse attacks. But it also needs target applications’ source code. Furthermore, it is vulnerable to information leakage attacks, vtable corruption and vtable injection attacks.

DieHard [4] proposes a custom memory allocator to provide probabilistic memory safety at runtime. By randomizing and replicating heap objects, it can tolerate various memory errors (e.g., heap-based buffer overflows, use after free) with a high probability. As a result, some of the vtable hijacking exploits can be eliminated by this solution. The average overhead of this solution is about 8%.

Binary Analysis. Many studies have been made to analyze binaries. Schwarz et al. [43] discuss the disassembly problem in detail, including two standard algorithms and a new combination. Many other approaches [18, 19, 39] are proposed to identify code and data in binaries. Some systems such as Vulcan [11] provide a general framework for binary rewriting. The work [9] also discusses some efforts to identify virtual function dispatches. It also uses some similar heuristics as VTint, but targets different problems. In addition, it is built on top of the commercial disassembler IDA Pro [18], and relying on the LLVM compiler framework to analyze target binaries.

IX. CONCLUSION

In this paper, we propose a novel approach VTint to defeat vtable hijacking attacks. It validates the vtables’ integrity by checking the memory’s read-only property and the ID associated with vtables. It is able to defeat all vtable corruption and vtable injection attacks, and most vtable reuse attacks. It is also resilient to information leakage attacks.

VTint can be applied through binary rewriting on executables generated by modern compilers. It provides a good modularity support and backward compatibility support. Its runtime performance overhead is low (less than 2%). It can defeat real world vtable hijacking attacks.

ACKNOWLEDGMENT

This research was supported in part by the Natural Science Foundation award CCF-0424422, DARPA award HR0011-12-2-005, and FORCES (Foundations Of Resilient CybEr-Physical Systems), which receives support from the National Science Foundation (NSF award numbers CNS-1238959, CNS-1238962, CNS-1239054, CNS-1239166).

REFERENCES


