An Evil Copy: How the Loader Betrays You

Xinyang Ge
Microsoft Research
xing@microsoft.com

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

Trent Jaeger
The Pennsylvania State University
tjaeger@cse.psu.edu

***Abstract*—Dynamic loading is a core feature used on current systems to (i) enable modularity and reuse, (ii) reduce memory footprint by sharing code pages of libraries and executables among processes, and (iii) simplify update procedures by eliminating the need to recompile executables when a library is updated. The Executable and Linkable Format (ELF) is a generic specification that describes how executable programs are stitched together from object files produced from source code to libraries and executables. Programming languages allow fine-grained control over variables, including access and memory protections, so programmers may write defense mechanisms assuming that the permissions specified at the source and/or compiler level will hold at runtime.**

Unfortunately, information about memory protection is lost during compilation. We identify one case that has significant security implications: when instantiating a process, constant external variables that are referenced in executables are forcefully relocated to a writable memory segment without warning. The loader trades security for compatibility due to the lack of memory protection information on the relocated external variables. We call this new attack vector CoRev for Copy Relocation Violation. An adversary may use a memory corruption vulnerability to modify such "read-only" constant variables like vtables, function pointers, format strings, and file names to bypass defenses (like FORTIFY_SOURCE or CFI) and to escalate privileges.

We have studied all Ubuntu 16.04 LTS packages and found that out of 54,045 packages, 4,570 packages have unexpected copy relocations that change read-only permissions to read-write, presenting new avenues for attack. The attack surface is broad with 29,817 libraries exporting relocatable read-only variables. The set of 6,399 programs with actual copy relocation violations includes ftp servers, apt-get, and gettext. We discuss the cause, effects, and a set of possible mitigation strategies for the CoRev attack vector.

I. Introduction

Software written in C/C++ is prone to memory corruption vulnerabilities through memory safety violations and type confusions, allowing an attacker to corrupt both data and code pointers. A generic memory corruption vulnerability allows an attacker to overwrite an arbitrary memory address with attacker-controlled data. Each memory corruption is different and some only allow partial control of the target address and/or the value that is being written. Despite significant investment in bug finding techniques, memory corruption is still an important problem, as 745 individual CVEs for 2015 and 692 CVEs for 2016 are reported. While not all these vulnerabilities allow an attacker to compromise a system with arbitrary code execution, many do.

Without any defense, attackers inject and execute code to take control of a system through memory corruption vulnerabilities. Over the past decade, a set of defense mechanisms have been deployed on commodity systems. Data execution prevention [5] is a common defense that enforces code integrity. Code integrity prohibits an attacker from injecting new code into a running process and is usually enforced by hardware (e.g., through the non-execute flag on a per-page basis on x86). With the rise of code-injection protection [32], [39], [22], attackers have moved towards code-reuse attacks. In a code-reuse attack, the attacker combines existing code fragments (called gadgets) to achieve arbitrary computation. While code-reuse attacks are Turing complete, they are generally used to disable code integrity and to allow an attacker to execute injected code. Two other deployed defense mechanisms, stack canaries [25] and address space layout randomization [40], protect against some control-flow hijack attacks and make it harder for an attacker to find suitable gadgets. Unfortunately, as the list of CVEs shows, these defenses are often mitigated by an attacker.

New defenses like Control-Flow Integrity (CFI) [3], [11] are on the verge of being widely adopted. Several proposed mechanisms are highly practical, have low overhead, and are suitable for production (i.e., they support common features like modularity and do not require source code annotations, changes, or blacklisting) [41], [43], [30], [20], [21], [16], [26], [10], [29], [31], [34]. CFI verifies that the target address observed at runtime is feasible according to a statically constructed control-flow graph. Individual CFI mechanisms differ in the underlying analysis of the control-flow graph and in the enforcement mechanism.

Current programs rely on dynamic loading (e.g., through the `ld.so` dynamic loader on Linux) to support shared libraries, position independent code, and defense mechanisms like address space layout randomization (ASLR). Dynamic loading is central to how current systems execute applications. Through clever design, dynamic loading enables sharing of code across multiple processes, thereby reducing memory usage. As most pages of a library are read-only code, they can easily be shared across processes that use the same library as long as the pages do not contain hard-coded addresses. Shared libraries solve this problem by using relative addressing that redirects accesses through a set of per-process pages that

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23199

are writable for each library. In ELF binaries, these pages are referred as Global Offset Tables (GOT). As part of the loading process, the dynamic loader has to allocate space for individual shared objects and resolve references among them.

Dynamic *libraries* are generally created by linking position independent code. All references are either relative to the current module or use indirection through a set of tables such as GOTs. These tables are then updated at runtime whenever the code is placed at a certain location. Dynamic *executables* (as opposed to static executables or executables located at a fixed address) on the other hand are generally not created from position independent code. Any reference to external data would therefore have to be resolved and patched at runtime. The ELF standard defines different types of relocations for this purpose, allowing online patching. While relocations generally patch the location of the reference (i.e., they patch the location with the correct address used at runtime), modifying read-only regions should be avoided. If the code region in the executable contains external references, then all such code pages would have to be modified, making it harder to share code among processes. Copy relocations fill this gap and enable dynamic executables to relocate the target object. The executable allocates space for the target object in its `.bss` section and the loader will then copy the object from the source shared library to the `.bss` section of the executable. As all shared objects use indirect references to access this object, the loader then modifies all these pointers to point to the copy in the executable.

Such copy relocations may result in severe security violations because the loader is unaware of the protection flags of the original symbol and can therefore no longer enforce memory protection. The original symbol may be allocated in read-only memory, but if the dynamic executable references this object, the loader will copy it to a writable memory location. An attacker can now use a memory corruption vulnerability to modify the presumed read-only symbol. This has security implications if defenses depend on assumed read-only memory permissions. Format string protections [27] assume that the format string is in a read-only section. Modifying the format string allows an attacker to read or write arbitrary memory and to execute Turing-complete code [12]. For CFI, many mechanisms [41], [10], [26] assume that C++ vtables are in read-only memory, as guaranteed by the compiler, but these guarantees are broken by the linker and the dynamic loader. As vtables are assumed to be immutable, they are not checked, and an attacker may circumvent any CFI or other control-flow hijacking mechanism that assumes immutable vtables. We call this attack vector CoREV for Copy Relocation Violation.

CoREV is not just a theoretic attack vector, but such dangerous relocations actually exist in current software. We have examined all 54,045 Ubuntu 16.04 packages and found that 6,339 binaries feature such relocations. We classify vulnerable relocations into the following seven categories: (i) vtables, (ii) function pointers, (iii) generic pointers, (iv) format strings, (v) file and path names, (vi) generic strings, and (vii) others. Writable format strings allow an attacker to mitigate any printf-based defenses and enable printf-oriented programming [12] while writable file and path names allow an attacker to, e.g., redirect input and output. Writable vtables and function pointers on the other hand allow an attacker to mitigate future defenses that protect against control-flow hijacking by overwriting code pointers that are assumed to be read-only and therefore not checked. In total, we have found 69,098 copy relocations that change the original memory protection. These include 24 format strings, 44 file and path names, 711 function pointers and 28,497 vtables. Our evaluation under approximates the total attack surface and shows the severity of CoREV. These dangerous copy relocations may not directly lead to successful exploitations, however, because adversaries must additionally find a memory corruption vulnerability to modify the relocated variables, but the prevalence of CoREVs provides adversaries with more opportunities for bypassing defenses and/or launching attacks.

We propose a set of three mitigations against CoREV. First, for existing binaries, the best we can do is to detect such malicious relocations and prohibit execution (or at least warn the user). Second, if compiler flags can be changed, we propose to recompile dynamic executables using `-fPIC` which compiles dynamic executables using the same indirection for references as used for dynamic libraries, removing the need for copy relocations. Third, if the binary cannot be compiled as position independent code, we propose to change the toolchain to make the constraints that are only available at the source code and compiler level explicit and preserved along the toolchain, so that both linker and loader are aware of the read-only nature of individual symbols. The loader can then update the permissions accordingly after initial relocation.

This paper presents the following contributions:

1) Discussion of a new attack vector called CoREV based on copy relocations that allow attackers to violate memory integrity assumptions.
2) An evaluation of the prevalence of CoREV by examining copy relocations for all packages of Ubuntu 16.04. We show that 4,570 packages have unexpected copy relocations that change memory protections.
3) A presentation of three possible mitigations of this new attack vector.

II. Background

A. Dynamic Linking and Loading

Modern operating systems adopt dynamic linking and loading to enable modularity. Dynamic linking has two major advantages over the traditional static linking. First, the library code can be shared among processes, so that a system needs only one physical copy in memory per binary. Second, once a bug is found in a library implementation, fixing the bug and distributing the updated library suffices if it is dynamically linked; otherwise, rebuilding every single binary that statically linked the vulnerable library is required. As a result, all major operating systems (e.g., Windows, Linux, and macOS) rely on dynamic linking by default. For the rest of the paper, we focus, without loss of generality, on the dynamic linking implementation in GNU/Linux. In Section VII-E, we discuss how other operating systems implement dynamic linking regarding CoREV attacks.

Most Linux systems use a unified format called Executable and Linkable Format (ELF) for executables, dynamic libraries (*.so) and object files (*.o) [19]. Conceptually, an ELF file

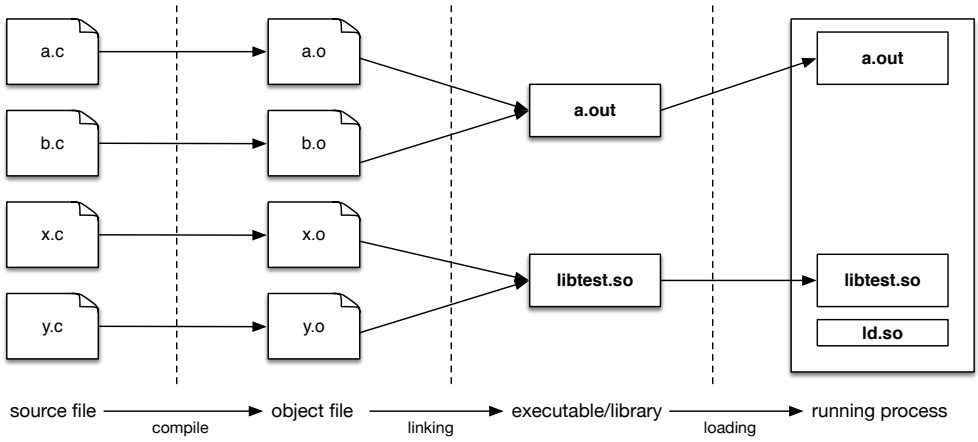

Fig. 1: An overview of compilation, static linking and dynamic loading.

contains a set of sections. Some sections are required for execution and will be mapped into the process address space at runtime (e.g., code and data), while others may store optional descriptive information (e.g., symbol table and debug information). Each mapped section is associated with a memory permission. For example, code sections and read-only data sections are mapped as non-writable while the other data sections are mapped as writable (but not executable). A special section called the relocation section stores a list of unresolved references (i.e., absolute/relative addresses) that require further attention at a later time (i.e., link time and/or runtime). For example, an object file may have a relocation entry on the operand of a direct call instruction. At link time, the relocation entry will be resolved so that the call goes to the right function. Each relocation entry contains necessary information to help determine *how* it should be resolved.

Dynamically-linked programs require runtime support to reference code and/or data that reside in different modules (e.g., a library). A program does not *statically* know (1) which module provides the required code/data and (2) where in the address space of the process the module is loaded at runtime. Consequently, modern systems rely on a small runtime called *dynamic loader* (`ld.so`) to handle both issues. The dynamic loader is responsible for loading libraries into the address space and resolving necessary inter-module references so that the program runs correctly.

In addition, dynamically-linked programs require cooperation from the compiler toolchain (including the static linker) so that the dynamic loader may resolve inter-module references at runtime without modifying their code sections. On Linux, this is achieved through the use of another level of indirection: the Global Offset Table (GOT). Each module has its own GOT section. The GOT contains the addresses used for external references needed by the module. Each address is filled by the dynamic loader at runtime, as a result of resolving the corresponding relocation entry. For example, to invoke the `printf` function in libc, an executable first makes a direct call to a local trampoline in the Procedure Linkage Table (PLT), which in turn *jumps* to the actual address of `printf` stored in the corresponding GOT entry. However, as we will show in Section III, not all external references use the GOT in practice, which creates security issues.

We present an overview of how a program is compiled, linked, and loaded in Figure 1. A program often consists of multiple source files. First, the compiler compiles each source file into an object file. A typical object file contains a code section, multiple data sections and necessary relocation information. Second, the static linker takes as input the object files, resolves references that can be done statically based on the relocation information, allocates GOT entries for references that need to be resolved by the dynamic loader at runtime, and outputs an executable or a library. Third, when executing a program, the dynamic loader loads its dependent libraries into the process, performs dynamic linking by resolving remaining references that require runtime relocation (e.g., filling GOT entries of each module), and transfers the control to the entry point (i.e., `main` function) of the program. Note that dynamic linking can be done lazily, resolving individual targets whenever first needed.

B. The Importance of Read-Only Data

It is now widely accepted that code sections must be read-only and executable while data sections must be non-executable to prevent attacks. Solutions like PaX [32] and DEP [5] prevent the execution of writable memory to prevent code injection attacks. If adversaries find a way to modify executable code, then they can attack the process by injecting and executing code of their choosing. PaX and DEP aim to partition the process into immutable and executable code sections and mutable but non-executable data sections to prevent such attacks. Researchers even argue that code sections should be execute-only [7], [15].

In addition, processes often include a variety of data that must be read-only. To enable the memory protection, the compiler toolchain produces the information necessary to inform the dynamic loader that certain program data should be restricted to read-only memory. Typically, an ELF file often includes a section for read-only data, namely `.rodata`. When the compiler detects constant variables in the program source, it adds those variables to the `.rodata` section of the generated object files. The linker then combines individual `.rodata` sections of the object files to form a single

`.rodata` section for the executable or library binary. Finally, the dynamic loader maps the ELF file's `.rodata` section into read-only memory to enable memory protection.

The security of the processes often depend on the read-only memory protection. The read-only data section in an ELF file consists of static constants and variables that are used by the program. Note that the compiler may also choose to place such read-only data in the code section to reduce the number of required memory pages. Such constants may include fixed strings (e.g., format strings or filenames), fixed data values (e.g., structured data, arrays, or IP addresses), and fixed code information (e.g., arrays of function pointers, C++ virtual tables, or jump tables). Programmers assume the values of static constant variables are stored in read-only memory and remain immutable after initialization. Failing to adhere to the assumption can lead to security breaches [42].

Program security often leverages read-only data. While normal program data may be maliciously modified when a memory corruption error is exploited by an adversary, read-only data cannot be modified. As a result, security experts encourage the use of read-only data to prevent attacks and sometimes apply read-only data in their defenses. For example, researchers have proposed that a solution to format string vulnerabilities is to hard-code format strings [14], [27]. If printf invocations leverage adversary-controlled format strings, then the call can be used to create a Turing-complete exploit environment [12]. As another example, some control-flow defenses for C++ programs depend on an adversary not being capable of modifying virtual tables [41]. These defenses assume the integrity of virtual tables and simply check if an intended virtual table is used during a virtual method invocation to restrict the possible targets and prevent code reuse attacks. While using the C++ virtual table to identify targets is not the only way to restrict control flow targets for C++ virtual method invocations, it presents performance advantages, as discussed in Section IV-A.

Programs may also leverage read-only data to prevent attacks on system calls. One problem is that adversary input may be used to construct resources names, such as file names and IP addresses. However, if such resource names are hard-coded in the program, then the program cannot be tricked into serving as a confused deputy [24]. Further, even if there is a memory corruption error in the program, as described above, adversaries cannot maliciously modify such resource names when they are read-only data.

III. Problem Definition

The loader may fail to maintain the implicit requirement that data in the `.rodata` section of an ELF file *must always* be protected read-only. When a program references read-only data that is defined in a library, the loader moves the data into a data section of the program that is writeable, enabling adversaries to modify such data. Since programmers and defense mechanisms depend on such data being read-only, this loader behavior introduces a new attack vector for adversaries to exploit. We call the new attack vector CoRev for *Copy Relocation Violation*. Next, we describe how and why current loaders enable CoRev.

main.c

```c
extern const int foo;

int main() {
  int *p = (int *)&foo;
  *p = 100; // page fault!
  return 0;
}
```

test.c

```c
const int foo = 10;
```

Makefile

```
all: main.c test.c
  gcc -fPIC -shared test.c \
      -o libtest.so
  gcc main.c -L. -ltest

run: a.out libtest.so
  LD_LIBRARY_PATH=. ./a.out
```

Fig. 2: An example program for problem demonstration.

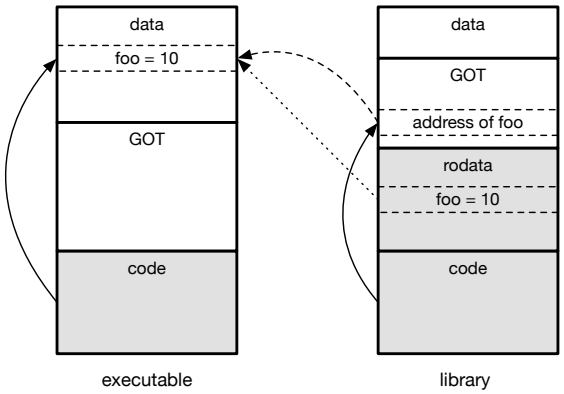

Fig. 3: Copy relocation for the example program. The dotted arrow shows the variable copy, and the dashed arrow indicates the points-to relationship. Solid arrows indicate data accesses in the program.

A. Example Scenario

Consider an example program in Figure 2. In this example, the main executable references a constant variable `foo` defined in a library and tries to change its value. Presumably, this will trigger a page fault because the constant variable `foo` resides in the library's read-only data section.

Counterintuitively, this access does not trigger a page fault in practice (the example program was tested on Ubuntu 16.04). When an executable references a constant variable defined in a library, the dynamic loader "relocates" (moves) the constant variable from the library's read-only data section to the executable's writable `.bss` section (which typically contains uninitialized data of a program). This action removes the read-only memory protection of the variable `foo`, allowing the variable to be written in the example program. Other types of read-only variables defined in a library, but used by an executable, are relocated in a similar way. As described in Section VII, 6,339 out of 34,291 programs exhibit this behavior, making them vulnerable to unexpected exploits.

Such variable movement is initiated by a special relocation type called *copy relocation* applied to variables in the executable. We show its effects in Figure 3. Basically, a copy relocation instructs the dynamic loader to move a variable to

the address specified by the relocation entry. The dynamic loader not only copies the variable value (shown as the dotted arrow), but also redirects the references in other libraries (including the one which actually defines the variable, e.g., `libtest.so` in our example) to the new location by setting up the variable's corresponding GOT entries accordingly (shown as dashed arrow). This ensures the old copy in the library can be safely discarded.

B. The Purpose of Copy Relocation

Copy relocations are an artifact of the process implemented by the compiler toolchain to unify how references to external variables are resolved between static and dynamic linking. Modern compiler toolchains split the build process into multiple stages as shown in Figure 1. In particular, the separation between the compilation and linking steps makes *separate compilation* possible, enabling program modules to be produced independently and linked either with other object files or libraries to run the executable. This feature improves build efficiency, encourages collaborations on the same project, and simplifies code management. However, such a design limits the information available at each stage. As we will show, the current practice of separate compilation for read-only variables is the root cause of CoREV.

As shown in Figure 1, the compiler takes a source file and generates an object file. When the source code references an external symbol (e.g., a variable or a function), the compiler creates a placeholder for its address in the emitted instruction and allocates a relocation entry for the placeholder. Consequently, the generated instructions do not go through GOT by default. The implicit assumption made by the compiler here is that the placeholder can always be updated with the actual address by resolving the relocation entry when the program is eventually linked. While this assumption may hold for statically-linked programs, it is not always the case for dynamically-linked programs where the referenced symbol may be externally defined in a library (e.g., the variable `foo` in Figure 2) and hence its address is not known until runtime.

There are two kinds of external references, and the linker handles them in different ways. In the first case, the executable references external code by calling library functions. To resolve the relocation statically and hence satisfy the compiler's assumption, the linker can *relay* the control transfer. Specifically, it updates the placeholder (e.g., a call operand) to point to a linker-generated trampoline in the Procedure Linkage Table (PLT), and makes the trampoline perform a GOT-based indirect jump to redirect the control to the actual library function.

In the second case, the executable references external data by using library variables. Unlike external code references, the linker cannot effectively relay data accesses. Thus, the linker has two choices. First, it can leave the placeholder to the dynamic loader and let it resolve the relocation entry at runtime. Unfortunately, since the placeholder resides in the program's code section, this implies that the loader has to modify the program's instructions at runtime. Thus, the same binary cannot be shared among concurrent processes. Furthermore, on x86-64 Linux, the placeholder generated by the compiler only has four bytes by default, which is insufficient to encode an eight-byte address of an arbitrary library variable. Second, the linker can collude with the loader by allocating a local copy within the executable as if it were locally defined. Specifically, the linker allocates zero-initialized space for the external variable in the `.bss` section of the executable, and updates the placeholder to reference the local copy when linking. The dynamic loader then copies the value of the originally referenced library variable to the local copy at runtime as shown in Figure 3. After moving the variable from the library to the executable, all libraries that are using this variable must update their references from the original location in the library to the location in the executable by updating their GOT sections. The library that hosts the original copy of the variable must update its location to the executable as well. This ensures a consistent program state for all libraries. Therefore, the linker is able to resolve the relocation entry for the placeholder statically. As a result, the current compiler toolchain adopts the second solution by using copy relocations.

However, as shown in the example in Figure 2, copy relocations move an external variable to the executable's writable data section regardless of its original memory protection. There are two reasons for this design. First, the linker cannot *reliably* determine the original memory protection set on the moved library variable. This is because dynamic linking allows the overriding of symbols based on the order in which libraries are loaded, while the actual loading order may not be statically known. Second, making a read-only variable writable does not break program functionality, while the opposite assumption could trigger page faults and crash the program. Compatibility is of paramount priority in software engineering practice and the current design of the dynamic loader favors compatibility over security.

How the change of memory protection affects security depends on the availability of memory corruption vulnerabilities and the type of the copied variables. Intuitively, if a program's security relies on the read-only protection of the moved variables, then copy relocations will increase the attack surface and/or even negate existing defenses. Copy relocations increase the program's attack surface because the adversary could potentially modify more program data than without copy relocation through memory corruption. As program defenses depend on the immutability of such data, adversaries may be able to circumvent defenses like control-flow integrity (e.g., based on vtables [41]), format string protection [14], or confused deputy mitigation [42].

IV. CoREV Implications

Unintended copy relocations that change memory protection are a new attack vector that enables several classes of attacks. Code pointers, format strings, and other static data assumed to be read-only by a defense or the program may suddenly be writable. We discuss these classes by examples.

A. Virtual Method Tables

C++ programs use a special data structure called *virtual method tables* (or *vtables*) to dispatch virtual functions for polymorphism. We show the conceptual memory layout of vtable data structures and the assembly code that makes virtual method calls in Figure 4. An object with virtual methods stores the pointer to the vtable at its beginning. The vtable is

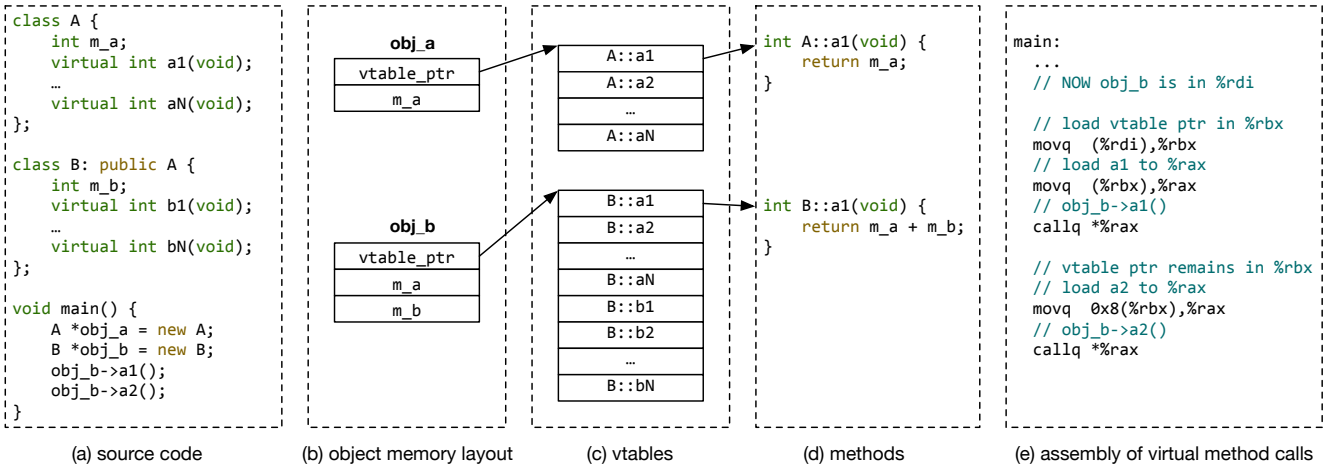

Fig. 4: vtable memory layout and virtual call sites.

```
main:
  ...
  // NOW obj_b is in %rdi

  // load vtable ptr to %rbx
  movq  (%rdi),%rbx
  // load a1 to %rax
  movq  (%rbx),%rax
  // obj_b->a1()
  assert %rax is valid target
  callq *%rax

  // vtable ptr remains in %rbx
  // load a2 to %rax
  movq  0x8(%rbx),%rax
  // obj_b->a2()
  assert %rax is valid target
  callq *%rax
```

```
main:
  ...
  // NOW obj_b is in %rdi

  // load vtable ptr to %rbx
  movq  (%rdi),%rbx
  // load a1 to %rax
  assert %rbx is a valid vtable
  movq  (%rbx),%rax
  // obj_b->a1()
  callq *%rax

  // vtable ptr remains in %rbx
  // load a2 to %rax
  movq  0x8(%rbx),%rax
  // obj_b->a2()
  callq *%rax
```

Fig. 5: Two types of instrumentations for protecting virtual method calls.

essentially an array of function pointers to the implementations of the virtual functions declared by its class (or its ancestor classes). Objects of different classes along the inheritance chain can have different implementations of the same virtual method (e.g., `A::a1` *vs* `B::a1`). Therefore, by embedding the vtable pointer into every object, the vtable can dispatch the virtual method calls based on the object's runtime type. Vtables are not supposed to change at runtime, thus they are statically initialized at compile time and allocated from read-only memory.

Attackers have demonstrated successful exploits by hijacking virtual method calls [38]. Broadly speaking, these attacks either corrupt the vtable in place or overwrite the vtable pointer stored in some object so that it points to something under the attacker's control. Current defenses [41] focus on the latter because the defenses (wrongly) assume that the vtable is allocated in a read-only section and cannot be modified by the attacker.

Furthermore, current defenses leverage the read-only nature of vtables to optimize their checks for better performance [41]. In Figure 5, we show two types of instrumentations for virtual call sites shown in Figure 4 (e). Both types of instrumentations aim to ensure that only valid methods can be targeted at each virtual call site. The first type of instrumentation directly checks the target function address obtained from the vtable. Alternatively, the second type of instrumentation assumes vtables are write protected, and checks if the referenced vtable is legitimate based on the object's static type.

There are two advantages of the second-type instrumentation compared to the first type. First, given that a vtable commonly contains multiple function pointers, there are fewer vtables than actual targets, making checking vtables more efficient. Second, if a program continuously makes virtual method calls on the same object and the compiler keeps the vtable pointer in a callee-saved register (e.g., `%rbx`), only one check is necessary when checking the vtable, while the first type of instrumentation needs to check on every virtual method call. As a result, researchers propose to check the vtable pointer to achieve better performance [41], [10]. Note that mechanisms using the second type of instrumentation must consider the security implications of spilling the register to the stack where it could potentially be overwritten [4].

However, vtables are not always read-only because of unintended copy relocations. This will render the defenses that are based on the second type of instrumentation ineffective, such as [41]. For instance, if an adversary corrupts the function pointers in the vtable but leaves the vtable pointer untouched, she can potentially redirect control flow to arbitrary code locations without detection.

To trigger vtables being moved to writable memory, a program must satisfy the following two invariants:

- A class having virtual methods is implemented in a dynamic library and its vtable is in the library's read-only region.
- The constructor of the same class is implemented in the executable. This is possible when a class does not have an explicit constructor or its constructor is defined in a header file.

We revisit the cause of copy relocations to illustrate why the two invariants lead to writable vtables. A copy relocation occurs when the executable references a symbol defined in an external dynamic library. In the case of vtables, the first invariant ensures that vtables are externally-defined symbols. The second invariant further ensures that there exists a symbol reference to the vtables in the executable because the constructor needs to initialize the object's memory including the vtable pointer (Figure 4 (b)). On the other hand, if the constructor is implemented as an external library function, the executable simply makes an inter-module call into the constructor for initializing objects, which eliminates the vtable reference from the executable.

B. Format Strings

Functions like `printf` use format strings as templates to direct outputs. Researchers have long known that such functions may be vulnerable if an adversary controls the format string input or the program uses directives that enable unauthorized memory accesses, which have become known as *format string vulnerabilities* [6]. More recently, researchers have shown that, by controlling the format string used in `printf`, an adversary can basically use the function as an interpreter and achieve Turing-complete computation [12] that evades control-flow defenses that they call *printf-oriented programming*.

One obvious defense against such vulnerabilities is to use static format strings. While such a defense may not always be possible, it is simple and encouraged where it is possible [14]. When enabled, the fortify gcc patch [27] enforces read-only format strings at the compiler level. The compiler-based check assumes that format strings allocated from read-only sections remain immutable (which is not true for CoREV). For example, printf-oriented programming requires that the adversary be able to modify the format string at runtime to implement branches in their attack. Thus, the current assumption is that the use of static format strings will prevent attacks on functions that use format strings.

Unfortunately, copy relocations can make format strings writable, enabling such attacks. Specifically, if a format string (i.e., a constant char array) is defined in a library and referenced from the executable, the loader will copy the entire string to writable memory and make it susceptible to memory corruption. Then, an adversary can implement printf-oriented programming simply by modifying the value of a supposedly static format string.

C. Other Static Data

Programs use a variety of other static data, such as static file names and IP addresses to utilize system resources unconditionally, such as program configuration files and well-known IP addresses like "*127.0.0.1*". Researchers have long been concerned about adversaries modifying the names of system resources accessed by programs. In general, various types of *confused deputy attacks* [24], such as link traversal [17] and Time-Of-Check-To-Time-Of-Use (TOCTTOU) [9], [28], enable an adversary to direct a vulnerable program to a resource of the adversary's choosing. Such an attack may enable an adversary to access a resource that is available to the victim, but not to the adversary (e.g., password files or secret key files). Alternatively, such attacks may enable an adversary to direct the victim to use an adversary-controlled resource instead (e.g., adversary-defined configuration or IP address). In these attacks, adversaries gain unauthorized access over victim resources or control inputs the victim depends upon.

Current defenses to prevent such attacks focus on filtering adversary-controlled file names [8] or restricting system calls that use adversary-controlled input in constructing file names [13], [35], [42]. For example, one recent defense identifies the data dependence between adversary-controlled inputs and the system call arguments that use them to restrict the resources accessible to system calls [42]. This approach uses a dynamic analysis to detect where system calls use adversary-controlled input in file names. However, such a dynamic analysis is unlikely to detect how to craft inputs necessary to overwrite file names that were thought to be immutable. As a result, when copy relocations make file names and other system resource names writeable, these will go undetected by such defenses.

V. Mitigation

The presented CoREV attack vector has existed for decades. We propose three fundamental mitigation approaches: (1) detection through a simple checker, (2) recompiling the underlying software as position independent code which does not require copy relocations (i.e., enabling a compiler switch), or (3) changing the toolchain to include additional information about the memory permissions of external symbols to enable permission-aware copy relocations. In addition, it is also possible to leverage source code annotations to eliminate copy relocations. We discuss the annotation approach in detail in Section VII-E when we evaluate other operating systems.

A. Detecting Permission Violations

A straightforward mitigation simply refuses execution of programs that violate the intended memory protection during copy relocations. The key idea is to *detect* copy relocations that violate permissions for any given executable. Our approach consists of three steps. First, we identify a list of symbols that are copied at runtime. This information is collected by parsing the relocation sections and identifying copy relocations from the program executable. Second, we locate the origins of those symbols. In this step, we parse the `.dynamic` section and follow the search order to enumerate dependent libraries. Finally, given a relocated symbol and a dependent library, the third step is to identify (i) whether the library defines the symbol and (ii) whether the symbol is in the library's read-only data section. If both are true, we report this memory protection violation and mark the program as potentially unsafe for execution.

B. Recompiling Software

Given that the static linker may not know the memory protection of referenced symbols in a dynamic library, a principled way to mitigate corruptions on read-only variables is to eliminate copy relocations. Therefore, the executable references external variables that reside in their original locations with untampered memory protections.

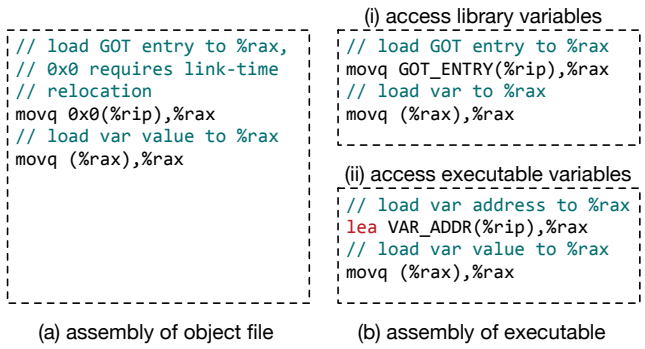

Fig. 6: Object files compiled with -fPIC and the resultant executable that accesses (i) library's globals and (ii) its own globals.

Recall that the motivation of copy relocations is to enable GOT-unaware code to reference external library variables. Therefore, to eliminate copy relocations, such references must go through the GOT, which is designed for serving inter-module references. To do so, we propose to change the compiler flags and add the `-fPIC` flag. This flag instructs the compiler to generate object files that are suitable for use in a dynamic library. It has two effects on the generated code. First, the code uses IP-relative addressing mode to access global variables. Second, the code accesses global variables through the GOT. We rely on the second effect to eliminate copy relocations. The `-fPIE` flag, which is typically used for creating position-independent executables (e.g., to support ASLR) as opposed to the `-fPIC` flag for creating dynamic libraries, cannot be used to eliminate copy relocations. This is because, the `-fPIE` flag only ensures the use of IP-relative addressing mode (the first effect), but does *not* force global variable accesses to go through the GOT (the second effect), which is key to avoiding copy relocations.

We show the generated instructions at the compilation stage in Figure 6 (a) to explain how the `-fPIC` flag helps eliminate copy relocations. For each symbol access, the compiler emits two memory accesses, where the first retrieves the address of the symbol from the GOT and the second actually loads its value. Consequently, at link time, the linker can allocate a GOT entry for variables that are externally defined in a dynamic library. This saves the linker from creating a copy relocation to cover the "mistake" of the clueless compiler.

One concern of this mitigation is the cost for accessing a global symbol. It seemingly incurs an unnecessary memory load operation for accessing globals that are defined within the executable, since their locations are statically known and hence do not require GOT indirections. In fact, on x86, the static linker can optimize the code sequence to save one memory access by changing the first instruction to be an `LEA` instruction, which simply computes the effective address instead of actually fetching the value from the memory (shown in Figure 6 (b)). This is possible because the `LEA` instruction has the same byte sequence as the corresponding `MOV` instruction except for the second byte in their opcodes.

However, despite this optimization, compiling executables as position independent does come with additional performance costs compared with copy relocations. First, accessing all library variables (including mutable variables that are not affected by CoREV) now require two memory accesses, while copy relocations only need one. Second, relative addressing mode can be costly especially for 32-bit x86 architecture where such a mode is not natively supported [33].

C. Adapting the Toolchain

The mapping between protection modifiers at the source code level and the protection enforced at runtime is crude. The protection modifiers change between languages and are mapped to read and write permissions at the linker and loader level. Especially for externally defined variables, this mapping can be imprecise. In the example shown in Figure 2, the declaration of variable `foo` has the keyword `const`. A dynamic library often has a header file that declares exported read-only variables in such a way, so that executables can reference them after including the header file.

Unfortunately, source-level protection information is *lost* when a source file is compiled into an object file (`*.o`), primarily due to how ELF specifies the memory protection for variables. Recall that ELF sets memory protections at the granularity of sections (see Section II). Thus, the way ELF specifies a variable as read-only is to allocate the variable from a read-only section (e.g., `.rodata`), so that the linker can preserve the intended memory protection when combining these sections into a single read-only section at link time. However, the compiler does *not* actually allocate externally-defined variables in object files. Instead, the compiler marks them as *undefined*. An undefined reference is insufficient for the linker to determine the originally intended memory protection settings for these variables.

Our proposed solution is to adapt the current toolchain to preserve such information along the compiler toolchain from source code to object files. To be compatible with the current ELF specification, we allocate a separate section (referred as CoREV section) in the object file to store memory protection information for each externally-defined variable. Specifically, each entry in the CoREV section contains a permission flag to specify the intended memory protection (i.e., read-only or read-write), as well as an ELF symbol index to specify the variable for which the permission flag applies. Therefore, based on the added information, the linker can create variable copies in corresponding data sections with respect to the originally intended memory protection.

Finally, we also adapt the dynamic loader so that it can perform copy relocations on read-only data sections by mapping them as writable during startup and protecting them as read-only afterwards. This process is similar to how the current dynamic loader handles the `.data.rel.ro` section as mentioned in Section VI. This approach requires changing the entire toolchain (i.e., compiler, linker, and loader), and we leave its prototype implementation to future work.

VI. CoREV Investigation

We have implemented two mechanisms to assess the new attack vector. Both tools are implemented in Python, using the

PyELF library to handle object files, with a total of 174 lines of code.

The first tool takes an executable as parameter and generates a list of copy relocations that may alter the memory protection set on the imported library variables. It has two components. The first part identifies all exported, read-only variables in a given dynamic library. Specifically, for each exported library variable, we check whether it resides in a read-only data section (e.g., `.rodata`). One subtlety here is that some *writable* data sections (e.g., `.data.rel.ro` and GOT) can be reprotected as read-only at runtime through the RELRO program header [2]. The dynamic loader implements functionality to handle relocations on read-only data by (1) grouping them into a dedicated section so that the other read-only data without relocations can still be shared among processes, and (2) resolving the relocations during startup and then remapping the writable section to read-only afterwards. We treat those sections as read-only (ignoring the small window for a TOCTTOU attack during startup). The second component enumerates all copy relocations in an executable. For each copy relocation, we iterate through each of the executable's dependent libraries (i.e., extracted from the executable's `.dynamic` section) and check whether the copied variable is present and read-only in the library. If so, we report this copy relocation as potentially unsafe.

The second tool infers the data type for a given library variable. To enable our tool to analyze arbitrary libraries, we do not require source code or debug information. Instead, we infer data types using binary analysis. We classify the symbols into seven categories: (i) C++ vtables, (ii) function pointers, (iii) generic pointers, (iv) format strings, (v) file and path names, (vi) generic strings, and (vii) other variables.

First, we broadly infer pointers and strings in the dynamic libraries. To discover pointer variables, we rely on relocation information in the library. Specifically, given a dynamic library can be loaded at an arbitrary address at runtime, a pointer in the data sections must be properly patched to run. That said, each pointer in the dynamic library will have a corresponding relocation entry. For example, an `R_X86_64_RELATIVE` entry instructs the loader to add the loading base address to a pointer variable so that it points to the correct location at runtime. Furthermore, for the discovered pointers, we check whether they point to code or data, and classify them into function pointers and data pointers accordingly (see below). A pointer can be part of a composite variable such as a structure or an array. For simplicity, we classify the entire variable into the pointer category as long as one of its fields is a pointer.

To determine strings, we check whether the variable contains only ASCII characters and if it is `NULL`-terminated. We further identify format strings and file names from discovered string variables (see below). We highlight format strings and file names because memory corruption on them could lead to security breaches.

Next, we discuss how we infer and classify the types of exported library variables in detail:

C++ vtables: if the variable contains a set of code pointers and is named through standard name mangling rules (e.g., if the variable name starts with `_ZTV`);

Function pointers: if the variable is a pointer and the pointer references a code segment;

Generic pointers: if the variable is a pointer into the current library or relocated to a different library and not a function pointer (i.e., references data);

Format strings: if the variable is a string and contains at least one format specifier (%);

File names and paths: if the variable is a string and contains at least one path separator (/);

Generic strings: if the variable is a string and is neither a format string nor a filename/path;

Other variables: all other variables.

While the current prototype uses simple heuristics, they work well in practice. We currently restrict automatic type discovery through heuristics and binary analysis as debug information is not always available. In future work we will evaluate further heuristics and approaches for identifying a broader set of data types, e.g., by using debug information (whenever available).

VII. Evaluation

A. Attack Surface

In this section, we study the distribution of read-only variables in dynamic libraries of a real Linux distribution. Specifically, we collect all packages available to Ubuntu 16.04 LTS and identify the exported read-only variables for each dynamic library. In theory, all these variables can be relocated into writable data memory at runtime if they are referenced by an executable; however, in practice, not all of them may be accessed. First, not all read-only variables in dynamic libraries are equally likely to be referenced by executables. For example, if the constructor of a C++ class is implemented in the library (see Section IV-A), it is highly unlikely for the executable to *directly* reference the vtable of the particular class. Second, if a library is used by an executable that is compiled with `-fPIC` and hence does not have copy relocations, the read-only variables in the library will not be relocated at runtime. Thus, we treat the set of exported read-only variables as a theoretical upper bound of the attack surface for CoREV attacks, and evaluate the actually relocated read-only variables in Section VII-B.

We analyzed 58,862 libraries from 54,045 packages. Among them, 29,817 libraries export read-only variables. In total, we found 5,114,127 exported read-only variables across these libraries, making an average of 86.9 such variables per library (including those that do not export any read-only variable). We show the number of exported read-only variables in each dynamic library in Figure 7 (sorted from low to high). There are 55 libraries with over 10,000 exported read-only variables. For example, the main library (`libxul.so`) used by Firefox browser has over 40,461 exported read-only variables. Fortunately, the Firefox executable is compiled with `-fPIC` and has no copy relocations. However, a broad attack surface for potential CoREV-based memory corruption attacks remains should another executable uses the same library but is not compiled as PIC.

We further classify these read-only variables based on their inferred data types. In our analysis we distinguish between C++ vtables, function pointers, generic pointers, format strings,

Variable Type	Attack Surface			Vulnerable		
	# variables	# libraries	# packages	# variables	# executables	# packages
C++ vtables	714,617	14,563	3,692	28,497	4,291	1,609
function pointers	115,071	1,054	541	711	105	78
generic pointers	694,846	12,118	3,830	33,057	4,910	2,082
format strings	874	161	107	24	14	12
file names	6,822	454	252	44	20	10
generic strings	654,429	13,220	4,145	1,347	197	108
others	2,927,468	19,437	5,185	5,418	1,890	671

TABLE I: Potential attack surface and vulnerable subset of variables for all available Ubuntu 16.04 packages. Under the Attack Surface column, we list the number of exported read-only variables, involved libraries and packages. Similarly, under the Vulnerable column, we list the number of actually copied read-only variables, involved executables and packages.

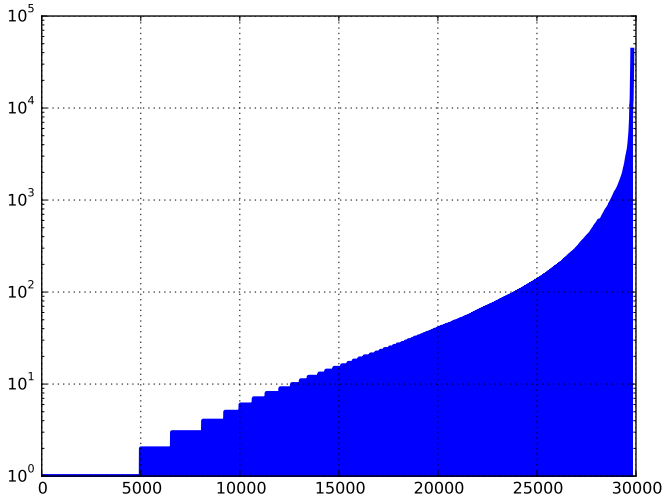

Fig. 7: Number of exported read-only variables in each dynamic library from all Ubuntu 16.04 packages (sorted from low to high). X-axis denotes each individual library and Y-axis indicates the number of exported read-only variables.

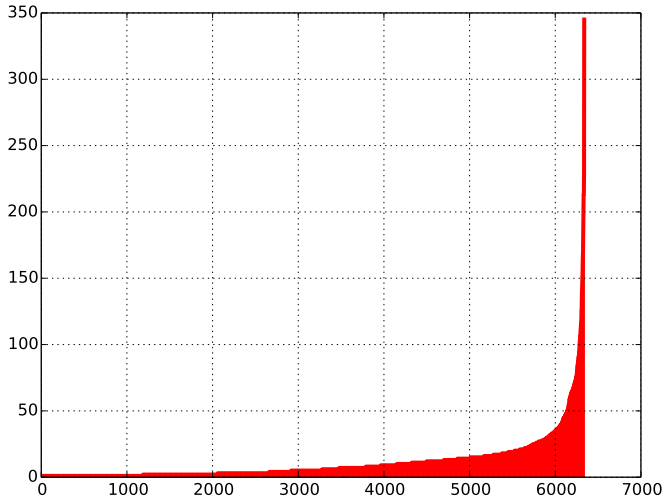

Fig. 8: Number of actually copied variables per executable that are vulnerable to CoREV from all Ubuntu 16.04 packages (sorted from low to high). X-axis denotes individual executables and Y-axis indicates the number of CoREVs.

file names, generic strings, and other variables. We single out vtables, function pointers, format strings, and file names due to the security implications if their permissions are changed through a copy relocation.

We show the results in the "attack surface" column in Table I. For the exported read-only variables with inferred types, C++ vtables are the majority – 32.68% of all the variables with inferred types. Function pointers occupy another 5.26%. These code pointers are often of interest to an adversary because corrupting code pointers may give her arbitrary control of the program execution [37], [38], [23], [18]. In the case of CoREV, these attacks are possible despite strong defenses as we show in Section VII-D.

Generic pointers have the second largest population (31.78%). Programs use generic pointers to access memory indirectly. Therefore, if a constant pointer becomes modifiable, an adversary could trick the program to access something vastly different that is under her control. The security implication depends on what the constant pointer points to. For example, if the constant pointer points to a format string, an adversary can then corrupt the constant pointer to point to a malicious format string to trigger printf-oriented programming. As mentioned in Section VI, our current type inference does not follow generic pointers, and we leave a more proactive type discovery to future work.

Format strings and file names are also exported by libraries, although many fewer are relocated in comparison to other types. Writable format strings allow an attacker to execute arbitrary computation and file names allow an attacker to possibly change the input and output of the program.

B. Real-World Permission Violations

In this section, we study the real-world programs that have unsafe copy relocations in Ubuntu 16.04 LTS. We have examined 34,291 executables across 54,045 packages. 6,339 of these executables have 166,543 copy relocations in total, among which, 69,098 alter the memory protection. In Figure 8, we display the number of CoREVs for each executable in all Ubuntu 16.04 packages. There are 54 executables that have more than 100 CoREVs (with a maximum of 345 CoREVs).

Variable	# Copies
(V) __cxxabiv1::__si_class_type_info	3,676
(V) __cxxabiv1::__class_type_info	2,988
(V) std::basic_ios⟨char, std::char_traits⟨char⟩⟩	1,842
(V) std::basic_streambuf⟨char, std::char_traits⟨char⟩⟩	1,819
(V) __cxxabiv1::__vmi_class_type_info	1,641
(V) std::__cxx11::basic_stringbuf⟨char, std::char_traits⟨char⟩, std::allocator⟨char⟩⟩	1,319
(T) std::exception	1,169
(T) std::runtime_error	1,020
(V) std::basic_filebuf⟨char, std::char_traits⟨char⟩⟩	953
(V) std::__cxx11::basic_ostringstream⟨char, std::char_traits⟨char⟩, std::allocator⟨char⟩⟩	894

TABLE II: The ten most commonly copied read-only variables in Ubuntu 16.04. They are all from the libstdc++ library. (V) denotes vtable and (T) denotes typeinfo. The "copies" column list the number of executables that actually copy the corresponding variable.

mysql-workbench: library/forms/mforms/container.h

```
class Container : public View {
public:
    Container() {}
    virtual void set_padding(...);
    virtual void set_back_image(...);
};
```

Fig. 9: An example of C++ vtable that is copied to mysql-workbench executable.

This experiment shows that CoREVs do commonly exist in real-world programs and present a real threat to the ELF-based dynamic linking procedure. However, an unsafe copy relocation is not exploitable by itself. Instead, it provides an adversary with more potential corruption targets to launch attacks and/or bypass existing defenses.

We list the types of these unsafe copy relocations in Table I in the "vulnerable" column. In particular, 44.75% of all relocated read-only variables with discovered types are C++ vtables. This is proportional to the exported C++ vtables listed in the attack surface, and makes CoREV an unignorable problem because it enables attacks that can potentially evade current defenses as shown in Section VII-D.

Finally, we study the common CoREVs in these executables. Surprisingly, the top 10 most commonly copied variables are all from libstdc++. We list them in Table II. Among the 10 CoREVs, 8 of them are actually vtables of widely used classes. For example, the vtable of class `__cxxabiv1::__si_class_type_info` are copied by 3,676 executables in Ubuntu 16.04. Given the prevalent use of libstdc++, it is likely that future C++ programs can also be susceptible to CoREV-based attacks.

C. Case Study

In this section, we study how CoREVs occur in real-world programs.

gettext-0.19.7

```
struct catalog_input_format {
    void (*parse)(…);
    bool produces_utf8;
};

const struct catalog_input_format
input_format_properties = {
    properties_parse,
    true
};
```

Fig. 10: A system library with read-only function pointers that are relocated to various executables.

1) C++ Vtable (mysql-workbench): We use mysql-workbench, a complex, GUI-based, network-facing C++ application, as an example of of C++ vtable relocations. Note that we have found many other C++ applications with such unsafe relocations that are not listed here. The mysql-workbench is a unified visual tool for database management that divides its functionality into multiple dynamic libraries. As a result, it relocates 19 vtables from 5 different libraries in total. We show one of them in Figure 9.

Container is a class implemented in the libmform library and serves as a base class for graphic components such as MySQL table. It defines two additional virtual functions, thus the Container class has a corresponding vtable for dispatching virtual method calls. In addition, given that the Container class does not have an explicit constructor, the mysql-workbench executable will define a default one. The default constructor will need to reference Container's vtable for initializing object memory. Consequently, the vtable is relocated to the executable's `.bss` section and becomes writable, potentially mitigating vtable-based defenses.

2) C++ Vtable (apt-get): We found that apt-get also contains a set of five vtables that are copy relocated from libstdc++ and libapt-pkg.so.5.0. Four of them overlap with the already mentioned vtables in Table II, the last one is the vtable of class OpTextProgress in libapt-pkg. Assuming that apt-get is compiled with an upcoming control-flow hijacking defense like VTV [41] and that a memory corruption vulnerability exists, the adversary may use CoREV to bypass such defenses.

3) Function Pointer (gettext): We use the gettext library to show how constant function pointers are copied in a typical program. The gettext library is in the gettext-base package which is installed on every Ubuntu 16.04 machine. It exports 6 read-only function pointers and affects 15 built-in executables. We show one of its exported function pointers in Figure 10. The exported function pointer is actually a field of a structure. Each structure corresponds to an input stream format and the function pointer points to an internal library function that parses the format. Thus, a program can leverage it to process the input stream based on its own needs. Similarly, these constant data structures defined in gettext become modifiable due to copy relocation and hence susceptible to memory corruption attacks.

libow-3.1: src/c/error.c

```
const char mutex_unlock_failed[] =
            "mutex_unlock failed rc=%d [%s]\n";
```

libow-3.1: src/include/ow_mutex.h

```
extern const char mutex_unlock_failed[];

#define my_pthread_mutex_lock(mutex) \
  do { \
    /* skip some code here */
    mrc = pthread_mutex_lock(mutex); \
    if (mrc != 0) {
      vsprintf(buf, mutex_unlock_failed, ...); \
    } \
  } while (0)
```

Fig. 11: An example of format string that is susceptible to CoREV in libow-3.1.

main.cpp

```
#include <stdio.h>
#include "A.hpp"

// hardcoded symbol for class A's vtable
extern unsigned long _ZTV1A[];

void hijack(void) {
    printf("vulnerable!\n");
}

int main() {
    // corrupt A's vtable slot for
    // method a1 as if an attack happens
    _ZTV1A[2] = (unsigned long) hijack;

    // allocate an object and make
    // the virtual call
    A *obj = new A();
    obj->a1();

    return 0;
}
```

A.hpp

```
class A {
public:
    virtual int a1();
};
```

A.cpp

```
#include "A.hpp"

int A::a1() {
    return 1;
}
```

Makefile

```
all: main.cpp A.cpp
  g++ -fPIC -shared \
    A.cpp -o libA.so
  g++ -L. -lA \
    main.cpp
```

Fig. 12: A test program for vtable defenses.

4) Format String (libow): We identified a dynamic library (libow-3.1 [1]) that exports a set of 22 format strings, causing three different executables (owftpd – an ftp server, owserver – a backend server for 1-wire control, and owexternal) to copy them into writable memory at runtime. These format strings are for debugging purposes, and we show one of them in Figure 11. The format string `mutex_unlock_failed` is defined in the libow-3.1 library and exported in a header file. As a result, an executable that includes the header file and uses the library-provided macro `my_pthread_mutex_lock` will cause the format string to be relocated. If an adversary corrupts the relocated format string and exploits a concurrency bug to cause the mutex lock operation to fail, she can potentially launch printf-oriented programming and achieve arbitrary code execution.

D. Affected Defenses

In this section, we evaluate how unsafe copy relocations affect current defenses.

Defenses	check fptr	check vtable	CoReV?
VTrust [43]	✓	✓	×
VTV [41]	×	✓	✓
vfGuard [36]	✓	×	×
Interleaving [10]	×	✓	✓
SafeDispatch [26]	✓	×	×
SafeDispatch (2)	×	✓	✓
RockJIT [30]	✓	×	×

TABLE III: Evaluation of vtable defenses, whether they check function pointers, vtables, or both. Three defenses [41], [10], [26] assume vtables are write-protected and only check the vtable pointer, thus are affected by CoReV and may allow vtable corruption attacks.

Recent research proposals on control-flow integrity (CFI) focus on protecting forward edges (i.e., indirect calls/jumps) [41], [43], [44], [10]. In particular, given the prevalence of virtual method calls in C++ programs (see Section IV-A), researchers have proposed many defenses to protect these dynamic calls. To evaluate how they are affected by unsafe copy relocations, we come up with a simple exploitation test as shown in Figure 12.

The test program has two parts. The first part is a library which defines a class `A` with a virtual method (`A.hpp` and `A.cpp`). The second part is an executable which allocates an instance `obj` of the class and invokes the virtual method (`main.cpp`). In the executable, we hard-code the symbol name used for `A`'s vtable (`_ZTV1A`) and perform an emulated memory corruption on the function pointer of the virtual method `A::a1` in `A`'s vtable and see if it is accepted by the evaluated defense deployed on this program.

We evaluate a set of six CFI defenses and show the results in Table III. We choose these defenses because they apply to C++ programs on Linux. An effective defense must check the pointer to the vtable to defend against COOP attacks [38] but must also check the actual value of the function pointer in the vtable to defend against CoREV. Only defenses that check both targets protect against CoREV and COOP.

Among the defenses, three are vulnerable to unintended copy relocations because they assume read-only protection for vtables and only check if the vtable pointer points to a valid vtable (see Section IV-A). Interleaving [10] does not currently support dynamic linking so we evaluate its vulnerability based on their proposed instrumentation. SafeDispatch [26] proposes two different instrumentations where the first checks the virtual method target and the other checks the vtable pointer. It claims the latter provides better security regarding COOP-style attacks [38]. However, the latter is vulnerable to CoREV-based vtable corruption attacks.

E. Other Platforms

Dynamic linking is enabled by default on major operating systems such as Windows and macOS. We evaluate CoREV implications on the dynamic linking implementations on both Windows and macOS.

Recall that the cause of copy relocations is due to the ambiguity in declaring external variables at source level (see

Figure 2). Compilers cannot know whether these variables are defined by another object file in the same binary or in a dynamic library. How compilers handle this ambiguity results in performance or security implications. To understand CoREV on other platforms, we compile the program in Figure 2 and examine the generated instructions that access the external variable `foo`.

1) Windows: The MSVC linker on Windows refuses to build the example program. This is because the MSVC compiler requires the program to explicitly specify an external library variable using the `__declspec(dllimport)` attribute in addition to the `extern` keyword. Otherwise, the declared external variable is assumed to be defined in another object file that links to the same binary. If the linker cannot find the symbol definition when performing static linking, it will report an error rather than creating a copy relocation. Thus, Windows removes the ambiguity by forcing declarations to be explicit, removing the need for copy relocations. Through these annotations, the MSVC compiler can achieve high performance for symbols in the same module and keep permissions for symbols in other modules. Hence CoREV does not affect Windows.

2) macOS: macOS handles the ambiguity by making the opposite assumption of Linux, trading performance for safety. The compiler assumes all variables declared as external are potentially from dynamic libraries, and generates instructions in Figure 6 (a) for external variables. This is the mitigation approach we proposed in Section V-B. Naturally, accesses to library variables use the GOT indirection. Consequently, copy relocations do not exist on macOS and CoREV does not affect macOS either.

However, memory corruption over "read-only" data is still possible on macOS. Specifically, based on our observation, the compiler allocates read-only data that potentially requires runtime relocation from the `__DATA.__const` section. For example, code pointers in vtables may require adding the module loading base at runtime, and thus are allocated from the `__DATA.__const` section. This supposedly read-only section is, however, mapped as read-write at runtime. We tested the example program in Figure 12 and examined several system libraries such as `libc++.dylib` on macOS 10.12. This design simplifies the implementation of the dynamic loader. If the relocated data (i.e., the code pointers) resides on writable pages, the loader can freely patch relocations at any time without worrying about page faults. Unfortunately, similar to the security concerns raised by CoREV, this design weakens the security of applications by exposing memory corruption targets to an adversary, enabling her to launch attacks and/or bypass defenses.

VIII. Conclusion

Dynamic loading enables modularity and reduces the memory footprint of applications. Due to the incomplete mapping between source level primitives (like `extern const`) and imported and exported symbols on the ELF/binary level, memory protection information is inadvertently lost. When an executable references a read-only variable exported from a library, the dynamic loader `ld.so` relocates this variable into the writable `.bss` section of the executable, which effectively strips the const attribute specified by the programmer. As a result, this enables an adversary to modify "read-only" variables when exploiting a memory corruption vulnerability. We call this attack CoREV for Copy Relocation Violation. This attack vector has existed for decades and, as we show for Ubuntu 16.04, is widespread. The attack surface is broad with 29,817 libraries exporting relocatable read-only variables. The set of 6,399 programs with actual unsafe copy relocations includes ftp servers, apt-get, and gettext out of 4,570 packages.

An attacker can use CoREV to escalate her privileges, leveraging a memory corruption vulnerability to modify format strings, file names, vtables, code pointers, or other supposedly read-only data. We discuss three possible mitigation strategies that (i) detect the attack vector by analyzing binaries and libraries – as a fast mitigation, (ii) mitigate the attack through recompilation (if possible), or (iii) change the toolchain to make the linker and loader aware of source level permissions even for externally defined variables.

Acknowledgement

We thank our shepherd, Engin Kirda, and the anonymous reviewers for their constructive feedback on this work. The work was supported, in part, by the National Science Foundation under grants number CNS-1408880, CNS-1513783, and CNS-1657711. The research reported here was supported in part by the Defense Advanced Research Projects Agency (DARPA) under agreement number N66001-13-2-4040. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[1] libow-3.1. http://packages.ubuntu.com/xenial/libow-3.1-1.

[2] RELRO - a memory corruption mitigation technique. http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "A theory of secure control flow," in *Proceedings of the 7th International Conference on Formal Methods and Software Engineering*, ser. ICFEM'05, 2005.

[5] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies," 2004.

[6] S. V. Archives, "Wu-ftpd remote format string stack overwrite vulnerability," 2008.

[7] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1342–1353.

[8] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE Symposium on Security and Privacy (Oakland 2008)*. IEEE, 2008, pp. 387–401.

[9] M. Bishop and M. Digler, "Checking for race conditions in file accesses," *Computer Systems*, vol. 9, no. 2, Spring 1996.

[10] D. Bounov, R. Kici, and S. Lerner, "Protecting c++ dynamic dispatch through vtable interleaving," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[11] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, 2017.

[12] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th Usenix Security Symposium (USENIX Security)*, 2015.

[13] S. Chari, S. Halevi, and W. Venema, "Where do you want to go today? escalating privileges by pathname manipulation." in *Network and Distributed System Security Symposium (NDSS)*, 2010.

[14] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "Formatguard: Automatic protection from printf format string vulnerabilities." in *USENIX Security Symposium (USENIX Security)*, vol. 91. Washington, DC, 2001.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy (S&P 2015), 18-20 May 2015, San Jose, California, USA*, 2015.

[16] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014, pp. 292–307.

[17] CWE, "CWE-59: Improper Link Resolution Before File Access," http://cwe.mitre.org/data/definitions/59.html.

[18] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 401–416.

[19] U. Drepper, "How to write shared libraries," *Retrieved Jul*, vol. 16, p. 2009, 2006.

[20] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding control flows using intel processor trace," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017.

[21] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *IEEE European Symposium on Security and Privacy (EuroSP)*. IEEE, 2016.

[22] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," in *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST 2014)*, May 2014.

[23] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, May 2014.

[24] N. Hardy, "The confused deputy," *Operating Systems Review*, vol. 22, pp. 36–38, 1988.

[25] E. Hiroaki and Y. Kunikazu, "ProPolice: Improved stack-smashing attack detection," *IPSJ SIG Notes*, pp. 181–188, 2001.

[26] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: Securing c++ virtual calls from memory corruption attacks." in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[27] J. Jelinek, "FORTIFY_SOURCE," https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html, 2004.

[28] W. S. McPhee, "Operating system integrity in OS/VS2," *IBM Syst. J.*, 1974.

[29] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, p. 58.

[30] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGPLAN Conference on Computer and Communications Security*. ACM, 2014, p. 58.

[31] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[32] PaX Team, "Documentation for the PaX project - overall description," https://pax.grsecurity.net/docs/pax.txt, 2008.

[33] M. Payer, "Too much PIE is bad for performance," ETH Zurich Technical Report http://nebelwelt.net/publications/files/12TRpie.pdf, 2012.

[34] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[35] M. Payer and T. R. Gross, "Protecting Applications Against TOCTTOU Races by User-Space Caching of File Metadata," in *VEE'12: Proc. 8th Int'l Conf. Virtual Execution Environments*, 2012.

[36] A. Prakash, X. Hu, and H. Yin, "vfGuard: Strict protection for virtual function calls in cots c++ binaries." in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[37] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[38] 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 *2015 IEEE Symposium on Security and Privacy (Oakland)*. IEEE, 2015, pp. 745–762.

[39] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

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

[41] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *USENIX Security Symposium*, 2014.

[42] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "JIGSAW: Protecting resource access by inferring programmer expectations," in *23rd USENIX Security Symposium (USENIX Security)*, 2014, pp. 973–988.

[43] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "VTrust: Regaining trust on virtual calls," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[44] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting virtual function tables' integrity." in *Network and Distributed System Security Symposium (NDSS)*, 2015.