UNIVERSITY OF CALIFORNIA SAN DIEGO

Compilers and Software Security: Opportunities and Challenges

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Zhaomo Yang

Committee in charge:

Professor Kirill Levchenko, Chair
Professor Farinaz Koushanfar
Professor Sorin Lerner
Professor Stefan Savage
Professor Hovav Shacham

2019

The Dissertation of Zhaomo Yang is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2019

DEDICATION

Dedicated to my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGEMENTS

I would like to acknowledge all of my thesis committee members for their help and time. Especially, I would like to thank my adviser Prof. Kirill Levchenko for his support and guidance in the past few years, Prof. Sorin Lerner for his great compiler class, which has significant influence on this dissertation, and Prof. Hovav Shacham for his inspirations.

I would also like to acknowledge my friends and collaborators in the CSE department. It is their support that helped me in an immeasurable way. I would like to especially acknowledge Vector Lee and Nishant Bhaskar for their generous help at my defense.

Chapter 1, in part, is currently being prepared for submission for publication of the material. Yang, Zhaomo; Collingbourne, Peter; Levchenko, Kirill. The dissertation author was the primary investigator and author of this material.

Chapter 2 is an adpated reprint of the material as it appears in Dead Store Elimination (Still) Considered Harmful in USENIX Security 2017. Yang, Zhaomo; Johannesmeyer, Brian; Olesen, Anders T.; Lerner, Sorin; Levchenko, Kirill., Proceedings of the USENIX Security, 2017. The dissertation author was the primary investigator and author of this paper.

VITA

2011	Bachelor of Science, Beijing University of Technology, Beijing, China
2013	Masters of Science, University of California, San Diego, USA
2014–2019	Research Assistant, University of California, San Diego, USA
2019	Doctor of Philosophy, University of California, San Diego, USA

ABSTRACT OF THE DISSERTATION

Compilers and Software Security: Opportunities and Challenges

by

Zhaomo Yang

Doctor of Philosophy in Computer Science

University of California San Diego, 2019

Professor Kirill Levchenko, Chair

Compilers play a critical role in software security. On the one hand, compilers are an ideal place to secure software against some classes of vulnerabilities due to their knowledge of the programs under protection and little developer efforts they require. Exploiting this opportunity, I designed and implemented a highly efficient compiler-based Control-Flow Integrity (CFI) scheme for C++ virtual calls. My scheme introduces minimal performance and space overhead even for programs that use virtual calls heavily, which makes it more likely to be deployed to real-world programs that have strict requirements on performance and code size.

On the other hand, compilers can also be detrimental to software security. Dead Store Elimination (DSE) is a well-known compiler optimization that may weaken the security of a

program by accidentally removing memory scrubbing operations for sensitive data. Security-savvy developers have long been aware of this phenomenon and have devised ways to prevent the compiler from eliminating these data scrubbing operations. To understand the current state of this problem, I examined the existing techniques found in the wild to circumvent the elimination of scrubbing operations and investigated eleven security-critical open source projects on this issue. The results are surprising: even for such a known issue, only three out of eleven open source projects I surveyed had a reliable scrubbing method and used it consistently.

Introduction

Compilers are a fundamental part of software development. In the process of translating a high-level programming language into a lower-level format, the compiler has significant impact on many qualities of the generated code such as correctness, performance, size, and security. In this dissertation, We explore the impact compilers have (or may have) on software security.

On the one hand, compilers could have great positive impact on software security. From a security engineer's perspective, the compiler presents great opportunities to implement security mechanisms because it has access to the program under protection, and many built-in program analyses (e.g. points-to analyses) of the compiler are useful for implementing and optimizing security mechanisms. Also, from the software developer's perspective, compiler-based security schemes are attractive because they are easy to use: they can often be enabled with simple compiler flags and offer complete protection against certain types of vulnerabilities. Compiler-based security schemes have been particularly popular for programs written in memory-unsafe languages such as C and C++ where a simple out-of-bounds write vulnerability may allow the attacker to take complete control of the program. For example, stack canaries, a mechanism to ensure that the return address has not been corrupted before a function returns, was first proposed about 20 years ago, and today more than 80% of the top 20 packages with the most reverse dependencies on Ubuntu 18.04 are built with stack canaries [18]. Control-Flow Integrity (CFI) is another popular security mitigation for control-flow hijacking in C and C++. At a high-level, CFI restricts targets of indirect jumps and returns to a small set of allowed targets. CFI is becoming increasingly popular, with some variant of CFI integrated into GCC and LLVM [58]. Today, Google, for example, uses CFI to secure Chrome and multiple system components of

Android [10] [30]. However, not all programs can enjoy the advances in compiler-based security mechanisms. Because compiler-based mechanisms tend to be optimized for performance at the cost of space overhead, programs running on resource-constrained systems cannot always take advantage of them. At the same time, more and more resource-constrained systems running software written in C and C++ are coming to our lives, so there is an urgent need to secure them. In Chapter 1, we investigate the opportunities to harden software on resource-constrained systems. Particularly, we focus on virtual calls in C++, a unique type of dynamic control transfer in C++ that is subject to control-flow hijacking. Virtual calls are normally implemented using a data structure called virtual table (vtable), which contains metadata and function pointers. Inspired by the work of Bounov *et al.* [39] that creatively secures virtual calls by interleaving vtables together, we developed and implemented a highly efficient CFI scheme for virtual calls. Unlike the original work [39], our scheme interleaves C++ vtables at the vtable entry level, which can keep more properties in the interleaved layout. This makes our scheme more compatible with the existing programs and libraries. In addition, interleaving vtables at a finer granularity further drives down space overhead, because the scheme can eliminate unused vtable entries and virtual functions. In fact, our scheme can often even reduce the size of a program, making it possible to enable additional space-consuming security schemes. Moreover, our scheme inserts more efficient CFI checks and has higher protection precision.

Compilers may also impact software security negatively. In the famous Turing award lecture "Reflections on Trusting Trust" [57], Ken Thompson warned that a malicious compiler could stealthily inject backdoors into victim programs. Setting aside malicious and bug-ridden compilers, benign correctly-implemented compilers may still surprise the developer by undermining security-related code she puts in place. For example, researchers have found that compilers may weaken the security of programs by undefined behaviours [59] or destroy the constant-time property of the code [38]. In Chapter 2, we examine the current state of a well-known issue of this category. Due to the concerns over memory disclosure vulnerabilities in C and C++ programs, security application developers explicitly scrub sensitive data from memory after use.

However, compiler optimizations such Dead Store Elimination (DSE) may deem such scrubbing operations useless since the values they set are never used and thus remove them. Security-savvy developers have long been aware of this phenomenon and have devised ways to prevent the compiler from eliminating these data scrubbing operations. To understand the current state of the problem, in Chapter 2 I examined the existing techniques found in the wild to circumvent the elimination of scrubbing operations and investigated eleven security-critical open source projects on this issue. The results are surprising: even for such a known issue, only three out of eleven open source projects we surveyed had a reliable scrubbing method and used it consistently.

The remaining dissertation is organized as follows. In Chapter 1, we describe a highly efficient compiler-based security scheme for protecting dynamic dispatch in C++ programs. In Chapter 2, we present a systematic analysis of the scrubbing problem, including an in-depth analysis of the existing techniques to prevent scrubbing removal and a case study of open source security programs on this issue. Finally, we conclude the dissertation in Chapter 3 and discuss future directions of research.

Chapter 1

Opportunities

Resource-constrained systems like Internet-of-Things (IoT) devices are increasingly common in our lives. C++ is a popular programming language for such resource-constrained systems due to its powerful features and the availability of the libraries and frameworks in C++. However, C++ is not a memory-safe language, and thus many of these systems are potentially subject to memory corruption, or even control-flow hijacking when an attacker manages to manipulate control data. C++ programs make control-flow hijacking easier due to the proliferation of virtual calls, whose control data is stored in writable memory.

Many security schemes have been proposed to mitigate control-flow hijacking via virtual calls. However, most of the schemes are optimized for performance overhead at the cost of space overhead, making them unusable for resource-constrained systems where storage space is very limited. Inspired by the original interleaving scheme of Bounov *et al.* [39], we designed and implemented an advanced virtual vtable interleaving scheme that interleaves only necessary table entries together and does efficient CFI checking for virtual calls. Compared with the original interleaving scheme, our scheme has lower space overhead, better compatibility with existing programs and libraries, and higher protection precision.

1.1 Introduction

The recent years have witnessed the rapid growth of resource-constrained systems including mobile devices and Internet-of-Things (IoT) devices. C++ is one of the most popular programming languages for these systems thanks to its speed and object-oriented features. From popular apps on smartphones, to simple programs running in smart sensors and actuators, to complex IoT gateways, C++ is everywhere.

Deploying C++ at such a scale, however, has serious security implications. Since C++ is not memory-safe, memory corruption bugs are not uncommon. Also, C++ programs often contain many computed control transfers that rely on control data in writable memory to compute the transfer destination. Computed control transfers can be divided into two categories: forward transfers (e.g. transfers via function pointers) and backward transfers (e.g. transfers via `ret` instructions). If the attacker manages to corrupt the control data of a computed transfer by, for example, exploiting a memory-safety bug, they can hijack the control flow of the program.

Researchers have long been aware of the danger of control-flow hijackings in memory-unsafe languages and have thus proposed numerous security mitigations including DEP [17] and ASLR [26] for general mitigation, stack canaries [41] and shadow stacks [34] for protecting backward transfers, and control-flow integrity (CFI) [37] for forward transfers. These techniques have been gaining popularity due to their efficacy and ease of use. They have not only become available in major compilers like GCC and LLVM, but are also deployed to secure important real-world software. For example, more than 80% of the top 20 packages with the most reverse dependencies on Ubuntu 18.04 are built with stack canaries [18], and Google builds some components of Android [30] and the Linux version of Chrome [10] with CFI enabled by default.

However, the advances in the control-flow hijacking mitigations have not fully benefited resource-constrained systems. A major obstacle for deploying existing mitigations on resource-constrained systems is their space overhead. Existing security mitigations, especially CFI schemes, are optimized for performance at the cost of space overhead. This approach is

reasonable and effective for environments where memory and storage space are sufficient, but makes the mitigations non-starters for resource-constrained systems. With the ever-growing number of mobile and IoT devices running C++ programs, there is an urgent need for security mitigations optimized for both performance overhead and space overhead.

In this chapter, we focus on protecting virtual calls, a type of forward computed transfer unique to C++. Previously, Bounov *et al.* [39] proposed a novel CFI scheme for virtual calls that achieves both low performance overhead and low space overhead by interleaving virtual tables (vtables) of a program together. Inspired by this work, we push the interleaving idea a step further: instead of interleaving whole vtables in a program, we only interleave vtable entries that are actually used. This allows us to further drive down the space overhead significantly, making it usable for resource-constrained systems and potentially enabling other security mitigations that require space. What's more, interleaving vtables at a finer granularity allows us to keep more properties in the interleaved layout, which makes our scheme more compatible with the existing programs and libraries. In addition, our scheme offers better protection precision and uses more efficient checks compared to the original interleaving scheme due to our more advanced vtable ordering algorithm. In this chapter, we:

* present an in-depth analysis of the relationship between C++ types and their vtable layouts and a new structure called vtable layout inheritance tree that accurately captures this relationship among types.

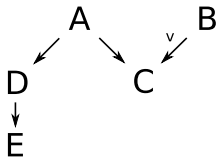
* present a fine-grained vtable interleaving algorithm, which can be used alone as an optimization pass to eliminate unused vtable entries and virtual member functions.

* present an efficient CFI scheme for virtual calls in CFI based on the interleaved layout that our interleaving algorithm generates.

* evaluate our scheme on popular real-world C++ applications running on multiple platforms with different resource constraints, showing our scheme's efficacy and applicability to a

```
1  struct A {
2    int a;
3    virtual void f0();
4  };
5
6  struct B {
7    int b;
8    virtual void g0();
9    virtual void g1();
10 };
11
12 struct C : A, virtual B {
13   int c;
14   virtual void f0();
15   virtual void g1();
16 };
17
18 struct D : A {
19   int d;
20   virtual void f1();
21   virtual void f2();
22 };
23
24 struct E : D {
25   int e;
26   virtual void f0();
27   virtual void f2();
28 };
```

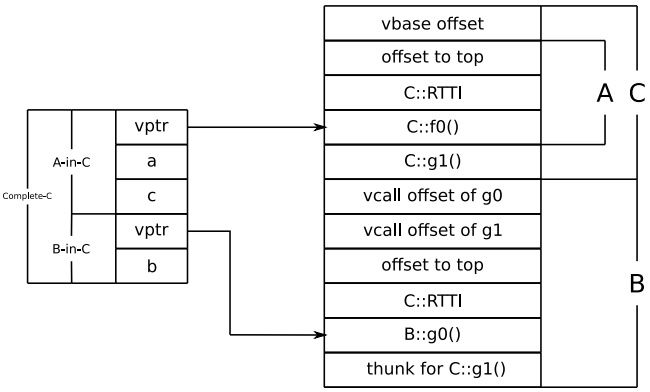
(a) Class Definitions



(b) Class Type Hierarchy

```
A * ptr = ...
...
ptr->f0();
```

(c) Virtual Call Example



(d) The layout of a complete object of `C` and its vtable object

Figure 1.1. The running example in C++

broad range of C++ applications.

1.2 Background

This section covers the related background of this work. We will first review C++ basics including inheritance and dynamic dispatch, from both the language perspective and the implementation perspective, and discuss the security implications of dynamic dispatch in C++. Then, we will look at how CFI schemes protect dynamic dispatch in general with a special focus on the original vtable interleaving scheme [39] that inspired our work.

1.2.1 C++ Basics

Inheritance and dynamic dispatch are two important features of C++ that are closely related to our work, so for these two features we will review the language-level semantics. We

will also describe how they are implemented at the binary level, which helps one to see the potential attacks C++ programs are subject to and to understand our scheme operating at the same level. There are two major C++ ABIs: the Itanium C++ ABI and the Microsoft Visual C++ ABI. For the rest of the paper, we will assume the use of the Itanium C++ ABI since that is the ABI our implementation currently supports. However, our scheme should be applicable to the Microsoft Visual C++ ABI without significant changes.

Inheritance

As C++ is an object-oriented programming language, inheritance is a crucial feature that allows developers to create derived object types based on base object types. The C++ standard requires that each instance of a derived type contains a copy of every base type in the inheritance hierarchy above it; such a copy is called a **subobject**. An object that is not a subobject of any other object is a **complete object**. Henceforth, we will use *X-in-Y* to refer to the subobject of a base type `X` in a complete object of a derived type `Y`, *Complete-X* to refer to a complete object of type `X`, and *object* to refer to both a complete object and a subobject, unless we need to differentiate them. C++ also allows for more complex forms of inheritance such as multiple inheritance and virtual inheritance. Multiple inheritance is when a type inherits directly from multiple base types. For example, type `C` inherits from `A` and `B` in Figure 1.1. Virtual inheritance is a feature to ensure that a base type `T1` has exactly one subobject in an object of the derived type `T2`, even when there is more than one base type of `T2` virtually inheriting `T1`. Type `B` in Figure 1.1 is a virtual base of `C`.

At the binary level, in a complete object of a derived type `T`, the Itanium C++ ABI overlaps the subobjects for `T`'s base type(s) that share data fields and lays out objects that cannot be overlapped linearly. The memory layout of a complete object of `C` in Figure 1.1 is shown in Figure 1.1d. We say type `T1` and type `T2` are **aligned** if an object of `T1` and an object of `T2` start at the same address. A complete object may have several groups of aligned types. For example, in a complete object of `C`, types `A` and `C` are in the same group while `B` is in its own group. We

call such a group an **aligned group**. The types in an aligned group form a linear hierarchy. For `Complete-C` in Figure 1.1d, types `A` and `C` are in the same aligned group, and the inheritance hierarchy between them is `A` $\rightarrow$ `C`. If a complete object of a type has multiple aligned groups, we say this type is a **complex type**. Virtual inheritance makes the layout more complex. Due to the semantics of virtual inheritance, the Itanium C++ ABI only allocates one subobject for a virtual base in each complete object of a derived type. This implies that the subobject of a virtual base may be "floating" in the complete objects of its derived types. That is, the distance between the subobject of a virtual base `T` and the object of a derived type of `T` may change, depending on what the complete object is. For the example in Figure 1.17, in a complete object of `C`, the distance between the object of `C` and the subobject of `A` is zero. However, in a complete object of `D`, a deriver of `C`, the distance between the subobject of `C` and the subobject of `A` is no longer zero, as is shown in Figure 1.17d.

Dynamic Dispatch

Polymorphism is fundamental to object-oriented programming languages like C++. One way polymorphism manifests itself is through **dynamic types**, which are types that define or inherit at least one virtual member function. Polymorphic behaviors appear when a virtual member function is called via a pointer or reference to a dynamic type. Such a call is called a **virtual call**. For example, line 3 in Figure 1.1c is a virtual call to `f0` via a pointer `ptr` to dynamic type `A`. Because the pointer or reference may point to `Complete-A` or any `A-in-X`, where `X` is a deriver of `A`, and C++ requires a virtual call to invoke the final overrider of the virtual member function, the virtual call in Figure 1.1c may invoke `A::f0`, `C::f1`, or `E::f1`, depending on what object `ptr` actually points to. Since in general the actual type of the object pointed to by `ptr` cannot be determined at compile-time, the call dispatch has to delay until run-time. This run-time dispatch is called dynamic dispatch.

A common way to implement dynamic dispatch in C++ is using virtual tables (vtables). The two most popular C++ ABIs, the Itanium C++ ABI and the Microsoft Visual C++ ABI,

use this approach. For the Itanium C++ ABI, every object (including complete objects and subobjects) `Obj` of a dynamic type `T` has a vtable. A vtable contains two kinds of entries: metadata entries and function pointer entries. Metadata entries, located at the upper part of a vtable, are used to find related objects and data structures of `Obj`. For example, the offset-to-top entry can be used to locate the start of the complete object that contains `Obj` when `Obj` is a subobject; the Run-Time-Type-Information (RTTI) entry points to the `TypeInfo` object of `Obj`, providing the type information of the complete object containing `Obj`. Function pointer entries, located at the lower part of a vtable, point to the final overriders of virtual member functions of `T`.

In this work, we make a clear distinction between a *vtable* and a *vtable object*. We use *vtable* to refer to the logical structure that enables dynamic dispatch and *vtable object* to refer to the physical storage of vtable(s). Every object of a dynamic type has a vtable. For all the complete objects of a dynamic type `T`, the Itanium C++ ABI creates a single vtable object that contains all the vtables for `T` itself and the subobjects of `T`. The layout of vtables in the vtable object of `T` is closely related to the layout of a complete object of `T`. Specifically, the vtables of an aligned group overlap so that the vtable of a less derived type is embedded in a vtable of a more derived type. The vtable of the most derived type of each group of aligned types contains all the vtables of this group. The Itanium C++ ABI combines the representative vtables of different aligned groups together to form the vtable object for `T`. For example, Figure 1.1d shows the layout of the vtable object for `C`. The upper half of the vtable object is for the group of aligned types `A` and `C`, while the lower half is for the group of type `B`. For the first aligned group, `A`'s vtable is embedded inside of `C`'s vtable.

We want to emphasize that the way we use the term *vtable* is different from the way the Itanium C++ ABI uses it, and this difference manifests itself with complex types. For a complex type `T`, the Itanium C++ ABI refer to the whole vtable object created for `Complete-T` as `T`'s vtable. For example, type `C` in Figure 1.1 is a complex type, and in the Itanium C++ ABI's terminology, the vtable object in Figure 1.1d is a vtable for `C`. However, we call the upper

```
%vptr = load %ptr
%f1_ptr = load (%vptr + 0)
call %f1_ptr
```

Figure 1.2. The low level code for the virtual call in Figure 1.1c

part of this vtable object (marked in Figure 1.1d) a vtable for `C` because that is the part used for dynamic dispatch on `C`. Our notion of vtable helps us to map a clear relationship between types and their vtable layouts, as we will see later in Section 1.3. Let vtable object `VTO` be the vtable object created for complete objects of `T`. We say that the vtable of `T` allocated in `VTO` is the **standard vtable** of `T`, and the layout of this vtable the **standard layout** of `T`. From "Chapter 2: Data Layout" of the Itanium C++ ABI, we can derive that every vtable of `T` inherits the standard layout of `T`.

An object connects to its vtable via a virtual pointer (`vptr`), which is the first element of the object data structure in memory. The `vptr` of an object points to the first virtual function pointer in the vtable. When the `vptr` of an object (complete object or subobject) of type `T` points to `offset` bytes into a vtable object `VTO`, we say that `VTO` has a **compatible address point** for `T` denoted `(T, offset)`. If a vtable object contains more than one address point, this vtable object has a **vtable group**. For example, the vtable object of `C` in Figure 1.1d has a vtable group. Note that a static type `T` may have multiple compatible address points because objects of `T` contained in different complete objects have different vtables located in different vtable objects. For example, in Figure 1.1 `Complete-A`, `A-in-C`, `A-in-D`, and `A-in-E` all have different vtables, so `A` has four compatible address points.

With vtables, at a virtual call site, the Itanium C++ ABI first loads the `vptr` of the object, calculates the address of the pointer to the virtual function to be called by adding its constant offset from the address point to `vptr`, then loads the function pointer, and finally invokes it. The pseudo LLVM IR code of the virtual function call in Figure 1.1c is shown in Figure 1.2 to illustrate this process.

Knowing how dynamic dispatch is implemented, we can represent a dynamic dispatch by a `(Type, Offset)` pair, where `Type` is the static type of the dispatch and `Offset` is the offset of the used vtable entry from the address point pointed by the `vptr`. For example, the virtual call in Figure 1.1c can be represented as `(A, 0)` because function `f0` is the first virtual member function located at offset zero. Note that such a dynamic dispatch may happen to any object of `Type`, therefore the vtable used by this dispatch may be any vtable of `Type`. For example, the dynamic dispatch `(A, 0)` may use the vtable of `Complete-A`, `A-in-C`, `A-in-D`, or `A-in-E`. We call the list of vtable entries that may be used by a dynamic dispatch its **entry group**. The entry group for `(A, 0)` contains the entries at offset zero from the address points in the four aforementioned vtables.

1.2.2 Dynamic Dispatch Hijacking

Dynamic dispatch in C++ is a powerful mechanism. At the same time, it presents a great opportunity for the attacker to hijack the control flow of C++ programs. For a dynamic dispatch, there is a static type and a run-time type. The static type is the type of the pointer or reference from which the virtual call is invoked, while the run-time type is defined by the address point pointed by the `vptr`, as the pointed vtable decides the dynamic behaviors. For example, the run-time behaviors of the virtual function call to `f0` in Figure 1.1c are decided by the `vptr` in the object pointed by `ptr`; different objects' `vptr`s point to different vtables, which contain function pointers to different versions of `f0`. Attacks on dynamic dispatch often start with creating a mismatch between the static type and the run-time type. This can be done by corrupting the `vptr` using an out-of-bounds write or by an invalid type casting. For example, consider:

```
1 void foo(void * vp) {
2   T1 * ptr = (T1 *) vp;
3   ptr->bar();
4 }
```

The function `foo` above expects a void pointer pointing to an object of type `T1`. If the attacker

manages to pass a vp pointing to an object of type T2, which may either be an unrelated type or a base type of T1, then a mismatch is created.

With such a mismatch, the attacker could access out-of-bounds memory if the size of T1 is greater than the size of T2, or, even worse, hijack the control flow when a virtual call is invoked via ptr. If the mismatch is created by type casting, the attacker may invoke a powerful virtual member function from an unrelated type. If the mismatch is created by corrupting the vptr and the address of a target function is stored somewhere in memory, the attacker may point the vptr to an address such that after calculation the address of the target function is used as the function pointer to virtual member function bar.

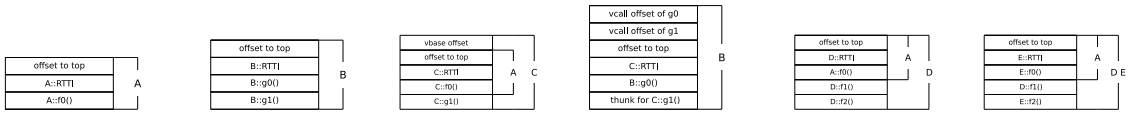
To mitigate attacks on dynamic dispatch, we need to detect the exploitation of such a mismatch between the static type and the run-time type.

1.2.3 Virtual Call CFI

Control-Flow Integrity (CFI) [37] is a general security mechanism to mitigate control-flow hijacking by restricting dynamic transfers to a small set of allowed targets. As virtual calls are also dynamic transfers, researchers have developed CFI schemes to specifically protect them. Because the corruptible control data for a virtual call is the vptr, virtual call CFI can be enforced by checking the validity of the vptr before it is used. Specifically, when a virtual call CFI scheme is enabled, the virtual call in Figure 1.1c will be instrumented as follows:

```
1 %vptr = load %ptr
2 call check_ptr(%vptr)
3 %f1_ptr = load (%vptr + 0)
4 call %f1_ptr
```

Here the function check_vptr checks the validity of vptr before it is used. If the check fails, the execution of the program terminates. What is a valid value for the vptr depends on the design of the CFI scheme. Generally, the vptr may legally point to any address point of the virtual call's static type. For instance, the vptr in Figure 1.1c may point to A's address points



(a) TVO-A **(b)** TVO-B **(c)** TVO-C1 **(d)** TVO-C2 **(e)** TVO-D **(f)** TVO-E

Figure 1.3. The vtable objects of the running example in Figure 1.1 after the vtable object of `C` has been split

in the vtable objects of `A`, `C`, `D`, and `E`. Since the addresses of these vtable objects are normally arbitrary, so are their address points, which means that `check_ptr` requires an expensive set membership check. To lower the performance impact, prior work used strategies including moving vtable objects and allocating additional data structures. For example, LLVM CFI-VCall [16] aggregates the vtable objects of each type hierarchy together and allocates a bitset encoding compatible address points for each dynamic type. Assuming there are `N` vtable entries in total, for each dynamic type the scheme allocates a bitset of `N` (a string of `N` bits) in which each set bit indicates that the address of the corresponding entry in the aggregated vtable object is a compatible address point for the type. Although there are a number of optimizations to reduce the size of bitsets, this strategy to speed up `check_ptr` still introduces a high space overhead, which is a serious problem for resource-constrained platforms (more in Section 1.5). For a virtual call CFI implementation to be usable on those platforms, it has to have low space overhead as well.

Bounov, Kici, and Lerner proposed a novel CFI scheme for virtual calls that is optimized for both performance and space overhead [39]. Throughout the rest of this dissertation, we refer to this as the BKL scheme. The core idea is rearranging vtable objects and reassign address points so that for any type `T`, the compatible address points for it are normally placed consecutively in memory. This means that the function `check_vptr` normally can be implemented as an efficient range check. The first step of the BKL scheme is sorting vtable objects so that for any type `T`, the vtable objects containing compatible address points for `T` are consecutive. The insight behind this step is that for any type `T`, the compatible address points are only in the vtable objects of `T` and `T`'s derived types (because only those vtable objects contain a vtable for `T`). Intuitively,

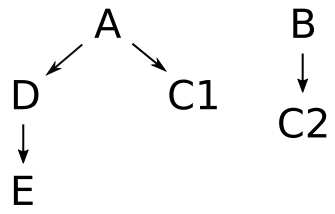


Figure 1.4. The decomposed type hierarchies of Figure 1.1b

a depth-first traversal of the type hierarchy would achieve this. However, this does not work when there are complex types. As we mentioned above, for any complex type, the Itanium C++ ABI glues the vtable objects of different aligned groups together. For example, the vtable object for `C` in Figure 1.1d contains address points for both `A` and `B`, which are two unrelated types, so it is impossible to sort vtable objects so that for both `A` and `B`, the vtable objects containing a compatible address point for them are consecutive. To solve this, the BKL scheme decomposes complex types into multiple sub-types, each of which corresponds to one aligned group. For example, type `C` in Figure 1.1 is split into `C1` and `C2`, representing the aligned group of `A` and `C` and the aligned group of `B`, respectively. Similarly, the vtable objects of complex types are also split so that every vtable object corresponds to exactly one aligned group. For example, `C`'s vtable object in Figure 1.1d will be split into two vtable objects: the upper half becomes the vtable object for `C1` and the lower half becomes the vtable object for `C2`. After these two steps, every type corresponds to exactly one vtable object, which has exactly one address point. The vtable objects of Figure 1.1 after decomposition are shown in Figure 1.3. Also, a decomposed type hierarchy, the type hierarchy with the complex types replaced by the newly created sub-types, is normally a tree (more in Section 1.3.5). For example, the decomposed type hierarchies of Figure 1.1 are shown in Figure 1.4. Then, the scheme orders the vtable objects by a depth-first traversal of the decomposed type hierarchies. For the hierarchies in Figure 1.4, the sorted vtable object lists are shown in Figure 1.5.

In the sorted lists, for any type `T` in a decomposed type hierarchy, the vtable objects that have compatible address points for `T` are normally consecutive (more in Section 1.3.5).

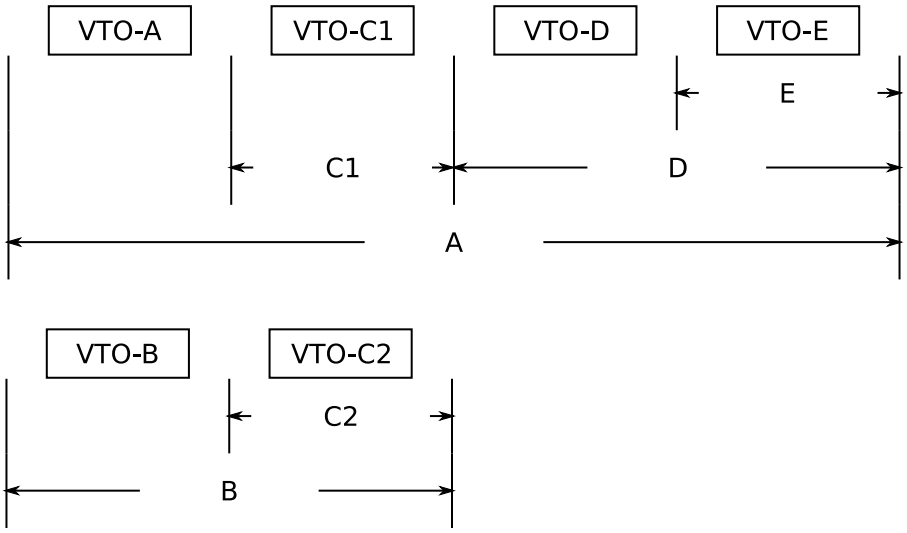


Figure 1.5. The vtable objects of the running example in Figure 1.1 sorted by the BKL scheme. The vtable objects are shown in Figure 1.3. The range of compatible vtable objects of each type is marked.

For example, for type `D`, the vtable objects that contain compatible address points are `VTO-D`, `VTO-E`, and they are consecutive in the first list. However, this property by itself does not make implementing `check_vptr` much easier since vtable objects may have different sizes, so the locations for address points are still arbitrary. The next key insight from the BKL scheme is that dynamic dispatch works as long as all the entries of the same kind have the same offset from their respective address points. For example, all the entries for `f0` have offset 0 from the address points in `VTO-A`, `VTO-C1`, `VTO-D`, and `VTO-E` in Figure 1.3. In addition, the absolute value of this offset does not matter as long as all the entries have the same offset from the address points. Taking advantage of this insight, the BKL scheme interleaves vtable objects together by a simple algorithm: it picks a vtable entry from a vtable object in the order computed previously and puts it in the interleaved layout until no entries are left in any vtable objects. Table 1.6 is the interleaved layout of vtable objects of the type hierarchy on the left in Figure 1.4. Note that interleaving only changes the physical layout of vtable objects. After interleaving, the number of vtable objects does not change, and these vtable objects still function independently. For each vtable object, the BKL scheme reassigns the address point to the first function pointer in the

object. For example, the address points in the interleaved layout in Table 1.6 are at the entries for function `f0`. An important property of the interleaved layout is that the vtable entries of the same kind (for the same metadata or function) are consecutive. Because all the entries for `f0` are consecutive, the address points (which are placed at the `f0` entries) are consecutive too. In the interleaved layout, the address points are consecutive, and entries for the same virtual member function or metadata are consecutive, so entries for the same virtual member function or metadata have the same offset from the corresponding address points, ensuring that dynamic dispatch still works. For example, entries for function `f2` are at offsets `0x80` and `0x88` and their address points are at `0x58` and `0x60`. Since the offset of any entry for function `f2` is `0x28` bytes from its address point, dynamic dispatch for function `f2` works with the interleaved layout. In addition, because for a type `T`, the vtable objects that have compatible address points are normally consecutive in the sorted vtable object list, and the vtable objects' address points are located in the same order as their containing vtable objects in the interleaved layout, the compatible address points for `T` are normally consecutive (more in Section 1.3.5). For example, type `A`'s range of compatible address points are from `0x48` to `0x60`. To check the validity of a `vptr` for the static type `T`, the scheme normally only needs to check if the `vptr` points to one of the address points in `T`'s compatible address point range. However, the authors do point out that in certain occasions they have to insert checks for multiple ranges, which we will discuss more in Section 1.3 where we compare our design with the BKL scheme.

1.3 Our Advanced Scheme

In this section, we will introduce the design of the advanced vtable interleaving scheme, which is inspired by the BKL scheme [39], but has a number of important improvements. We start off this section with the deployment model of general CFI schemes for C++ dynamic dispatch. Our advanced scheme achieves low space overhead by exploiting the assumptions of this model. We then describe our threat model and provide an overview of our scheme with an

Offset	VTO-A	VTO-C1	VTO-D	VTO-E
0x00		vbase offset		
0x08	offset-to-top			
0x10		offset-to-top		
0x18			offset-to-top	
0x20				offset-to-top
0x28	A::RTTI			
0x30		C::RTTI		
0x38			D::RTTI	
0x40				E::RTTI
0x48 (Addr Pt for VTO-A)	A::f0			
0x50 (Addr Pt for VTO-C1)		C::f0		
0x58 (Addr Pt for VTO-D)			A::f0	
0x60 (Addr Pt for VTO-E)				E::f0
0x68		C::g1		
0x70			D::f1	
0x78				D::f1
0x80			D::f2	
0x88				E::f2

Figure 1.6. The interleaved layout generated by the BKL scheme of the vtable objects `VTO-A`, `VTO-C1`, `VTO-D`, and `VTO-E` in Figure 1.3

emphasis on what our scheme does differently and its advantages. After that, we dive into the details of our scheme, focusing on two core components: how we order vtable objects and how we interleave vtable objects.

1.3.1 Deployment Model

Figure 1.7 shows the standard work flow of building a link unit, which is an executable or dynamic shared object. Compiler-based CFI schemes are normally implemented at the Link-Time Optimization (LTO) phase, which provides a global view over the program under protection. We call the set of compilation units passed into the LTO phase the *LTO unit* of the program, which is also the part of the program under protection. In addition, for CFI schemes protecting C++ dynamic dispatch, we denote by $\texttt{Types}_\texttt{p}$ the C++ types under protection. $\texttt{Types}_\texttt{p}$ must be a set of *closed* C++ types defined in the LTO unit—for any type `T` in $\texttt{Types}_\texttt{p}$, `T` and any of the base types or derived types of `T` are also in $\texttt{Types}_\texttt{p}$, and `T` is only referenced in the LTO unit. In

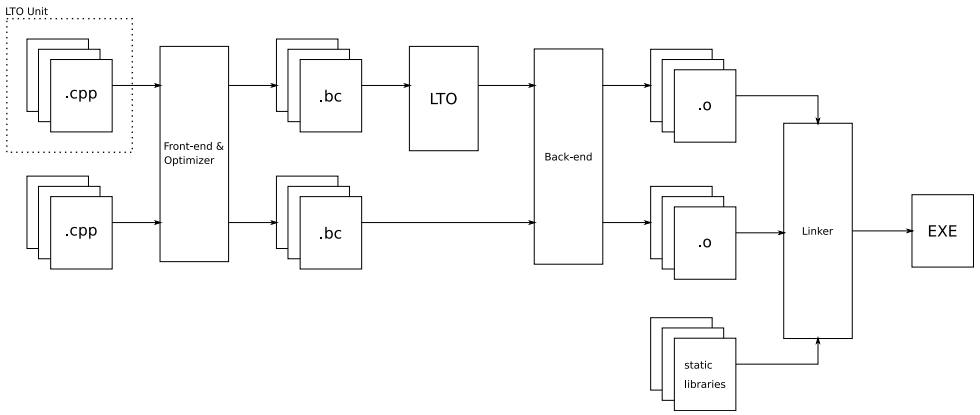


Figure 1.7. The work flow of building a link unit.

addition, we denote by $\texttt{VTO}_\texttt{p}$ the set of vtable objects related to $\texttt{Types}_\texttt{p}$—for any vtable object VTO in $\texttt{VTO}_\texttt{p}$, there is at least one type in $\texttt{Types}_\texttt{p}$ that has a compatible address point in VTO. Note that $\texttt{VTO}_\texttt{p}$ may not contain all the vtable objects that contain compatible address points of $\texttt{Types}_\texttt{p}$ in the program due to common compiler optimizations that remove unused functions and global variables (e.g. dead code elimination). The existing schemes [39] [16] achieve efficient `vptr` checking for dynamic dispatch on $\texttt{Types}_\texttt{p}$ by rearranging the layout of vtable objects in $\texttt{VTO}_\texttt{p}$. One key idea of our scheme is that we add another ingredient to the scheme: the instances of dynamic dispatch on $\texttt{Types}_\texttt{p}$. This allows our advanced scheme to know all the dynamic dispatch instances that it needs to protect, and thus also knows which parts of vtable objects in $\texttt{VTO}_\texttt{p}$ are actually in use. Specifically, we denote by $\texttt{DD}_\texttt{p}$ the set of all the instances of dynamic dispatch on $\texttt{Types}_\texttt{p}$ in the LTO unit.

Our observation is that this deployment model fits standalone C++ programs very well, because their LTO units normally contain all the types that the programs reference. For example, $\texttt{Types}_\texttt{p}$ of Chromium on Linux contains all the C++ types in it, including types in the `std` name space.

1.3.2 Threat Model

We assume that the attacker cannot corrupt non-writable regions of a program where instructions and constant variables, including vtable objects, are stored. Further, we assume

that the attacker cannot manipulate the values in registers or register spills on stack, otherwise the attacker could change the `vptr` in register after it was loaded from memory and passed the checking. Other than these exceptions, we assume that the attacker can corrupt `vptr`s by casting or out-of-bound writes. The attacker's goal is to subvert the control flow of the program by exploiting dynamic dispatch. Our scheme detects the exploitation of type confusion by making sure the run-time type of a pointer or reference is valid for the static type. Specifically, we ensure that the address point pointed by the underlying `vptr` is compatible with the static type. Compared with the BKL scheme, our scheme has higher protection precision (more in Section 1.3.7).

1.3.3 Overview

The core idea of the BKL scheme [39] is rearranging the storage layout of all the vtable objects of a program to allow faster `vptr` validity checking while maintaining some important layout properties to ensure that dynamic dispatch still works. In this work, we take the interleaving idea a step further: instead of meshing all the vtable objects in $\texttt{VTO}_\texttt{p}$ together, our scheme meshes only the *used* vtable entries in $\texttt{VTO}_\texttt{p}$ together. Interleaving vtables at a finer granularity not only further drives down the space overhead, which is crucial for the deployment of a compiler-based security scheme on resource-constrained platforms, but also allows us to maintain additional layout properties specified by the Itanium C++ ABI that the BKL scheme loses, making our scheme more compatible with the existing C++ programs. In addition, our scheme also provides higher protection precision because of our advanced vtable ordering and arrangement algorithms.

Our scheme consists of four parts: preparation, vtable ordering, vtable interleaving, and check instrumentation. For the rest of this section, we will describe each part in detail. To better illustrate our scheme, we continue to use the running example in Figure 1.1 for the rest of this section. Specifically, we assume the setup for our scheme in Figure 1.8. In this setup, all the types defined in Figure 1.1 are under protection. $\texttt{VTO}_\texttt{p}$ contains all the vtable objects created for

$Types_p = \{A, B, C, D, E\}$
$VTO_p = \{VTO\text{-}A, VTO\text{-}B, VTO\text{-}C, VTO\text{-}D, VTO\text{-}E\}$
$DD_p = \{(A, 0), (B, -3), (B, 1), (C, 0), (D, 1), (E, 2)\}$

Figure 1.8. The initial setup of our advanced scheme for the running example in Figure 1.1

the types `A`, `B`, `C`, `D`, and `E`. $\texttt{DD}_\texttt{p}$ contains the dynamic dispatch instances on $\texttt{Types}_\texttt{p}$, which only use a subset of vtable entries in $\texttt{VTO}_\texttt{p}$.

1.3.4 Preparation

Our advanced scheme relies on the relationship between C++ types and vtable object layouts. However, vtable objects generated by compilers normally make it hard to uncover this relationship so first we need to normalize the vtable objects in the program. As we mentioned in Section 1.2, for a complex type `T`, the Itanium C++ ABI combines vtable objects for different aligned groups of `T` together to form a vtable group, which obscures the clear relationship between types and vtable objects' layouts. As the first step of normalization, we split vtable objects that contain vtable groups into individual vtable objects. Note that the BKL scheme also does this splitting as part of its decomposition process. For example, the vtable object of `C` in Figure 1.1d will be split into two objects: the upper half becomes vtable object `VTO-C1` in Figure 1.3c, and the lower half becomes the vtable object `VTO-C2` in Figure 1.3d. Note that this splitting does not affect the compatibility with the Itanium C++ ABI since the ABI explicitly states that the relative positions of individual vtables in a vtable group do not matter. After the splitting, each vtable object has exactly one address point and corresponds to exactly one aligned group. Since there is only one address point in each vtable object, we say a vtable object `VTO` and a type `T` are **compatible** if the address point in `VTO` is compatible for `T`. After this step, $\texttt{VTO}_\texttt{p}$ for the running example in Figure 1.1 contains the vtable objects shown in Figure 1.3.

An important difference between our scheme and the BKL scheme is how vtable objects are associated with types. For any vtable object created for complete objects of a non-complex type `T`, the BKL scheme associates the vtable object with `T`. For example, `A` is a non-complex type

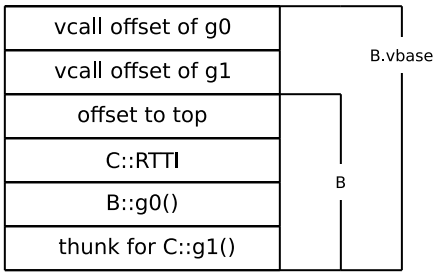


Figure 1.9. The vtables in `VTO-C2` after preparation. `VTO-C2` was the lower half of the vtable object for `C` in Figure 1.1d

in the running example in Figure 1.1 so `VTO-A`, the vtable object for `Complete-A`, is associated with `A`. For any vtable object that is split from a vtable group of a complex type, the BKL scheme associates it with the corresponding pseudo-type created during the decomposition. For example, `VTO-C2` in Figure 1.3 is split from the lower half of the vtable object for `C` in Figure 1.1d, and it is associated with the pseudo-type `C2`. On the other hand, our advanced scheme associates each vtable object `VTO` with the most derived type `T` that is compatible with `VTO`, and we say that `VTO` is *owned* by `T`. For example, `VTO-C2` in Figure 1.3 is owned by type `B` because `B` is the only type that has an address point in it. Furthermore, we have the following property about vtable objects:

Property 1. *If a vtable object `VTO` is owned by a type `T`, then `T`'s vtable allocated in `VTO` occupies `VTO` completely.*

Proof. Because `VTO` is owned by `T`, then `T` is the most derived type whose vtable is allocated in `VTO`. All other types that are compatible with `VTO` are base types of `T` and their vtables allocated in `VTO` are embedded inside `T`'s vtable in `VTO`. Therefore, `T`'s vtable allocated in `VTO` occupies `VTO` completely. □

As we mentioned in Section 1.2, all the vtables of the same type `T` inherit the standard layout of `T`. In fact, most of vtables for `T` generated by the Itanium C++ ABI have exactly the standard layout of `T` except for vtables for `T-in-S`, where `T` is a virtual base of `S`. For any virtual base `T` of a derived type `S`, the Itanium C++ ABI adds a `vcall offset` entry for every function defined in `T` to the vtable for `T-in-S`. For example, in Figure 1.1, `B` is a virtual base of `C`, and `B`

has two virtual functions (g0 and g1), so in the vtable for B-in-C in Figure 1.1d, there are two vcall offset entries. We want all the vtables of the same type T to have the standard layout of T, so we introduce a pseudo-derived type T.vbase for every virtual base type T. When a type S virtually inherits T, effectively S virtually inherits T.vbase, which in turn inherits T. With such pseudo-types, we can redefine vtables for virtual base type T and its pseudo-derived type T.vbase. For example, the vtable object VTO-C2, which was the lower part of the vtable object created for C in Figure 1.1d before splitting, now contains a vtable for C.vbase and a vtable for C, as they are shown in Figure 1.9. Note that the vtable for C.vbase includes the entries for vcall offset but the vtable for C does not. Since vtables for C no longer contain vcall offset entries, we also change dynamic dispatch instances to access vcall offset entries. For example, (B, -3) is a dynamic dispatch instance to access the vcall offset entry for function g1. We changed such a dynamic dispatch to (B.vbase, -3) because only vtables of B.vbase contain vcall offset entries. After this normalization step, we have the following property:

Property 2. *All of the vtables for the same type T have the standard layout of T.*

Combined with Property 1, we can prove the following property:

Property 3. *All of the vtable objects owned by T have the standard layout of T.*

Proof. For any vtable object VTO that is owned by T, with Property 1, we know that the vtable for T allocated in VTO has the same layout as VTO does. With Property 2, we know that all of the vtables for T have the standard layout of T, so VTO has the standard layout of T. □

After this step, the ownership of vtable objects in $\texttt{VTO}_\texttt{p}$ is shown in Table 1.1. VTO-C2 is owned by the pseudo-type B.vbase as it is the vtable for B-in-C where B is a virtual base for C. In addition, $\texttt{Types}_\texttt{p}$, $\texttt{VTO}_\texttt{p}$, and $\texttt{DD}_\texttt{p}$ are also updated and are shown in Figure 1.10.

Table 1.1. The mapping between the types and the vtable objects in which they have address points from the example in Figure 1.1. The vtable objects have been split and are shown in Figure 1.3. The asterisk (*) indicates that the vtable object is owned by the type.

Type	VTable Objects
A	VTO-A*, VTO-C1, VTO-D, VTO-E
B	VTO-B*, VTO-C2
B.vbase	VTO-C2*
C	VTO-C1*
D	VTO-D*, VTO-E
E	VTO-E*

$Types_p = \{A, B, B.vabse, C, D, E\}$
$VTO_p = \{VTO\text{-}A, VTO\text{-}B, VTO\text{-}C1, VTO\text{-}C2, VTO\text{-}D, VTO\text{-}E\}$
$DD_p = \{(A, 0), (B.vbase, -3), (B, 1), (C, 0), (D, 1), (E, 2)\}$

Figure 1.10. The updated setup of our advanced scheme for the running example in Figure 1.1 after preparation

1.3.5 VTable Object Ordering

The goal of the vtable object ordering phase is ordering vtable objects in $\texttt{VTO}_\texttt{p}$ so that for every dynamic dispatch `(type, offset)` in $\texttt{DD}_\texttt{p}$, the vtable objects containing a vtable entry for this dispatch are consecutive. In other words, we want to arrange vtable objects having the same layout (the same kinds of vtable entries) consecutively in this order. To order vtable objects in $\texttt{VTO}_\texttt{p}$, we rely on the following properties of vtable objects, which are derived from "Chapter 2: Data Layout" of the Itanium C++ ABI and the discussions from Section 1.2 and Section 1.3.4.

Property 4. *If `T1` and `T2` are compatible with the same vtable object `VTO` in $\texttt{VTO}_p$, then `T1` and `T2` are aligned.*

Proof. Because `T1` and `T2` are compatible with the same vtable object `VTO`, an object of `T1` and an object of `T2` share the same `vptr` pointing to `VTO`. Since `vptr` is the first element of any object data structure in memory, the object of `T1` and the object of `T2` start at the same address, thus `T1` and `T2` are aligned. □

In addition, we say that type T1 is an **aligned base** of type T2 if T1 is a base type of T2, and there exist at least one vtable object in $\texttt{VTO}_\texttt{p}$ that both T1 and T2 are compatible with. For aligned bases, we have the following properties:

Property 5. *If T1 is an aligned base of T2, then in Complete-T2, T1-in-T2 starts at the beginning of the object.*

Property 5 can be derived from "Chapter 2: Data Layout" of the Itanium C++ ABI. With this property, we can prove the following property about vtable object layout inheritance relationship among types and their aligned bases:

Property 6. *If type T1 is an aligned base for type T2, the standard layout of T2 inherits the standard layout of T1.*

Proof. Because T1 is an aligned base of T2, with Property 5, we know that Complete-T2 and T1-in-T2 share the same vptr pointing to the same vtable object VTO. Since VTO is created for Complete-T2, T2 is the most derived type that has a compatible address point in VTO, thus VTO is owned by T2. With Property 3, we know that VTO has the standard layout of T2. Since VTO contains the vtable for T1-in-T2, which has the standard layout of T1, thus the standard layout of T2 inherits the standard layout of T1. □

These important properties of vtable layout inspire us to unravel a hidden type hierarchy called the *vtable layout inheritance tree*, which accurately captures the vtable layout inheritance relationship among types. A vtable layout inheritance tree is a tree of types in which each node T is an aligned base for any of its children node D. Due to the existence of multiple inheritance and virtual inheritance, a C++ program may have multiple vtable layout inheritance trees, each of which corresponds to a disjoint vtable inheritance lineage. To build vtable layout inheritance trees of a program, we first need to partition $\texttt{Types}_\texttt{p}$ and $\texttt{VTO}_\texttt{p}$ into equivalence classes, each of which corresponds to a separate vtable layout inheritance tree. Our partition algorithm partition, shown in Figure 1.11, initializes every element in $\texttt{Types}_\texttt{p}$ and $\texttt{VTO}_\texttt{p}$ as a separate class. For each

```
partition (DD_p):
  // Set of equivalence classes of types
  // and compatible vtable objects.
  EC = { }

  for (Type, Offset) : DD_p
    ClassId = EC.createClass(Type)
    for VT : getCompatibleVTables(Type)
      if (EC.hasClass(VT))
        EC.merge(getClass(VT), ClassId)
      else
        EC.add(VT, ClassId)

  return EC
```

Figure 1.11. The algorithm to partition $\texttt{Types}_\texttt{p}$ and $\texttt{VTO}_\texttt{p}$ into equivalence classes, each of which forms a vtable layout inheritance tree.

type in $\texttt{Types}_\texttt{p}$, `partition` merges its class with the class of every compatible vtable object in $\texttt{VTO}_\texttt{p}$. Note that some types in $\texttt{Types}_\texttt{p}$ may not have any compatible vtable objects in $\texttt{VTO}_\texttt{p}$. For a type `T`, if `T` and `T`'s derived types are never instantiated in a program, the vtable objects that have a compatible address point for `T` will be removed by dead code elimination because they are not referenced. This means that some types in $\texttt{Types}_\texttt{p}$ may not appear in any class. Such types are not important as far as our scheme is concerned because they are never used in any dynamic dispatch. The classes created by `partition` have the following properties:

Property 7. *For any class `C` and any type `T` in `C`, any type `S` for which `T` is an aligned base is also in the class `C`.*

Proof. For any type `S` that `T` serves as an aligned base, there exists an vtable object in $\texttt{VTO}_\texttt{p}$ that both `T` and `S` are compatible with. Because `partition` merges the class of each vtable object in $\texttt{VTO}_\texttt{p}$ with the class of every type that it is compatible with it, it is guaranteed that `S` and `T` are in the same class after `partition` finished. □

Property 8. *For any class `C`, all the vtable objects in $\texttt{VTO}_\texttt{p}$ that are compatible with the types in `C` are also in `C`.*

```
build_tree (TypeSet):
  // Map each vtable obj to the
  // last seen type compatible with it.
  LastType = { }
  // Map each type to its children types.
  TypeChildren = { }
  // Map each type to the vtable objects
  // it owns.
  TypeOwnedVTO = { }

  sort(TypeSet)
  for Type : TypeSet
   for VTO : getCompatibleVTableObjs(Type)
    if (!LastType[VTO])
     TypeOwnedVTO[Type].add(VTO)
    else
     TypeChildren[Type].add(LastType[VTO])
    LastType[VTO] = Type

  return (TypeChildren, TypeOwnedVTO)
```

Figure 1.12. The algorithm to build vtable layout inheritance tree of each equivalence class.

Proof. For any type T in a class C, partition merges the class of every compatible vtable object of T in $\texttt{VTO}_\texttt{p}$ with C, so after partition finished, C must contain all compatible vtable objects of T in $\texttt{VTO}_\texttt{p}$. □

Each generated class corresponds to a disjoint vtable layout inheritance tree of the program. Property 7 ensures that each class contains all the types of the corresponding vtable layout inheritance tree, while Property 8 ensures that each class contains all the related vtable objects owned by the types in the tree. For our running example in Figure 1.1, we have the following two equivalence classes:

```
Class 1: A, C, D, E, VTO-A, VTO-C1, VTO-D, VTO-E
Class 2: B, B.vbase, VTO-B, VTO-C2
```

Next, we run the algorithm build_tree in Figure 1.12 on each equivalence class to build the vtable layout inheritance tree from bottom up and assign each vtable object to its owning

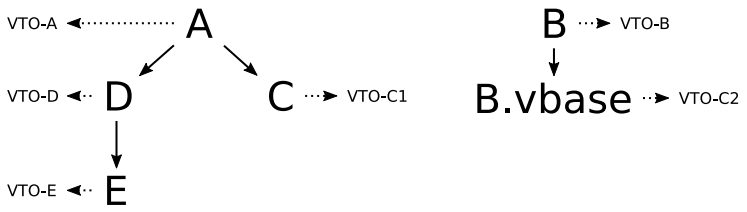


Figure 1.13. The vtable layout inheritance trees of the running example in Figure 1.1

type. The algorithm first sorts the types so that for each type, the derived types of it appear before it. Then the algorithm uses a nested loop to iterate through all the compatible vtable objects of every type in this order. For a type `T`, if a compatible vtable object of it has not been seen before, then `T` must be the most derived type compatible with the vtable object, thus `T` owns this vtable object. Otherwise, the vtable object is owned by some type `T´`. Since `T´` and `T` are compatible with the same vtable object, and `T´` is more derived than `T` (because `T´` has been processed), we know that `T` is an aligned base for `T´`. Because we process types from the most derived ones to the least derived ones, we know that `T` is the most derived aligned base for `T´`, so we add `T´` as a child node to `T`. After `build_tree` finished, we have the vtable layout inheritance tree corresponding to this equivalence class. The vtable layout inheritance trees for our running example in Figure 1.1 are shown in Figure 1.13. Since we have two equivalence classes before this step, `build_tree` created two trees.

Finally, for each vtable layout inheritance tree, we traverse it in the pre-order to sort the vtable objects owned by the types in the tree and collect entry groups needed for dynamic dispatch in $\texttt{DD}_\texttt{p}$. The traversal algorithm `traverse` in Figure 1.14 stores the ordered vtable objects attached to the tree in `SortedVTO`. In addition, for each type `T`, `traverse` records the range of vtable objects attached to the sub-tree rooted at `T` in `SortedVTO`. Specifically, at each type `T`, `traverse` first records the current length of `SortedVTO`, which will be used as the inclusive start index of `T`'s range in `SortedVTO`. Then `traverse` adds the owned vtable objects of `T` to `SortedVTO` and traverses `T`'s children recursively. When the traversal of all the descendants of `T` is done, `traverse` records the length of `SortedVTO` again as the exclusive end

```
traverse (Type, SortedVTO, TypeToRange,
          DD, UniqueGroups, [byval] UsedOffsets):
  Start = SortedVTO.length
  SortedVTO.append(Type.getOwnedVTObjs())

  for Off : DD[Type]
    if Off not in UsedOffsets
      UniqueGroups.add(Type, Off)
      UsedOffsets.add(Off)

  for T : Type.children
    traverse(T, SortedVTO, TypeToRange,
             DD, UniqueGroups, UsedOffsets)
  End = SortedVTO.length
  TypeToRange.add(Type, Start, End)

  return
```

Figure 1.14. The algorithm to traverse the vtable layout inheritance tree to sort the related vtable objects and collect necessary entry groups.

index of `T`'s range in `SortedVTO`. For our running example in Figure 1.1, the sorted vtable lists are shown in Figure 1.15.

In addition, `traverse` collects the unique entry groups for dynamic dispatch on the types in the tree. Note that for two instances of dynamic dispatch `(T1, offset)` and `(T2, offset)`, if `T1` is an ancestor of `T2` in a vtable layout inheritance tree, then the entry group of `(T2, offset)` is a subset of the entry group of `(T1, offset)` because the range of vtable objects of `T2` is within the range of `T1`. To collect the unique set of entry groups, for a type `T`, we need to keep track of the entry groups of `T`'s ancestors that we have collected and only add `(T, offset)` if none of `T`'s ancestors have dynamic dispatch on `offset`. To achieve this, `traverse` uses `UsedOffsets`, a pass-by-value parameter, to keep track of the collected entry groups of the ancestors. For our running example in Figure 1.1, the unique entries groups `UniqueGroups` of each vtable layout inheritance tree are shown in Figure 1.16. Note that `(C, 0)` in $\texttt{DD}_\texttt{p}$ in Figure 1.10 is omitted because `C` is a child node of `A` in Figure 1.13, thus the entry group of `(C, 0)` is a subset of the entry group of `(A, 0)`. After `traverse` finished, the sorted vtable object

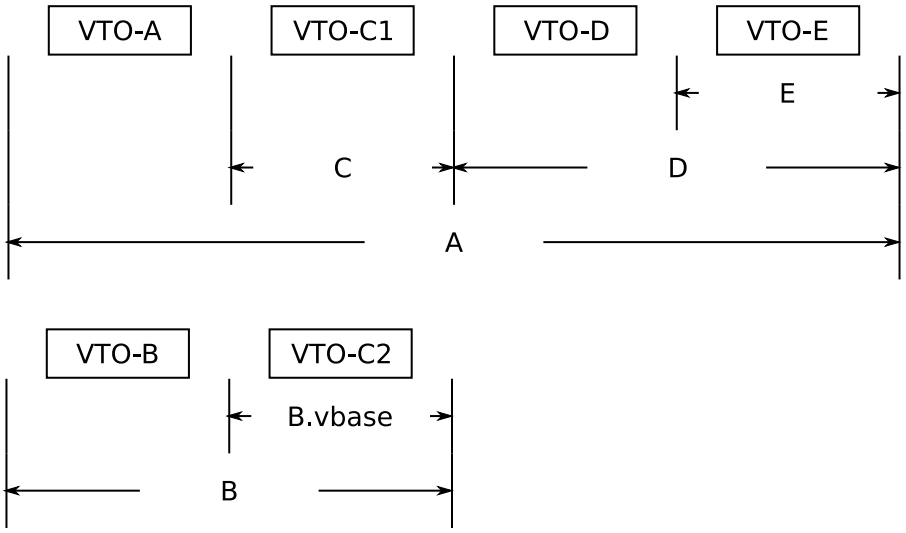


Figure 1.15. The vtable objects of the running example in Figure 1.1 sorted by our advanced scheme. The vtable objects are shown in Figure 1.3. The range of compatible vtable objects of each type is marked.

UniqueGroups `for the tree of A, C, D and E` $= \{(A,0),(D,1),(E,2)\}$
UniqueGroups `for the tree of B and B.vbase` $= \{(B.vbase,-3),(B,1)\}$

Figure 1.16. The unique entry groups for the running example in Figure 1.1

list `SortedVTO` has the following two properties:

Lemma 1. *For any dynamic dispatch `(T, Off)`, for any vtable object `VTO` in the range of `T`, `VTO` contains an entry at offset `Off` from its address point for this dynamic dispatch.*

Proof. All the vtable objects in the range of `T` are owned by the types in the sub-tree rooted at `T` in the vtable layout inheritance tree. For any vtable object `VTO` in the range, if it is owned by `T`, then it must have an entry at `Off` for this dynamic dispatch because Property 3 ensures that every vtable object owned by `T` has the same layout; otherwise, `VTO`'s most derived type is `D`, a derived type of `T`. Assume that the path between `T` and `D` is $\mathtt{M}_1$, $\mathtt{M}_2$, ..., $\mathtt{M}_n$. Since every type in this path is an aligned base for the next one, by Property 6, we know that the standard vtable layout of every type in the path inherits the standard vtable layout of the type before. In addition, Property 3 ensures that all of the vtable objects owned by the same type `T` have the standard layout of `T`, thus `VTO` inherits the standard layout of `T`, which means that at offset `Off` of `VTO` there is an entry

for this dynamic dispatch. □

Lemma 2. *For any type T, all the compatible vtable objects of T in VTO_p are in T's range.*

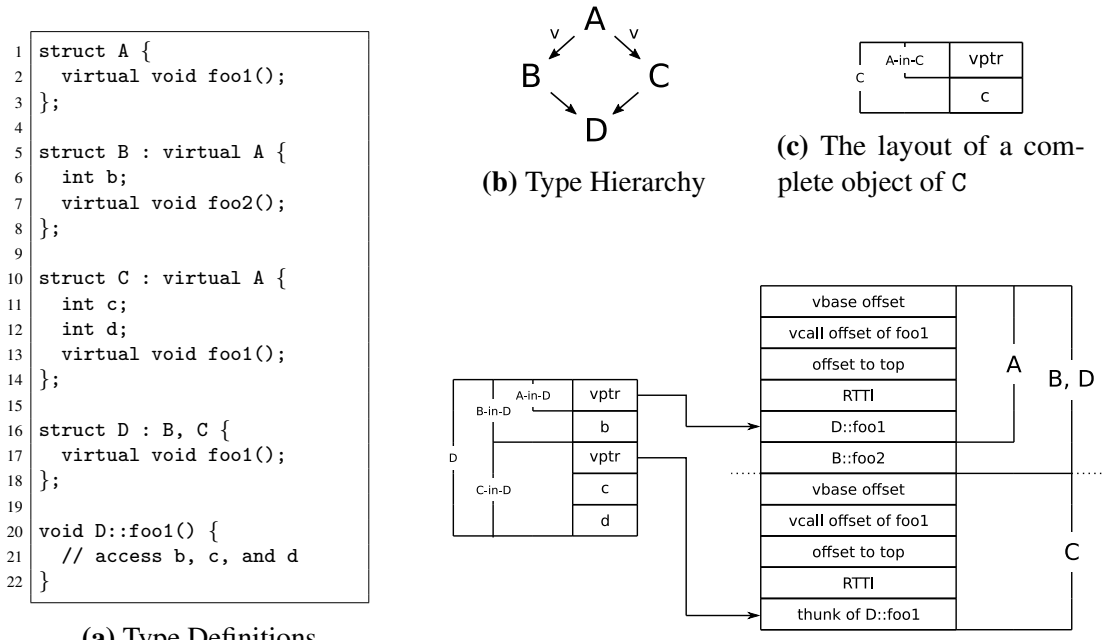
Proof. The algorithm `partition` ensures that all the compatible vtable objects of `T` are in the same equivalence class as `T`. The algorithm `build_tree` ensures that all the compatible vtable objects of `T` are attached to the types within the sub-tree rooted at `T`. Since the range of vtable objects of `T` are the vtable objects owned by the sub-tree rooted at `T`, it must contain all the compatible vtable objects of `T` in $\texttt{VTO}_\texttt{p}$. □

We want to point out that for any type `T`, although all the vtable objects in `T`'s range inherit the standard layout of `T`, some of them may not be compatible with `T`. This happens when `T` is a virtual base in a virtual diamond and `T` does not have any non-static data fields[1]. For example, in Figure 1.17, type `A` is an aligned base of `C` in a complete object of `C` (Figure 1.17c), thus `C` is a child node of `A` in the vtable layout inheritance tree, which means that all the vtable objects owned by `C` are in the range of `A`. The original vtable object for type `D` is shown in Figure 1.17d. After splitting, the lower half becomes a vtable object owned by `C`, so it is in the range of `A`, but it is *not* compatible with `A`. Note that the BKL scheme always assumes that all the vtable objects in the range of type `T` are compatible with `T`, which leads to lower protection precision as we shall see later in Section 1.3.7.

Comparison with the BKL scheme

As we mentioned in Section 1.2, to order vtable objects, the BKL scheme does a depth-first traversal of the decomposed type hierarchy, which is created by splitting complex types and their vtable objects. Compared to our vtable layout inheritance tree, the decomposed type hierarchy has less precision of capturing the relationship between vtable object layouts and types. For example, for the types in Figure 1.18, the type hierarchy, shown in Figure 1.18b, and the decomposed type hierarchy are the same because none of the types can be split. However,

[1] In the Itanium C++ ABI's terminology, `T` is a nearly empty virtual base.



(a) Type Definitions

(b) Type Hierarchy

(c) The layout of a complete object of `C`

(d) The layout of a complete object of `D` and its vtable object

Figure 1.17. A virtual diamond example in which the virtual base `A` is not compatible with every vtable object in its range

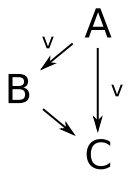
because type `C` has more than one parent type, the BKL scheme has to remove the edge between `B` and `C` to break the cycle and keeps two ranges for type `B`. The vtable layout inheritance tree of this case, on the other hand, captures precisely the linear vtable layout inheritance relationship among the three types, as is shown in Figure 1.18c. In addition, the BKL scheme is also subject to the incomplete range problem shown in Figure 1.17.

1.3.6 VTable Object Interleaving

The scheme next interleaves the vtable objects owned by the types in the same vtable layout inheritance tree together. We want to stress that interleaving is a process of rearranging the physical layout of related vtable objects. After interleaving, we have the same number of vtable objects that function independently. If an object points to a vtable object, then after interleaving, the object still points to the same vtable object, although the address point of the vtable object

```
1  struct A {
2    virtual void foo1();
3  };
4
5  struct B : virtual A {
6    virtual void foo2();
7  };
8
9  struct C : virtual A, B {
10   int c;
11   virtual void foo3();
12 };
```

(a) Type Definitions



(b) The type hierarchy of Figure 1.18a

(c) The vtable layout inheritance tree of Figure 1.18a

Figure 1.18. Example to illustrate the difference between the decomposed type hierarchy and the vtable layout inheritance tree

may move. To ensure that vtable interleaving does not break existing programs, our algorithm needs to maintain the following two layout properties in the vtables:

- Offset-to-top and RTTI entries layout property: The Itanium C++ ABI specifies that offset-to-top and RTTI entries appear consecutively at the offsets behind the corresponding address point.

- Entry group layout property: For the entry group of each dynamic dispatch `(Type, Offset)`, the distance between a vtable entry of this group and the corresponding address point is always the same. This property ensures that dynamic dispatch functions properly after interleaving.

Our interleaving algorithm `interleave`, shown in Figure 1.19, is designed to maintain the two properties above after interleaving. A common operation of `interleave` is collecting the entries of each unique entry group in Figure 1.16 from the ordered vtable object list in Figure 1.15. Specifically, for an entry group `(Type, Offset)`, we iterate through `Type`'s vtable object range and collect the entry at offset `Offset` of every vtable object. For example, for the entry group `(A, 0)`, `A`'s vtable object range consists of `VTO-A`, `VTO-C1`, `VTO-D`, and `VTO-E`, thus the entry group is `(A::f0, C::f0, A::f0, E::f0)`, where each entry is at offset zero in the

```
interleave (SortedVTO, TypeToRange, Groups):
  Result = [ ]
  Wl1 = [ ]
  Wl2 = [ ]

  for VTO : SortedVTO
    // collect offset-to-top entries
    Wl1.append(VTO[-2])
    // collect RTTI entries
    Wl2.append(VTO[-1])

  // sort Groups in the order of
  // decreasing size
  sortBySize(Groups)

  CurList = Worlist1
  for (Type, Offset) : Groups
    for I : TypeToRange[Type]
      CurList.append(SortedVTO[I][Offset])
    CurList = (size(Wl1) <= size(Wl2)) ?
                Wl1 : Wl2

  while size(Wl1) != size(Wl2)
    CurList.append(0)

  for I : [0, size(Wl1))
    Result.append(Wl1[I])
    Result.append(Wl2[I])

  return Result
```

Figure 1.19. The algorithm to interleave the vtable objects associated with the same vtable layout inheritance tree

Offset	VTO-A	VTO-C1	VTO-D	VTO-E
0x00	offset-to-top			
0x08	A::RTTI			
0x10 (Addr Pt for VTO-A)		offset-to-top		
0x18		C::RTTI		
0x20 (Addr Pt for VTO-C1)			offset-to-top	
0x28			D::RTTI	
0x30 (Addr Pt for VTO-D)				offset-to-top
0x38				E::RTTI
0x40 (Addr Pt for VTO-E)	A::f0			
0x48			D::f1	
0x50		C::f0		
0x58				D::f1
0x60			A::f0	
0x68				E::f2
0x70				E::f0
0x78		padding entry		

Figure 1.20. The interleaved layout generated by the advanced scheme of the vtable objects `VTO-A`, `VTO-C1`, `VTO-D`, and `VTO-E` in Figure 1.3

corresponding vtable object. Note that the Itanium C++ ABI requires that the offset-to-top and RTTI entries appear in every vtable, so `interleave` collects these two kinds of entries from every vtable object even if they are not used. In the algorithm, we first create two work lists `WL1` and `WL2`, which are initialized with the entry groups of offset-to-top and RTTI, respectively. After sorting the unique entry groups `UniqueGroups` in Figure 1.16 in the order of decreasing size, `interleave` collects the entries of each entry group and adds them to the shorter work list. If the two lists have different lengths after all the entry groups are processed, `interleave` pads the shorter one with zeros so that they are of the same length. Finally, `interleave` merges the two work lists together by moving, in an alternating fashion, the current head of each list to the final interleaved layout. The address point of each vtable object is the next index of the vtable object's RTTI entry in the interleaved layout. For the `i`th vtable object in the sorted vtable object list, the byte offset of its address point in the interleaved layout is `(2i+1)*entry_size` where `entry_size` is the size of a vtable entry. The interleaved layout for the first vtable layout inheritance tree in Figure 1.13 is shown in Figure 1.20. Compared with the layout generated by

the BKL scheme in Figure 1.6, in the layout generated by our scheme, offset-to-top and RTTI entries of each vtable object are located immediately above the corresponding address point, which means that the offset-to-top and RTTI entries layout property is preserved. In addition, because of our advanced scheme's ability to remove unnecessary vtable entries, this layout contains fewer entries, which shows the lower space overhead of our scheme.

Correctness

A vtable interleaving algorithm is correct if the interleaved layout it produces always satisfies the two properties mentioned above. It is straightforward to see that the interleaved layout always maintains the offset-to-top and RTTI entries layout property because of how the two work lists are merged and how address points are set. For the entry group layout property, we need to show that:

- The entry group of a dynamic dispatch `(T, Off)` contains all the vtable entries that may be used.
- All the entries in an entry group have the same distance from their address points in the interleaved layout.

The first property can be restated as: the entry group of a dynamic dispatch `(T, Off)` contains all the vtable entries at offset `Off` from the address points of the vtable objects compatible with type `T`, which is guaranteed to be true by Lemma 2. For the second property, let us assume that the ordered list of vtable objects as $\texttt{VTO}_1$, ..., $\texttt{VTO}_\texttt{n}$. For any dynamic dispatch `(T, Off)`, let us assume that the range for `T` in the ordered vtable object list is $\texttt{VTO}_\texttt{i}$, $\texttt{VTO}_{\texttt{i+1}}$, ..., $\texttt{VTO}_{\texttt{i+k}}$ where $1 <= i <= i+k <= n$. Suppose that the index of the address point of $\texttt{VTO}_\texttt{i}$ is `p` and the index of the entry for this dynamic dispatch in $\texttt{VTO}_\texttt{i}$ is q. For any vtable object $\texttt{VTO}_{\texttt{i+t}}$ in the range of `T` where $0 <= t <= n$, the index of its address point in the interleaved layout is $p + 2 * i$ (we have two times the index of the vtable object because the address points are two entry size apart due to the merge of the two work lists) and the index of the entry for this dynamic dispatch (the entry

at offset `Off`) is $q + 2 * i$, so the offset of this entry in the interleaved layout is $q - p$. Since the distance does not depend on i, we know that the distance is the same for all the entries in the entry group of this dynamic dispatch, thus the second property holds in the interleaved layout.

Comparison with the BKL scheme

The BKL scheme does not maintain the offset-to-top and RTTI layout property in the interleaved layout. However, this layout property is commonly assumed by low-level support libraries. For example, libsupc++ and libc++abi are the low-level support libraries for GCC's and LLVM's C++ standard libraries, respectively, and they both assume this property. This means that the BKL scheme does not work with any programs that access the offset-to-top or the RTTI entries (e.g. by using the `dynamic_cast` operator or exceptions) and built by GCC and LLVM, which significantly restricts its applicability.

1.3.7 Check Instrumentation

At each dynamic dispatch site for a static type `T`, the check instrumentation phase inserts a check to ensure that the `vptr` to be used points to a valid address point for `T`. Depending on whether `T` is compatible with all the vtable objects in its range, this phase may instrument one of two kinds of checks:

Full Range Check

When `T` is compatible with all the vtable objects in its range, we only need the following range check to ensure the validity of the `vptr`.

```
1 $diff = sub $vptr, $first_ap
2 $index = rol $diff, $align
3 cmp $index, $max_index
4 jgt FAIL
5 ... // check passed
```

First, the code extracts the index of the address point the `vptr` points to by substracting `first_ap`, the address of the first address point in `T`'s range, from `vptr` and rotating difference

to right by the alignment of address points, which is $log_2(entry\_size * 2)$ (address points are two times entry size apart). The result `index` is the index of the address point pointed by `vptr`. Since `T` is compatible with all the vtable objects in the range, we compare `index` with the biggest index `max_index` for this range. If `index` is greater than `max_index`, the check fails. Note that the rotate instruction at line 2 also ensures that `vptr` is properly aligned because the lower bits will be fed into the higher bits. If `vptr` is not aligned, the unaligned lower bits will appear at higher bits in `index`, which is guaranteed to be greater than `max_index`.

Incomplete Range Check

We adopt the bit vector technique used by LLVM CFI-VCall [16] to check types having incomplete ranges. Specifically, we store a bit vector for every type that has an incomplete range. A set bit in the bit vector indicates that the type is compatible with the corresponding vtable object (address point) in the type's range. In the check, we first use the same sequence of instructions for the full range check to ensure that the index of the address point pointed by `vptr` is in range, then check if this address point's corresponding bit is set in the bit vector.

Comparison with the BKL scheme

As we mentioned in Section 1.3.5, due to the limitation of its vtable ordering algorithm, the BKL scheme may have multiple ranges for a type. When the BKL scheme instruments a dynamic dispatch on such a type, it needs to instrument one check for each range.

In addition, the original scheme does not account for the scenario when a type is not compatible with all the vtable objects in its range, leading to lower protection precision. For the example in Figure 1.17, when the BKL scheme is enabled, the address point pointed by `C-in-D` in Figure 1.17d would be considered valid for type `A` because in the decomposed type hierarchy of the original scheme, `C` is a child of `A` and thus any address points valid for `C` are also valid for `A`. Suppose that in the program a pointer `ptr` of type `A` points to an object of `B` and later virtual member function `foo1` is invoked via `ptr` as is shown in the code snippet below.

```
1 A * ptr = new B();
2 // manipulate the vptr pointed by ptr
3 ptr->foo1();
```

On a little-endian machine, if the attacker has the ability to overwrite only the first few bytes (low order bytes) of the `vptr` pointed by `ptr`, they could likely make `vptr` point to the address point of `C-in-D` in Figure 1.17d, because after interleaving, address points are located closely. The check inserted by the BKL scheme would allow `ptr->foo1()` to run, which would access field `d`, but the object pointed by `ptr` does not have such a field. In this way, a modest memory bug is amplified by the imprecision of the original scheme, giving the attacker ability to access more out-of-bounds memory. Our advanced scheme, on the other hand, does not consider the address point pointed by `C-in-D` valid for `A`, and will disallow calling `foo1` in this scenario.

1.4 Implementation

We implemented our design on top of the LLVM compiler infrastructure (Rev. 971cb8b6). Our implementation consists of a modified Clang front-end and a vtable interleaving pass as part of the LLVM Link-Time Optimization (LTO). Because the existing LLVM VCall does a similar job, we reuse a significant amount of code from it. Users can enable our scheme by passing the `-vtable-interleaving` flag to the compiler driver at both the compile stage and the linking stage. Our implementation supports protecting both virtual calls and type casting in C++ with the existing LLVM flags `-fsantize=cfi-vcall`, `-fsanitize=cfi-derived-cast` and `fsanitize=cfi-unrelated-cast`. For the rest of this section, we will briefly describe the components that make up our advanced scheme.

Components in the front-end

The front-end components are intended to keep necessary information for the interleaving pass in LTO. Specifically, they do the following things:

* Marking address points in vtable objects: mark each vtable with its address points in the

form of `(type, offset)` where `offset` is the byte offset from the start of the vtable object. We reuse the existing code for this.

* Inserting `vptr` check placeholder: at each dynamic dispatch site where `vptr` is used, the front-end instruments a `vptr` check. We reuse the existing code for this.

* Creating pseudo-types: as part of Section 1.3.4, in the codegen part of Clang, for each vtable that is compatible with a virtual base `T` and has vcall offset entries for `T`, we insert an address point for the pseudo sub-type `T.vbase` of `T`.

* Intercepting vtable accesses: a vtable access can be represented as a `(static_type, offset)` pair. The modified front-end intercepts every vtable access `(static_type, offset)` and uses a placeholder for the offset. Such a placeholder will be replaced once the interleaved layout is determined.

Components in the LTO phase

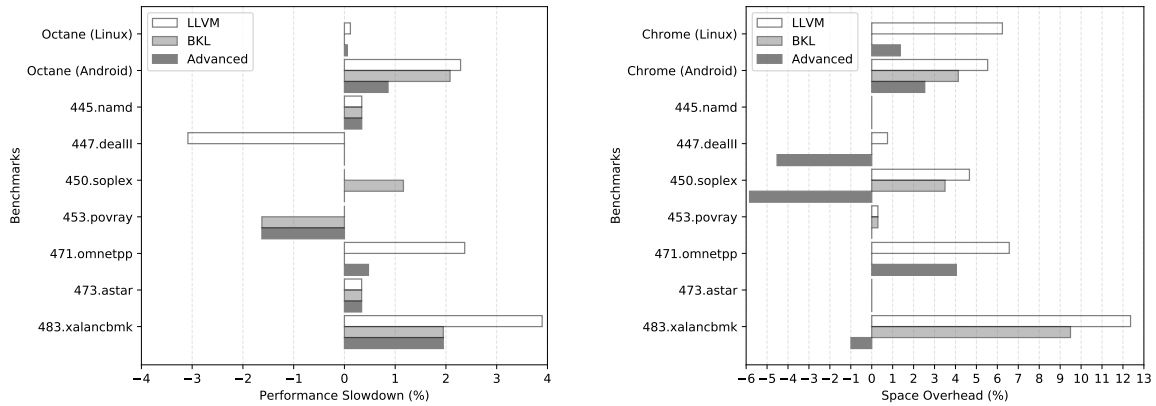
* The vtable splitting pass: before interleaving vtables, we first split vtable groups as we discussed in Section 1.3.4. An existing pass named `GlobalSplit`[2] already does this.

* The interleaving pass: we implemented the algorithms described in Section 1.3 in a transformation pass named `InterleaveVTables` at the LTO stage.

1.5 Evaluation

To show the applicability of our advanced interleaving scheme, we evaluated it with some popular C++ programs and benchmarks. We tried to keep the default compilation setup of the programs to better illustrate the performance and space overhead our scheme would introduce in real-world setting. To put the performance numbers of our scheme into perspective, for each benchmark we also evaluated the LLVM CFI-VCall, which is the state-of-art CFI scheme for

[2] https://llvm.org/doxygen/GlobalSplit_8cpp_source.html



(a) Performance slowdown compared to version without dynamic dispatch protection

(b) Space overhead compared to version without dynamic dispatch protection

Figure 1.21. The percentage performance slowdown and space overhead of LLVM CFI-VCall, the BKL scheme, and our advanced scheme

dynamic dispatch, and the BKL scheme that inspired our work. To provide a fair comparison between our scheme and the original scheme, we implemented the BKL scheme on the same LLVM base. The BKL scheme's vtable ordering algorithm is not fully integrated into the LLVM code base, so in our implementation of this scheme, we used our vtable ordering algorithm, which should have no impact on the space overhead and strictly lower performance overhead since it does not need to instrument multiple range checks for any dynamic dispatch. We used an Ubuntu 18.04 machine (AMD Ryzen 5 Pro 2400GE, 32GB RAM) for the desktop tests and a Google Pixel G-2PW4100 (Android 9) for the mobile tests. Since all three schemes only affect static regions (instructions and constant global variables) of a program, we evaluated the space overhead of a scheme by the increase in code size compared with the baseline build. We took the size of a binary after it had been stripped.

1.5.1 Chromium

Chromium is the open-source version of the popular Google Chrome browser that does not include Google's proprietary components. Chromium is a complex C++ program with extensive use of dynamic dispatch. We evaluated the Linux version and the Android version

of Chromium built from the same revision of the code base. These two versions are different in two aspects. First, the Linux version, by default, is built with LLVM CFI-VCall while the Android version is not, due to the space overhead of the scheme[3]. Second, LLVM CFI-VCall is applied to different scopes in the two versions. Specifically, in the Linux version of Chromium, ICU and libc++—the only two libraries that use RTTI—are also protected by LLVM CFI-VCall, whereas in the Android version they are not. As we mentioned in Section 1.3.6, the BKL scheme does not support RTTI, so it cannot be used as a drop-in replacement for LLVM CFI-VCall for the Linux version. One possible way to enable the BKL scheme on the Linux version to leave out ICU and libc++, but that would mean all the types in the `std` namespace and in ICU are no longer protected.

Methodology

We built the Linux and the Android versions of Chromium with the default settings using our advanced scheme, the BKL scheme, and LLVM CFI-VCall. The BKL scheme cannot build the Linux version of Chromium due to the aforementioned reasons. We use the popular browser benchmark Octane 2.0 [21] to evaluate the performance overhead of the three schemes. The documentation of Octane 2.0 suggests running the benchmark multiple times and reporting the best result; we followed this advice and ran the benchmark with each build ten times.

Results

The results of performance tests are shown in Figure 1.21a and the comparison of space overhead is in Figure 1.21b. Our scheme introduces less than 1% performance overhead to the Linux version and the Android version of Chromium, better than both the BKL scheme and LLVM CFI-VCall. As for the space overhead, our scheme introduces only 1.37% and 2.53% to the Linux version and the Android version, respectively, outperforming the other schemes on both platforms. The results show that our scheme is a potential solution to protect the Android version of Chromium, which is currently not protected by any CFI scheme.

[3]LLVM CFI-VCall can be manually enabled for the Android version.

1.5.2 SPEC 2006

We also evaluated our scheme on the SPEC 2006 benchmark suite. Because of the popularity of SPEC 2006 for compiler-related research, this evaluation helps compare our scheme with other compiler-based mitigation schemes. Since our scheme protects C++ programs, here we focus on the C++ benchmarks in SPEC 2006: 444.namd, 447.dealII, 450.soplex, 453.povray, 471.omnetpp, 473.astar, and 483.xalancbmk.

Methodology

We built the C++ benchmarks with our scheme, the BKL scheme, and LLVM CFI-VCall. We used optimization level `O2` for both compile-time optimization and link-time optimization for all the builds, including the baseline. For each scheme, we used the SPEC 2006's built-in utility to conduct a reportable run, which is suitable for public reporting according to SPEC 2006's documentation. We had to remove the invalid type casting in 483.xalancbmk where a pointer to the base type `Grammar` is cast to a derived type `SchemaGrammar`, but the pointer may point to an object of a sibling type of `SchemaGrammar`. In fact, this is exactly the kind of type confusion that virtual call CFI schemes aim to protect from. All of the three schemes terminated the program when the invalid `SchemaGrammar` was used for dynamic dispatch. Benchmarks 447.dealII and 471.omnetpp use the `dynamic_cast` operator in C++. As we discussed in Section 1.3.6, the BKL scheme does not support RTTI-related operations thus cannot build these two benchmarks.

Results

Figure 1.21a shows the performance and space overheads of the three schemes on C++ benchmarks in SPEC 2006. Across benchmarks, the advanced schemes introduces no more than 2% performance overhead. In general, the advanced scheme has similar performance overhead as the BKL scheme does, and is more efficient than the LLVM CFI-VCall, especially on benchmarks like 71.omnetpp and 483.xalancbmk, where virtual calls are extensively used. The results for two benchmarks do not follow this pattern. For 447.dealII LLVM CFI-VCall outperforms all other builds including the baseline, and for 450.soplex the BKL scheme is

Table 1.2. The comparison of space overheads introduced by LLVM CFI-VCall, the BKL scheme, and the advanced scheme to PX4.

Build	Space Overhead (%)
LLVM	1.41
BKL	1.24
Advanced	-0.88

slower than LLVM CFI-VCall. We believe this is caused by the effects of caching or alignment, as all the three schemes change the memory layout in a unique way. As for space overhead, our scheme outperforms the other two schemes significantly. Particularly, for benchmarks like 483.xalancbmk and 450.soplex, our scheme does not introduce significant space overhead like the BKL scheme and LLVM CFI-VCall do, but even reduces the binary size due to its ability to pick out necessary vtable entries. This again shows the potential of our scheme to protect programs on resource-constrained systems where storage space is highly precious.

1.5.3 PX4

PX4 [28] is a popular open source flight control firmware framework for unmanned vehicles (e.g. drones, boats, and submarines) written in C++. PX4 comes with a set of standard functionalities but also allows users to customize it by adding additional modules, which will be built as static libraries by default. As it is written in C++, PX4 is potentially subject to control-flow hijacking. However, enabling compiler-based security mitigations poses a challenge because unmanned vehicles like drones may have a limited program memory. For example, Pixhawk 3 Pro [27] has 2MB flash and Omnibus F4 SD [22] only has 1MB.

Methodology

We built the standard PX4 v1.9.2 for the simulator target with the three schemes. We used `O2` optimization level for both compile-time optimization and link-time optimization for all the builds, including the baseline. Note that the target-dependent code in PX4 is mostly in C, so our results in this section should reflect the space overhead of the three schemes for other targets.

Results

Table 1.2 shows the comparison of the sizes generated by the three schemes along with the baseline build. Both the BKL scheme and the LLVM CFI-VCall introduce significant space overhead to PX4, which may be too expensive for a user who also wants to add additional modules to PX4. Our advanced scheme, on the other hand, actually manages to reduce the overhead by eliminating unnecessary vtable entries and virtual member functions. Our scheme not only secures dynamic dispatch in PX4, but also frees up space for additional modules or other hardening schemes that requires additional space. For example, we could further secure PX4 by enabling LLVM CFI-ICall [16], a CFI scheme that protects dynamic transfers via function pointers by allocating jump tables.

1.6 Related Work

The devastating consequences of control-flow hijacking have spurred extensive research on countermeasures. In this section, we will focus on practical solutions that may be used to protect C++ programs on resource-constrained systems, examining them in three aspects: performance overhead, space overhead, and protection precision.

LLVM's CFI-VCall [16] is the state-of-art mitigation for dynamic dispatch in C++. For every type used in dynamic dispatch, the scheme allocates a bit vector indicating the compatible address points, and at each dynamic dispatch site, this bit vector is used to determine whether the `vptr` to be used is compatible with the static type. As you can see in Section 1.5, this scheme introduces slightly higher overhead on SPEC 2006 and Chromium than our scheme due to its less efficient validity checks. In addition, storing bit vectors leads to much higher space overhead. As we have seen in Figure 1.21b, this scheme constantly introduces higher space overhead than our scheme. For example, for 483.xalancbmk that has an extensive use of virtual calls, it introduces more than 12% space overhead.

Like LLVM's VCall CFI, SAFEDISPATCH [46] also stores the set of valid address points

for each type in additional data structures. This leads to its more than 7% space overhead on Chromium. Also, when the program is about to access a vtable at runtime, the scheme checks the validity of the `vptr` by a linear search in the set of valid address points. This naive way of checking membership of a vtable leads to high performance overhead. For example, Chromium built with SAFEDISPATCH experiences about 8% slowdown on the Octane benchmark.

VTrust [62] presents two layers of defense against control-flow hijacking of virtual calls. The first layer is Virtual Function Type Enforcement, which, at each virtual call site, checks if the actual called function has the same hash signature of the function name and argument list as the virtual function. The protection precision is much lower than our scheme since it does not use the type hierarchy at all. The second layer, VTable Pointer Sanitization, encodes each `vptr` as an index into the array of VTables and decodes it before using it to access a VTable. This protection precision is still low since it cannot prevent the attacker to point `vptr` to a vtable of a completely irrelevant type. Also, the performance overhead of VTable Pointer Sanitization for 483.xalancbmk, the C++ benchmark that uses virtual calls extensively in SPEC 2006, is 7.9%, which indicates that this solution may not be efficient enough for programs that have a heavy use of virtual calls.

Vip [45] uses static analysis to reduce possible candidate destinations at each virtual call site, which in some cases make Vip more precise than our scheme. However, such target reduction can be achieved by running optimizations like devirtualization, which we consider as orthogonal to our scheme. Also, to achieve fast validity checking at virtual call site, Vip relies on validity vectors, which significantly increases its space overhead. For example, the authors reported 6.5% overhead for 483.xalancbmk in SPEC 2006 whereas our scheme reduces the size of it by 0.99%.

All of the schemes that we have mentioned rely on the availability of the source code of the program under protection. There are also proposals that protect dynamic dispatch when only the binary is available. VTint [63] is a scheme that aims to secure vtable integrity of binaries. It detects vtables in the binary, allocates them in the read-only memory consecutively, and adds an

identical label at the start of every page that contains vtables. Note that this label is guaranteed not to appear in any page that does not have vtables. Then at each detected virtual call site, it inserts checks to ensure that the vtable to be used is in read-only memory and the page has the special label. Because VTint does not have information of the type hierarchy, the protection precision is very low.

Another scheme that operates on binaries, vfGuard [53], improves upon VTint by reconstructing C++ type hierarchy and inserting checks before (possible) dynamic dispatch sites based on the reconstructed hierarchy. However, since the reconstructed type hierarchy is an over-estimation of the actual hierarchy, the protection precision is lower than our scheme. Also, the authors report that it incurs an average 18.2% performance overhead on Internet Explorer's modules. Such a high overhead makes it unlikely to be used in practice.

PITTYPAT [43] goes further by enforcing path-sensitive CFI. To collect path information efficiently, PITTYPAT relies on Processor Tracing, a feature available in recent Intel processors, which prevents the scheme to be deployed where such a feature is not available. Also, since PITTYPAT computes the valid destination set online, the performance overhead is significant. For example, PITTYPAT introduced a 27.5% performance overhead for 450.soplex in SPEC 2006. In comparison, our scheme does not slow down this benchmark at all. PITTYPAT has higher precision than our scheme, but the high performance overhead makes it unlikely to be adopted widely.

Another approach to prevent control-flow hijacking in C++ programs is protecting `vptrs` from being corrupted, as is used by CPI and CPS from [47]. CPI and CPS work similarly – they allocate sensitive objects in a special safe region and accesses to the safe region are sandboxed unless they can statically be proved to be safe. CPI and CPS differ in what are considered as sensitive objects. In the context of C++, they both consider objects that contain `vptrs` sensitive, but only CPI considers pointers to such objects sensitive. This means that the attacker may corrupt `vptrs` via indirection even with CPS in place. Both CPI and CPS introduce signicantly higher performance overhead than our scheme. For example, CPI and CPS introduce

about 43% and 17% performance overhead respectively for 471.omnetpp in SPEC 2006 due to 471.omnetpp's heavy use of virtual calls. The authors of [47] do not mention the space overhead of CPI and CPS. We expect CPI to introduce significant space overhead due to the spatial and temporal metadata for sensitive pointers it keeps (which CPS does not) and the instrumented sandboxing code for every potentially unsafe memory access to the safe region.

CFIXX [40] is another scheme based on this approach. For every allocated object of some dynamic type (a type that has vtables), CFIXX stores the mapping from the start address of this object to its vtable object in the safe region. At each virtual call site, the `vptr` is loaded from the safe region, ensuring that the used vtable is legal for the underlying object. However, at each virtual call site, CFIXX does not check the relationship between the static type (the type of the pointer or reference used for the dynamic dispatch) and the runtime type (the type of the underlying object), which means the attacker may make a pointer to type `T` point an irrelevant type `T´`, and hijack the control flow when this pointer is used for dynamic dispatch. In terms of performance, CFIXX introduces much higher overhead. For 483.xalancbmk and 471.omnetpp from SPEC 2006, it introduces over 8% and 6% overhead, respectively. Due to needing to keep track of the mapping between every object and its vtable, [40] reports a 79% increase in memory usage in practice when CFIXX is enabled, which is normally not affordable for resource-constrained platforms.

1.7 Conclusion

There is an urgent need for highly efficient mitigation schemes for C++ as the trend of edge-computing pushes increasingly more C++ programs into resource-constrained systems. Dynamic dispatch in C++ is a convenient target for attackers to take over the program, in this paper we presented an advanced vtable interleaving algorithm and a highly efficient CFI scheme based on it. Compared with the original work that inspired our scheme, ours yields an interleaved layout that is fully compatible with the Itanium C++ ABI, has better protection precision, and

much lower space overhead without sacrificing the performance. Because of the low performance and space overhead of our scheme, it can be a potential solution to protect C++ programs running on resource-constrained systems, where all of the existing CFI schemes are considered too expensive to be deployed.

Acknowledgements

Chapter 1, in part, is currently being prepared for submission for publication of the material. Yang, Zhaomo; Collingbourne, Peter; Levchenko, Kirill. The dissertation author was the primary investigator and author of this material.

Chapter 2

Challenges

2.1 Introduction

Concerns over memory disclosure vulnerabilities in C and C++ programs have long led security application developers to explicitly *scrub* sensitive data from memory. A typical case might look like the following:

```
char * password = malloc(PASSWORD_SIZE);
// ... read and check password
memset(password, 0, PASSWORD_SIZE);
free(password);
```

The `memset` is intended to clear the sensitive `password` buffer after its last use so that a memory disclosure vulnerability could not reveal the password. Unfortunately, compilers perform an optimization—called *dead store elimination* (DSE)—that removes stores that have no effect on the program result, either because the stored value is overwritten or because it is never read again. In this case, because the buffer is passed to `free` after being cleared, the compiler determines that the memory scrubbing `memset` has no effect and eliminates it.

Removing buffer scrubbing code is an example of what D'Silva *et al.* [44] call a "correctness–security gap:" From the perspective of the C standard, removing the `memset` above is allowed because the contents of unreachable memory are not considered part of the semantics of the C program. However, leaving sensitive data in memory increases the damage posed

by memory disclosure vulnerabilities and direct attacks on physical memory. This leaves gap between what the standard considers correct and what a security developer might deem correct. Unfortunately, the C language does not provide a guaranteed way to achieve what the programmer intends, and attempts to add a memory scrubbing function to the C standard library have not seen mainstream adoption. Security-conscious developers have been left to devise their own means to keep the compiler from optimizing away their scrubbing functions, and this has led to a proliferation of "secure memset" implementations of varying quality.

The aim of this chapter is to understand the current state of the dead store elimination and programmers' attempts to circumvent it. We begin with a survey of existing techniques used to scrub memory found in open source security projects. Among more than half a dozen techniques, we found that several are flawed and that none are both universally available and effective. Next, using a specially instrumented version of the Clang compiler, we analyzed eleven high-profile security projects to determine whether their implementation of a scrubbing function is effective and whether it is used consistently within the project. We found that only three of the eleven projects did so.

To aid the current state of affairs, we developed a single best-of-breed scrubbing function that combines the effective techniques we found in our survey. We have shared our implementation with developers of the projects we surveyed that lacked a reliable scrubbing function and have made it available to the public. While not a perfect solution, we believe ours combines the best techniques available today and offers a developers a ready-to-use solution for their own projects.

We also developed a *scrubbing-aware* dead store elimination optimization pass based on Clang. Our pass protects scrubbing operations by inhibiting dead store elimination in case where a store operation may have been intended as a scrubbing operation by the programmer. Our solution does not completely disable DSE, minimizing the performance impact of our mechanism. Our performance evaluation shows that our scrubbing-safe DSE introduces virtually no performance penalty.

In total, our contributions are as follows:

* We survey scrubbing techniques currently found in the wild, scoring each in terms of its *availability* and *reliability*. In particular, we identify several flawed techniques, which we reported to developers of projects relying on them.

* We present a case study of eleven security projects that have implemented their own scrubbing function. We found that no two projects' scrubbing functions use the same set of techniques. We also identify common pitfalls encountered in real projects.

* We develop and make publicly available a best-of-breed scrubbing function that combines the most reliable techniques found in use today.

* We develop a scrubbing-safe dead store elimination optimization that protects memory writes intended to scrub sensitive data from being eliminated. Our mechanism has negligible performance overhead and can be used without any source code changes.

The rest of the chapter is organized as follows. Section 2.2 provides background for the rest of the paper. Section 2.3 surveys the existing techniques that are used to implement reliable scrubbing functions. Section 2.4 examines the reliability and usage of scrubbing functions of eleven popular open source applications. Section 2.5 describes our `secure_memzero` implementation. Section 2.6 describes our secure DSE implementation and evaluates its performance. Section 2.7 discusses our results. Section 2.8 describes the related work. Section 2.9 concludes the chapter.

2.2 Background

The negative effects of DSE are not new to many developers. Bug reports are littered with incidents of DSE negatively affecting program security, as far back as 2002 from Bug 8537 in GCC titled "Optimizer Removes Code Necessary for Security" [2], to January 2016 when OpenSSH patched CVE-2016-0777 which allowed a malicious server to read private SSH

keys by combining a memory disclosure vulnerability with errant `memset` and `bzero` memory scrubs [8]; or February 2016 when OpenSSL changed its memory scrubbing technique after discussion in Issue 445 [29]; or Bug 751 in OpenVPN from October 2016 about secret data scrubs being optimized away [35].

Despite developers' awareness of such problems, there is no uniformly-used solution. The CERT C Secure Coding Standard [54] recommends `SecureZeroMemory` as a Windows solution, `memset_s` as a C11 solution, and the volatile data pointer technique as a C99 solution. Unfortunately, each of these solutions has problems. The Windows solution is not cross-platform. For the recommended C11 `memset_s` solution, to the best of our knowledge, there is no standard-compliant implementation. Furthermore, while the CERT solution for C99 solution may prevent most compilers from removing scrubbing operations, the standard does not guarantee its correctness [52]. Furthermore, another common technique, using a volatile function pointer, is not guaranteed to work according to the standard because although the standard requires compilers to access the function pointer, it does not require them to make a call via that pointer [51].

2.3 Existing Approaches

Until recently, the C standard did not provide a way to ensure that a `memset` call is not removed, leaving developers who wanted to clear sensitive memory were left to devise their own techniques. Here we survey eleven security-related open source projects to determine what techniques developers were using to clear memory. In this section, we present the results of our survey. For each technique, we describe how it is intended to work, its **availability** on different platforms, and its **effectiveness** at ensuring that sensitive data is scrubbed. We rate the effectiveness of a technique on a three-level scale:

* **Effective.** Guaranteed to work (barring flaws in implementation).

* **Effective in practice.** Works with all compilation options and on all the compilers we

tested (GCC, Clang, and MSVC), but is not guaranteed in principle.

* **Flawed.** Fails in at least one configuration.

The scrubbing techniques we found can be divided into four groups based on how to they attempt to force memory to be cleared:

* **Rely on the platform.** Use a function offered by the operating system or a library that guarantees memory will be cleared.

* **Disable optimization.** Disable the optimization that removes the scrubbing operation.

* **Hide semantics.** Hide the semantics of the clearing operation, preventing the compiler from recognizing it as a dead store.

* **Force write.** Directly force the compiler to write to memory.

In the remainder of this section, we describe and discuss each technique in detail and conclude with a performance evaluation of each technique. While performance is not the primary consideration when choosing a technique, it is, nevertheless, interesting to observe the wide performance disparity of each.

2.3.1 Platform-Supplied Functions

The easiest way to ensure that memory is scrubbed is to call a function that guarantees that memory will be scrubbed. These *deus ex machina* techniques rely on a platform-provided function that guarantees the desired behavior and lift the burden of fighting the optimizer from the developers' shoulders. Unfortunately, these techniques are not universally available, forcing developers to come up with backup solutions.

Windows SecureZeroMemory

On Windows, `SecureZeroMemory` is designed to be a reliable scrubbing function even in the presence of optimizations. This is achieved by the support from the Microsoft Visual

Studio compiler, which never optimizes out a call to `SecureZeroMemory`. Unfortunately, this function is only available on Windows.

Used in: Kerberos's `zap`, Libsodium's `sodium_memzero`, Tor's `memwipe`.

Availability: Windows platforms.

Effectiveness: Effective.

OpenBSD explicit_bzero

Similarly OpenBSD provides `explicit_bzero`, a optimization-resistant analogue of the BSD `bzero` function. The `explicit_bzero` function has been available in OpenBSD since version 5.5 and FreeBSD since version 11. Under the hood, `explicit_bzero` simply calls `bzero`, however, because `explicit_bzero` is defined in the C standard library shipped with the operating system and not in the compilation unit of the program using it, the compiler is not aware of this and does not eliminate the call to `explicit_bzero`. As discussed in Section 2.3.3, this way of keeping the compiler in the dark only works if definition and use remain separate through compilation and linking. This is the case with OpenBSD and FreeBSD, which dynamically link to the C library at runtime.

Used in: Libsodium's `sodium_memzero`, Tor's `memwipe`, OpenSSH's `explicit_bzero`.

Availability: FreeBSD and OpenBSD.

Effectiveness: Effective (when `libc` is a shared library).

C11 memset_s

Annex K of the C standard (ISO/IEC 9899-2011) introduced the `memset_s` function, declared as

```
errno_t memset_s(void* s, rsize_t smax,
                 int c, rsize_t n);
```

Similar to `memset`, the `memset_s` function sets a number of the bytes starting at address `s`to the byte value `c`. The number of bytes written is the lesser of `smax` or `n`. By analogy to `strncpy`,

the intention of having two size arguments is prevent a buffer overflow when `n` is an untrusted user-supplied argument; setting `smax` to the size allocated for `s` guarantees that the buffer will not be overflowed. More importantly, the standard requires that the function actually write to memory, regardless of whether or not the written values are read.

The use of two size arguments, while consistent stylistically with other `_s` functions, has drawbacks. It differs from the familiar `memset` function which takes one size argument. The use of two arguments means that a programmer can't use `memset_s` as a drop-in replacement for `memset`. It may also lead to incorrect usage, for example, by setting `smax` or `n` to 0, and thus, while preventing a buffer overflow, would fail to clear the buffer as intended.

While `memset_s` seems like the ideal solution, it's implementation has been slow. There may be several reasons for this. First, `memset_s` is not required by the standard. It is part of the optional Appendix K. C11 treats all the function in the Annex K as a unit. That is, if a C library wants to implement the Annex K in a standard-conforming fashion, it has to implement *all* of the functions defined in this annex. At the time of this writing, `memset_s` is not provided by the GNU C Library nor by the FreeBSD, OpenBSD, or NetBSD standard libraries. It's poor adoption and perceived flaws have led to calls for its removal from the standard [50].

Used in: Libsodium's `sodium_memzero`, Tor's `memwipe`, OpenSSH's `explicit_bzero`, CERT's Windows-compliant solution [54].

Availability: No mainstream support.

Effectiveness: Effective.

2.3.2 Disabling Optimization

Since the scrubbing store elimination problem is caused by compiler optimization, it is possible to prevent scrubbing stores from being eliminated by disabling compiler optimization. Dead store elimination is enabled (on GCC and Clang) at optimization level `-O1`, so code compiled with no optimization would retain the scrubbing writes. However, disabling optimization completely can significantly degrade performance, and is eschewed by developers. Alternatively,

some compilers allow optimizations to be enabled individually, so, in principle, a program could be compiled with all optimizations except dead store elimination enabled. However, some optimization passes work better when dead stores have already been eliminated. Also, specifying the whole list of optimization passes instead of a simple optimization level like `O2` is cumbersome.

Many compilers, including Microsoft Visual C, GCC and Clang, provide built-in versions of some C library functions, including `memset`. During compilation, the compiler replaces calls to the C library function with its built-in equivalent to improve performance. In at least one case we found, developers attempted to preserve scrubbing stores by disabling the built-in `memset` intrinsic using the `-fno-builtin-memset` flag. Unfortunately, while this may disable the promotion of standard C library functions to intrinsics, it does not prevent the compiler from understanding the semantics of `memset`. Furthermore, we find that the `-fno-builtin-memset` flag does not not prevent the developer from calling the intrinsic directly, triggering dead store elimination. In particular, starting with glibc 2.3.4 on Linux, defining `_FORTIFY_SOURCE` to be an integer greater than 0 enables additional compile-time bounds checks in common functions like `memset`. In this case, if the checks succeed, the inline definition of `memset` simply calls the built-in `memset`. As a result, the `-fno-builtin-memset` option did not protect scrubbing stores from dead store elimination.

Used in: We are not aware of any programs using this technique.

Availability: Widely available.

Effectiveness: Flawed (not working when newer versions of glibc and GCC are used and optimization level is `O2` or `O3`).

2.3.3 Hiding Semantics

Several scrubbing techniques attempt to hide the semantics of the scrubbing operation from the compiler. The thinking goes, if the compiler doesn't recognize that an operation is clearing memory, it will not remove it.

Separate Compilation

The simplest way to hide the semantics of a scrubbing operation from the compiler is to implement the scrubbing operation (e.g. by simply calling `memset`) in a separate compilation unit. When this scrubbing function is called in a different compilation unit than the defining one, the compiler cannot remove any calls to the scrubbing function because the compiler does not know that it is equivalent to `memset`. Unfortunately, this technique is not reliable when link-time optimization (LTO) is enabled, which can merge all the compilation units into one, giving the compiler a global view of the whole program. The compiler can then recognize that the scrubbing function is effectively a `memset`, and remove it. Thus, to ensure this technique works, the developer needs to make sure that she has the control over how the program is compiled.

Weak Linkage

GCC and some compilers that mimic GCC allow developers to define *weak definitions*. A weak definition of a symbol, indicated by the compiler attribute `__attribute__((weak))`, is a tentative definition that may be replaced by another definition at link time. In fact, the OpenBSD `explicit_bzero` function (Section 2.3.1) uses this technique also:

```
__attribute__((weak)) void
__explicit_bzero_hook(void *buf, size_t len) { }
void explicit_bzero(void *buf, size_t len) {
    memset(buf, 0, len);
    __explicit_bzero_hook(buf, len);
}
```

The compiler can not eliminate the call to `memset` because an overriding definition of `__explicit_bzero_hook` may access `buf`. This way, even if `explicit_bzero` is used in the same compilation unit where it is defined, the compiler will not eliminate the scrubbing operation. Unfortunately, this technique is also vulnerable to link-time optimization. With link-time optimization enabled, the compiler-linker can resolve the final definition of the weak symbol,

determine that it does nothing, and then eliminate the dead store.

Used in: Libsodium's `sodium_memzero`, libressl's `explicit_bzero` [13].

Availability: Available on GCC and Clang.

Effectiveness: Flawed (defeated by LTO).

Volatile Function Pointer

Another popular technique for hiding a scrubbing operation from the compiler is to call the memory scrubbing function via a volatile function pointer. `OPENSSL_cleanse` of OpenSSL 1.0.2, shown below, is one implementation that uses this technique:

```
typedef void *(*memset_t)(void *,int,size_t);

static volatile memset_t memset_func = &memset;

void OPENSSL_cleanse(void *ptr, size_t len) {

    memset_func(ptr, 0, len);

}
```

The C11 standard defines an object of volatile-qualified type as follows:

> An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously. What constitutes an access to an object that has volatile-qualified type is implementation-defined.

The effect of declaring `memset_func` as volatile means that the compiler must read its value from memory each time its used because the value may have changed. The reasoning goes that because the compiler does not know the value of `memset_func` at compile time, it can't recognize the call to `memset` and eliminate it.

We have confirmed that this technique works on GCC, Clang and Microsoft Visual C, and we deem it to be effective. It is worth noting, however, that while the standard requires the

compiler to read the value of `memset_func` from memory, it does *not* require it to call `memset` if it can compute the same result by other means. Therefore, a compiler would be in compliance if it inlined each call to `OPENSSL_cleanse` as:

```
memset_t tmp_fptr = memset_func;
if (tmp_fptr == &memset)
    memset(ptr, 0, len);
else
    tmp_fptr(ptr, 0, len);
```

If the memory pointed to by `ptr` is not read again, then the direct call to `memset`, the semantics of which are known, could be eliminated, removing the scrubbing operation. We know of no compiler that does this and consider such an optimization unlikely.

Used in: OpenSSL 1.0.2's `OPENSSL_cleanse` (also used in Tor and Bitcoin); OpenSSH's `explicit_bzero`, quarkslab's `memset_s` [3].

Availability: Universally available.

Effectiveness: Effective in practice.

Assembly Implementation

Because optimizations often take place at compiler's intermediate representation level, it is possible to hide the semantics of a memory scrubbing operation by implementing it in assembly language. In some cases, this may also be done as a way to improve performance, however, our results indicate that the compiler's built-in intrinsic `memset` performs as well as the assembly implementation we examined. So long as the compiler does not perform assembly-level link-time optimization, this technique is effective at ensuring scrubbing stores are preserved.

Used in: OpenSSL's `OPENSSL_cleanse` (also used by Tor and Bitcoin); Crypto++'s `SecureWipeBuffer`.

Availability: Target-specific.

Effectiveness: Effective.

2.3.4 Forcing Memory Writes

The fourth set of techniques we found attempts to force the compiler to include the store operation without hiding its nature.

Complicated Computation

Several related techniques attempt to force the compiler to overwrite sensitive data in memory by forcing the compiler to carry out a computation. `OPENSSL_cleanse` from OpenSSL prior to version 1.0.2 is one example:

```
unsigned char cleanse_ctr = 0;
void OPENSSL_cleanse(void *ptr, size_t len) {
    unsigned char *p = ptr;
    size_t loop = len, ctr = cleanse_ctr;

    if (ptr == NULL) return;

    while (loop--) {
        *(p++) = (unsigned char)ctr;
        ctr += (17 + ((size_t)p & 0xF));
    }
    p = memchr(ptr, (unsigned char)ctr, len);
    if (p) ctr += (63 + (size_t)p);
    cleanse_ctr = (unsigned char)ctr;
}
```

This function reads and writes the global variable `cleanse_ctr`, which provides varying garbage data to fill the memory to be cleared. Because accesses to the global variable have a global impact on the program, the compiler cannot determine that this function is useless without extensive interprocedural analysis. Since such interprocedural analysis is expensive, the compiler

most likely does not perform it, thus it cannot figure out that `OPENSSL_cleanse` is actually a scrubbing function. However, this particular implementation is notoriously slow. OpenSSL gave up this technique in favor of the volatile function pointer technique (Section 2.3.3) starting with version 1.0.2.

Another way to scrub sensitive data is to simply rerun the computation that accesses sensitive data again. This is used in the musl libc [19] implementation of `bcrypt`, which is a popular password hashing algorithm. musl's `bcrypt` implementation `__crypt_blowfish` calls the hashing function `BF_crypt` twice: the first time it passes the actual password to get the hash, the second time it passes a test password. The second run serves two purposes. First, it is a self-test of the hashing code. `__crypt_blowfish` compares the result of the second run with the hardcoded hash value in the function. If they do not match, there is something wrong in the hashing code. (In fact, the developers of musl libc found a bug in GCC that manifested in their hashing code [9].) Second, the second run of `BF_crypt` can also clear sensitive data left on the stack or in registers by the first run. Since the same function is called twice, the same registers will be used, thus the sensitive data left in registers will be cleared. Since the two calls to `BF_crypt` are in the same scope and the stack pointer points to the same position of the stack before the two calls, the sensitive data left on the stack by the first run should be cleared by the second run. The advantage of this solution is that it clears sensitive data not only on the stack but also in registers.

While the complicated computation technique appears effective in practice, there is no guarantee that a compiler will not someday see through the deception. This technique, especially re-running the computation, has a particularly negative performance impact.

Used in: `OPENSSL_cleanse` from OpenSSL 1.0.1 (also used in Tor and Bitcoin), `crypt_blowfish` from musl libc [19].

Availability: Universal.

Effectiveness: Effective in practice.

Volatile Data Pointer

Another way to force the compiler to perform a store is to access a volatile-qualified type. As noted in Section 2.3.3, the standard requires accesses to objects that have volatile-qualified types to be performed explicitly. If the memory to be scrubbed is a volatile object, the compiler will be forced to preserve stores that would otherwise be considered dead. Cryptography Coding Standard's `Burn` is one of the implementations based on this idea:

```
void burn( void *v, size_t n ) {
  volatile unsigned char *p =
      ( volatile unsigned char * )v;
  while( n-- ) *p++ = 0;
}
```

In the function above, the memory to be scrubbed is written via a pointer-to-volatile `p` in the while loop. We have found that this technique is effective on GCC, Clang, and Microsoft Visual C. Unfortunately, this behavior is *not* guaranteed by the C11 standard: "What constitutes an access to an object that has volatile-qualified type is implementation-defined." This means that, while accessing an object declared volatile is clearly an "access to an object that has volatile-qualified type" (as in the case of the function pointer that is a volatile object), accessing a non-volatile object via pointer-to-volatile may or may not be considered such an access.

Used in: `sodium_memzero` from Libsodium, `insecure_memzero` from Tarsnap, `wipememory` from Libgcrypt, `SecureWipeBuffer` from the Crypto++ library, `burn` from Cryptography Coding Standard [56], David Wheeler's `guaranteed_memset` [61], `ForceZero` from wolfSSL [36], `sudo_memset_s` from sudo [31], and CERT's C99-compliant solution [54].

Availability: Universal.

Effectiveness: Effective in practice.

Memory Barrier

Both GCC and Clang support a memory barrier expressed using an inline assembly statement. The clobber argument `"memory"` tells the compiler that the inline assembly statement may read or write memory that is not specified in the input or output arguments [1]. This indicates to the compiler that the inline assembly statement may access and modify memory, forcing it to keep stores that might otherwise be considered dead. GCC's documentation indicates that the following inline assembly should work as a memory barrier [1]:

```
__asm__ __volatile__(""::::"memory")
```

Our testing shows the above barrier works with GCC, and since Clang also supports the same syntax, one would expect that the barrier above would also work with Clang. In fact, it may remove a `memset` call before such a barrier [5]. We found that Kerberos (more in Section 2.4.2) uses this barrier to implement its scrubbing function, which may be unreliable with Clang. A more reliable way to define memory barrier is illustrated by Linux's `memzero_explicit` below:

```
#define barrier_data(ptr) \
__asm__ __volatile__("": :"r"(ptr) :"memory")


void memzero_explicit(void *s, size_t count) {
    memset(s, 0, count);
    barrier_data(s);
}
```

The difference is the `"r"(ptr)` argument, which makes the pointer to the scrubbed memory visible to the assembly code and prevents the scrubbing store from being eliminated.

Used in: `zap` from Kerberos, `memzero_explicit` from Linux [15].

Availability: Clang and GCC.

Effectiveness: Effective in practice.

2.3.5 Discussion

Our survey of existing techniques indicates that **there is no single best technique for scrubbing sensitive data**. The most effective techniques are those where the integrity of scrubbing operation is guaranteed by the platform. Unfortunately, this means that creating a scrubbing function requires relying on platform-specific functions rather than a standard C library or POSIX function.

Of the remaining techniques, we found that the volatile data pointer, volatile function pointer, and compiler memory barrier techniques are *effective in practice* with the compilers we tested. The first two of these, relying on the volatile storage type, can be used with any compiler but are not guaranteed by the standard. The memory barrier technique is specific to GCC and Clang and its effectiveness may change without notice as it has done already.

2.4 Case Studies

To understand the use of memory scrubbing in practice, we examined the 11 popular security libraries and applications listed in Table 2.1. Our choices were guided by whether or not the code handled sensitive data (e.g. secret keys), availability of the source code and our own judgement of the project's relevance. For each project, we set out to determine whether a memory scrubbing function is **available**, **effective**, and **used consistently** by the projects' developers. We used the latest stable version of each project as of October 9, 2016.

Availability

To determine whether a scrubbing function is available, we manually examined the program source code. All 11 projects used one or more of the techniques described in Section 2.3 to clear sensitive data, and seven of them relied on a combination of at least two techniques.

If a project relied on more than one technique, it automatically chose and used the first technique available on the platform in order of preference specified by the developer. Columns under the *Preference* heading in Table 2.1 show the developer preference order for each technique,

with 1 being highest priority (first chosen if available). The scrubbing techniques listed under the *Preference* heading are: *Win* is Windows' `SecureZeroMemory`, *BSD* is BSD's `explicit_bzero`, *C11* is C11's `memset_s`, *Asm.* is a custom assembly implementation, *Barrier* is the memory barrier technique, *VDP* is the volatile data pointer technique, *VFP* is the volatile function pointer technique, *Comp.* is the complicated computation technique, *WL* is the weak linkage technique, and *memset* is a call to plain `memset`. If a project used a function that can be one of many techniques depending on the version of that function—for example, projects that use OpenSSL's `OPENSSL_cleanse`, which may either be *VFP* or *Comp.* depending on if OpenSSL version $\geq$1.0.2 or $<$1.0.2 is used—the newer version is given a higher preference. An $*$ indicates an incorrectly implemented technique.

For example, Tor uses Windows' `SecureZeroMemory` if available, then BSDs' `explicit_bzero` if available, and so on. Generally, for projects that used them, all chose a platform-supplied function (`SecureZeroMemory`, `explicit_bzero`, or `memset_s`) first before falling back to other techniques. The most popular of the do-it-yourself approaches are the volatile data pointer (VDP) and volatile function pointer (VFP) techniques, with the latter being more popular with projects that attempt to use a platform-provided function first.

Effectiveness

To answer the second question—whether the scrubbing function is effective—we relied on the manual analysis in Section 2.3. If a project used an unreliable or ineffective scrubbing technique in at least one possible configuration, we considered its scrubbing function ineffective, and scored it *flawed*, denoted ○ in the *Score* column. If the scrubbing function was effective and used consistently, we scored it *effective*, denoted ●. If it was effective but not used consistently, we scored it *inconsistent*, denoted ◐.

Consistency

To determine whether a function was used consistently, we instrumented the Clang 3.9 compiler to report instances of dead store elimination where a write is eliminated because the

memory location is not used afterwards. We did not report writes that were eliminated because they were followed by another write to the same memory location, because in this case, the data would be cleared by the second write. Additionally, if sensitive data is small enough to be fit into registers, it may be promoted to a register, which will lead to the removal of the scrubbing store [1]. Since the scrubbing store is not removed in the dead store elimination pass, our tool does not report it. We would argue such removals have less impact on security since the sensitive data is in a register. However, if that register spilled when the sensitive data in it, it may still leave some sensitive data in memory. We compiled each project using this compiler with the same optimization options as in the default build of the project. Then we examined the report generated by our tool and manually identified cases of dead store elimination that removed scrubbing operations.

Of the eleven projects we examined, all of them supported Clang. We note, however, that our goal in this part of our analysis is to identify sites where a compiler *could* eliminate a scrubbing operation, and thus identify sites where sensitive variables were not being cleared as intended by the developer. We then examined each case to determine whether the memory contained sensitive data, and whether dead store elimination took place because a project's own scrubbing function was not used or because the function was ineffective. If cases of the latter, we determined why the function was not effective; these findings are reflected in the results reported in Section 2.3. Columns under the heading *Removed ops.* in Table 2.1 show the number of cases where a scrubbing operation was removed. The *Total* column shows the total number of sites where an operation was removed. The *Sensitive* column shows the number of such operations where we considered the data to be indeed sensitive. (In some cases, the scrubbing function was used to clear data that we did not consider sensitive, such as pointer addresses.) The *Heap*, *Stack*, and *H/S* columns indicate whether or not the cleared memory was allocated on the heap, on the stack stack, or potentially on either heap or stack.

[1] For example, at the end of OpenSSH's `SHA1Transform` function, "`a=b=c=d=e=0;`" is used to scrub sensitive data. Because all the five variables are in virtual registers in the IR form, no store is eliminated in the DSE pass.

Table 2.1. Summary of open source projects' removed scrubbing operations and the scrubbing techniques they use. *Removed ops.* columns show the total number of removed scrubs, the number of removed scrubs dealing with sensitive data, and the locations of memory that failed to be scrubbed. *Preference* columns show the developer preference order for each technique, with 1 being highest priority (first chosen if available). The $*$ in the row for Kerberos indicates that its barrier technique was not implemented correctly; see Section 2.3.4 for discussion. A project's *Score* shows whether its scrubbing implementation is *flawed* (○), *inconsistent* (◐), or *effective* (●).

	Removed ops.					*Preference*										*Score*
Project	*Total*	*Sensitive*	*Heap*	*Stack*	*H/S*	*Win*	*BSD*	*C11*	*Asm.*	*Barrier*	*VDP*	*VFP*	*Comp.*	*WL*	*memset*	
NSS	15	9	3	12	0	-	-	-	-	-	-	-	-	-	1	○
OpenVPN	8	8	2	6	0	-	-	-	-	-	-	-	-	-	1	○
Kerberos	10	2	9	0	1	1	-	-	-	2*	-	-	-	-	3	○
Libsodium	0	0	0	0	0	1	3	2	-	-	5	-	-	4	-	○
Tarsnap	11	10	10	1	0	-	-	-	-	-	1	-	-	-	-	◐
Libgcrypt	2	2	0	2	0	-	-	-	-	-	1	-	-	-	-	◐
Crypto++	1	1	0	1	0	-	-	-	1	-	2	-	-	-	-	◐
Tor	4	0	4	0	0	1	2	3	4	-	-	5	6	-	-	◐
Bitcoin	0	0	0	0	0	-	-	-	1	-	-	2	3	-	-	●
OpenSSH	0	0	0	0	0	-	1	2	-	-	-	3	-	-	-	●
OpenSSL	0	0	0	0	0	-	-	-	1	-	-	2	3	-	-	●

Of the eleven projects examined, four had an effective scrubbing function but did not use it consistently, resulting in a score of *inconsistent*, denoted ◐ in Table 2.1. As the results in Table 2.1 show, **only three of the eleven projects had a scrubbing function that was effective and used consistently.**

We notified the developers of each project that we scored *flawed* or *inconsistent*. For our report to the developers, we manually verified each instance where a scrubbing operation was removed, reporting only valid cases to the developers. Generally, as described below, developers acknowledged our report and fixed the problem. Note that none of the issues resulted in CVEs because to exploit, they must be used in conjunction with a separate memory disclosure bug and these types of bugs are outside the scope of this work.

In the remainder of this section, we report on the open source projects that we analyzed. Our goal is to identify common trends and understand how developers deal with the problem of compilers removing scrubbing operations.

2.4.1 OpenVPN

OpenVPN is an TLS/SSL-based user-space VPN [25]. We tested version 2.3.12. OpenVPN 2.3.12 does not have a reliable memory scrubbing implementation since it uses a `CLEAR` macro which expands to `memset`. We found 8 scrubbing operations that were removed, all of which deal with sensitive data. Each of the removed operations used `CLEAR`, which is not effective.

Sample case

Function `key_method_1_read` in Figure 2.1 is used in OpenVPN's key exchange function to process key material received from an OpenVPN peer. However, the `CLEAR` macro fails to scrub the key on the stack since it is a call to plain `memset`.

Developer response

The issues were reported, although OpenVPN developers were already aware of the problem and had a ticket on their issue tracker for it that was opened 12 days prior to our notification [35]. The patch does not change the `CLEAR` macro since it is used extensively throughout the project, but it does replace many `CLEAR` calls with our recommended fix discussed in Section 2.5 [6].

```
1  /* From openvpn-2.3.12/src/openvpn/basic.h */
2  #define CLEAR(x) memset(&(x), 0, sizeof(x))
3
4  /* From openvpn-2.3.12/src/openvpn/ssl.c */
5  static bool key_method_1_read (struct buffer *buf, struct
6    tls_session *session) {
7
8    struct key key;
9    /* key is allocated on stack to hold TLS session key */
10   ...
11   /* Clean up */
12   CLEAR (key);
13   ks->authenticated = true;
14   return true;
15 }
```

Figure 2.1. A removed scrubbing operation in OpenVPN 2.3.12.

2.4.2 Kerberos

Kerberos is a network authentication protocol that provides authentication for client/server applications by using secret-key cryptography [11]. We tested Kerberos release krb5-1.14.4. The Kerberos memory scrubbing implementation, `zap`, is unreliable. First, it defaults to Windows' `SecureZeroMemory`, which is effective. Otherwise it uses a memory barrier that may not prevent the scrubbing operation from being removed when the code is compiled with Clang (see Section 2.3.4). Finally, if the compiler is not GCC, it uses a function that calls `memset`. While this is more reliable than a macro, `memset` may be removed if LTO is enabled (see Section 2.3.3). Furthermore, even though `zap` is available (and reliable on Windows), plain `memset` is still used throughout the code to perform scrubbing. We found 10 sites where scrubbing was done using `memset`, which is not effective; 2 of these sites deal with sensitive data.

Sample case

Function `free_lucid_key_data` in Figure 2.2 is used in Kerberos to free any storage associated with a lucid key structure (which is typically on the heap) and to scrub all of its sensitive information. However it does so with a call to plain `memset`, which is then removed by the optimizer.

Developer response

The issues have been patched with calls to `zap`. In addition, `zap` has been patched according to our recommended fix discussed in Section 2.5.

```
1 static void free_lucid_key_data(gss_krb5_lucid_key_t *key) {
2   if (key) {
3     if (key->data && key->length) {
4       memset(key->data,0,key->length);
5       xfree(key->data);
6       memset(key,0,sizeof(gss_krb5_lucid_key_t));
7     }
8   }
9 }
```

Figure 2.2. A removed scrubbing operation in Kerberos release krb5-1.14.4.

2.4.3 Tor

Tor provides anonymous communication via onion routing [33]. We tested version 0.2.8.8. Tor defines `memwipe`, which reliably scrubs memory: it uses Windows' `SecureZeroMemory` if available, then `RtlSecureZeroMemory` if available, then BSD's `explicit_bzero`, then `memset_s`, and then `OPENSSL_cleanse`, which is described below. Despite the availability of `memwipe`, Tor still uses `memset` to scrub memory in several places. We found 4 scrubbing operations that were removed, however none dealt with sensitive data.

Sample case

Function `MOCK_IMPL` in Figure 2.3 is used to destroy all resources allocated by a process handle. However, it scrubs the process handle object with `memset`, which is then removed by the optimizer.

Developer response

The bugs were reported and have yet to be patched.

```
1 MOCK_IMPL(void, tor_process_handle_destroy,(process_handle_t
2   *process_handle, int also_terminate_process)) {
3
4   /* process_handle is passed in and allocated on heap to
5    * hold process handle resources */
6   ...
7   memset(process_handle, 0x0f, sizeof(process_handle_t));
8   tor_free(process_handle);
9 }
```

Figure 2.3. A removed scrubbing operation in Tor 0.2.2.8.

2.4.4 OpenSSL

OpenSSL is a popular TLS/SSL implementation as well as a general-purpose cryptographic library [24]. We tested version 1.1.0b. OpenSSL uses `OPENSSL_cleanse` to reliably scrub memory. `OPENSSL_cleanse` defaults to its own assembly implementations in various architectures unless specified otherwise by the `no-asm` flag at configuration. Otherwise, starting with version 1.0.2, it uses the volatile function pointer technique to call `memset`. Prior to version 1.0.2, it used the complicated computation technique. We found no removed scrubbing operations in version 1.1.0b.

2.4.5 NSS

Network Security Services (NSS) is an TLS/SSL implementation that traces its origins to the original Netscape implementation of SSL [20]. We tested version 3.27.1. NSS does not have a reliable memory scrubbing implementation since it either calls `memset` or uses the macro `PORT_Memset`, which expands to `memset`. We found 15 scrubbing operations that were removed, 9 of which deal with sensitive data. Of the 15 removed operations, 6 were calls to `PORT_Memset` and 9 were calls to plain `memset`.

Sample case

Function `PORT_ZFree` is used throughout the NSS code for freeing sensitive data and is based on function `PORT_ZFree_stub` in Figure 2.4. However `PORT_ZFree_stub`'s call to `memset` fails to scrub the pointer it is freeing.

Developer response

The bugs have been reported and Mozilla Security forwarded them to the appropriate team, however they have not yet been patched.

```
1 extern void PORT_ZFree_stub(void *ptr, size_t len) {
2   STUB_SAFE_CALL2(PORT_ZFree_Util, ptr, len);
3   memset(ptr, 0, len);
4   return free(ptr);
5 }
```

Figure 2.4. A removed scrubbing operation in NSS 3.27.1.

2.4.6 Libsodium

Libsodium is a cross-platform cryptographic library [14]. We tested version 1.0.11. Libsodium defines `sodium_memzero`, which does not reliably scrub memory. First, it defaults to Windows' `SecureZeroMemory`, then `memset_s`, and then BSD's `explicit_bzero` if available, which are all reliable. Then if weak symbols are supported, it uses a technique based on weak linkage, otherwise it uses the volatile data pointer technique. Techniques based on weak linkage are not reliable, because they can be removed during link-time optimization. All memory scrubbing operations used `sodium_memzero`, and since `Libsodium` is not compiled with link-time optimization, no scrubbing operations using `sodium_memzero` were removed.

2.4.7 Tarsnap

Tarsnap is a online encrypted backup service whose client source code is available [32]. We tested version 1.0.37. Tarsnap's memory scrubbing implementation, called `insecure_memzero`, uses the volatile data pointer scrubbing technique. Although

`insecure_memzero` is an effective scrubbing function, Tarsnap does not use it consistently. We found 10 cases where `memset` was used to scrub memory instead of `insecure_memzero` in its `keyfile.c`, which handles sensitive data.

Sample case

Function `read_encrypted` in Figure 2.5 attempts to scrub a buffer on the heap containing a decrypted key. It is used throughout the project for reading keys from a Tarsnap key file. However, instead of using `insecure_memzero`, it uses plain `memset`, and is thus removed by the optimizer.

Developer response

Out of the 11 reported issues, the 10 in `keyfile.c` were already patched on July 2, 2016 but were not in the latest stable version. The one non-security issue does not require a patch, since the removed `memset` was redundant as `insecure_memzero` is called right before it.

```
1  static int read_encrypted(const uint8_t * keybuf, size_t
2    keylen, uint64_t * machinenum, const char * filename,
3    int keys) {
4
5    uint8_t * deckeybuf;
6    /* deckeybuf is allocated on heap to hold decrypted key */
7    ...
8    /* Clean up */
9    memset(deckeybuf, 0, deckeylen);
10   free(deckeybuf);
11   free(passwd);
12   free(pwprompt);
13   return (0);
14 }
```

Figure 2.5. A removed scrubbing operation in Tarsnap 1.0.37.

2.4.8 Libgcrypt

Libgcrypt is a general purpose cryptographic library used by GNU Privacy Guard, a GPL-licensed implementation of the PGP standards [12]. We tested version 1.7.3. Libgcrypt defines `wipememory`, which is a reliable way of scrubbing because it uses the volatile data pointer technique. However, despite `wipememory`'s availability and reliability, `memset` is still used to

scrub memory in several places. We found 2 cases where scrubs were removed, and for both, `memset` is used to scrub sensitive sensitive data instead of `wipememory`.

Sample case

Function `invert_key` in Figure 2.6 is used in Libgcrypt's IDEA implementation to invert a key for its key setting and block decryption routines. However, `invert_key` uses `memset` to scrub a copy of the IDEA key on the stack, which is removed by the optimizer.

Developer response

The bugs have been patched with calls to `wipememory`.

```
1 static void invert_key(u16 *ek, u16 dk[IDEA_KEYLEN]) {
2   u16 temp[IDEA_KEYLEN];
3   /* temp is allocated on stack to hold inverted key */
4   ...
5   memcpy(dk, temp, sizeof(temp));
6   memset(temp, 0, sizeof(temp));
7 }
```

Figure 2.6. A removed scrubbing operation in Libgcrypt 1.7.3.

2.4.9 Crypto++

Crypto++ is a C++ class library implementing several cryptographic algorithms [7]. We tested version 5.6.4. Crypto++ defines `SecureWipeBuffer`, which reliably scrubs memory by using custom assembly if the buffer contains values of type `byte`, `word16`, `word32`, or `word64`; otherwise it uses the volatile data pointer technique. Despite the availability of `SecureWipeBuffer`, we found one scrubbing operation dealing with sensitive data that was removed because it used plain `memset` rather than its own `SecureWipeBuffer`.

Sample case

The `UncheckedSetKey` function, shown in Figure 2.7, sets the key for a CAST256 object. `UncheckedSetKey` uses plain `memset` to scrub the user key on the stack, which is removed by the optimizer.

Developer response

The bug was patched with a call to `SecureWipeBuffer`.

```
1 void CAST256::Base::UncheckedSetKey(const byte *userKey,
2   unsigned int keylength, const NameValuePairs &) {
3
4   AssertValidKeyLength(keylength);
5   word32 kappa[8];
6   /* kappa is allocated on stack to hold user key */
7   ...
8   memset(kappa, 0, sizeof(kappa));
9 }
```

Figure 2.7. A removed scrubbing operation in Crypto++ 5.6.4.

2.4.10 Bitcoin

Bitcoin is a cryptocurrency and payment system [4]. We tested version 0.13.0 of the Bitcoin client. The project defines `memory_cleanse`, which reliably scrubs memory by using `OPENSSL_cleanse`, described below. The source code uses `memory_cleanse` consistently; we found no removed scrubbing operations.

2.4.11 OpenSSH

OpenSSH is a popular implementation of the SSH protocol [23]. We tested version 7.3. OpenSSH defines its own `explicit_bzero`, which is a reliable way of scrubbing memory: it uses BSD's `explicit_bzero` if available, then `memset_s` if available. If neither are available, it uses the volatile function pointer technique to call `bzero`. We found no removed scrubbing operations.

2.4.12 Discussion

Our case studies lead us to two observations. First, **there is no single accepted scrubbing function.** Each project mixes its own cocktail using existing scrubbing techniques, and there is no consensus on which ones to use. Unfortunately, as we discussed in Section 2.3, some of the scrubbing techniques are flawed or unreliable, making scrubbing functions that rely on

such techniques potentially ineffective. To remedy this state of affairs, we developed a single memory scrubbing technique that combines the best techniques into a single function, described in Section 2.5.

Second, even when the project has reliable scrubbing function, **developers do not use their scrubbing function consistently.** In 4 of the 11 projects we examined, we found cases where developers called `memset` instead of their own scrubbing function. To address this, we developed a scrubbing-safe dead-store elimination pass that defensively compile bodies of code, as discussed in Section 2.6.

2.5 Universal Scrubbing Function

As we saw in Section 2.3, there is no single memory scrubbing technique that is both universal and guaranteed. In the next section, we propose a compiler-based solution based on Clang, that protects scrubbing operations from dead-store elimination. In many cases, however, the developer can't mandate a specific compiler and must resort to imperfect techniques to protect scrubbing operations from the optimizer. To aid developers in this position, we developed our own scrubbing function, called `secure_memzero`, that combines the best effective scrubbing techniques in a simple implementation. Specifically, our implementation supports:

* Platform-provided scrubbing functions (`SecureZeroMemory` and `memset_s`) if available,
* The memory barrier technique if GCC or Clang are used to compile the source, and
* The volatile data pointer technique and the volatile function pointer technique.

Our `secure_memzero` function is implemented in a single header file `secure_memzero.h` that can be included in a C/C++ source file. The developer can specify an order of preference in which an implementation will be chosen by defining macros before including `secure_memzero.h`. If the developer does not express a preference, we choose the first available implementation in the order given above: platform-provided function if available, then memory barrier on GCC and

Clang, then then volatile data pointer technique. Our defaults reflect what we believe are the best memory scrubbing approaches available today.

We have released our implementation into the public domain, allowing developers to use our function regardless of their own project license. We plan to keep our implementation updated to ensure it remains effective as compilers evolve. The current version of `secure_memzero.h` is available at https://compsec.sysnet.ucsd.edu/secure_memzero.h and is shown in Appendix A.1.

2.6 Scrubbing-Safe DSE

While we have tested our `secure_memzero` function with GCC, Clang, and Microsoft Visual C, by its very nature it cannot *guarantee* that a standard-conforming compiler will not remove our scrubbing operation. To address these cases, we implemented a scrubbing-safe dead store elimination option in Clang 3.9.0.

2.6.1 Inhibiting Scrubbing DSE

Our implementation works by identifying all stores that may be explicit scrubbing operations and preventing the dead store elimination pass from eliminating them. We consider a store, either a `store` IR instruction, or a call to LLVM's `memset` intrinsic, to be a potential scrubbing operation if

* The stored value is a constant,
* The number of bytes stored is a constant, and
* The store is subject to elimination because the variable is going be out of scope without being read.

The first two conditions are based on our observation how scrubbing operations are performed in the real code. The third allows a store that is overwritten by a later one to the same location before being read to be eliminated, which improves the performance. We note that our techniques

preserves all dead stores satisfying the conditions above, regardless of whether the variables is considered sensitive or not. This may introduce false positives, dead stores to non-sensitive variables in memory that are preserved because they were considered potential scrubbing operations by our current implementation. We discuss the performance impact of our approach in Section 2.6.2.

It is worth considering an alternative approach to ensuring that sensitive data is scrubbed: The developer could explicitly annotate certain variables as *secret*, and have the compiler ensure that these variables are zeroed before going out of scope. This would automatically protect sensitive variables without requiring the developer to zero them explicitly. It would also eliminate potential false positives introduced by our approach, because only sensitive data would be scrubbed. Finally, it could also ensure that spilled registers containing sensitive data are zeroed, something our scrubbing-safe DSE approach does not do (see Section 2.7 for a discussion of this issue).

We chose our approach because it does not require *any* changes to the source code. Since developers are already aware of the need to clear memory, we rely on scrubbing operations already present in the code and simply ensure that they are not removed during optimization. Thus, our current approach is compatible with legacy code and can protect even projects that do not use a secure scrubbing function, provided the sensitive data is zeroed after use.

2.6.2 Performance

Dead store elimination is a compiler optimization intended to reduce code size and improve performance. By preserving certain dead stores, we are potentially preventing a useful optimization from improving the quality emitted code and improving performance. To determine whether or not this the case, we evaluated the performance of our code using the SPEC 2006 benchmark. We compiled and ran the SPEC 2006 benchmark under four compiler configurations: `-O2` only, `-O2` and `-fno-builtin-memset`, `-O2` with DSE disabled, and `-O2` with our scrubbing-safe DSE. In each case, we used Clang 3.9.0, modified to allow us to disable DSE completely

or to selectively disable DSE as described above. Note that `-fno-builtin-memset` is *not* a reliable means of protecting scrubbing operations, as discussed in Section 2.3.2. The benchmark was run on a Ubuntu 16.04.1 server with an Intel Xeon Processor X3210 and 4GB memory.

Our results indicate that the performance if our scrubbing-safe DSE option is within 1% of the base case (`-O2` only). This difference is well within the variation of the benchmark; re-running the same tests yielded differences of the same order. Disabling DSE completely also did not affect performance by more than 1% over base in all but one case (`483.xalancbmk`) where it was within 2%. Finally, with the exception of the `403.gcc` benchmark, disabling built-in memset function also does not have a significant adverse effect on performance. For the `403.gcc` benchmark, the difference was within 5% of base.

2.7 Discussion

It is clear that, while the C standard tries to help by defining `memset_s`, in practice the C standard does not help. In particular, `memset_s` is defined in the optional Annex K, which is rarely implemented. Developers are then left on their own to implement versions of secure memset, and the most direct solution uses the volatile quantifier. But here again, the C standard does not help, because the corner cases of the C standard actually give the implementation a surprising amount of leeway in defining what constitutes a volatile access. As a result, any implementation of a secure memset based on the volatile qualifier is guarantee to work with every standard-compliant compiler.

Second, it's very tricky in practice to make sure that a secure scrubbing function works well. Because an incorrect implementation does not break any functionality, it cannot be caught by automatic regression tests. The only reliable way to test whether an implentation is correct or not is to manually check the generated binary, which can be time-consuming. What's worse, a seemingly working solution may turn out to be insecure under a different combination of platform, compiler and optimization level, which further increases the cost to test an implementation. In

fact, as we showed in Section 2.4.2, developers did make mistakes in the implementing of secure scrubbing functions. This is why we implemented `secure_memzero` and tested it on Ubuntu, OpenBSD and Windows with GCC and Clang. We released it into the public domain so that developers can use it freely and collaborate to adapt it to future changes to the C standard, platforms or compilers.

Third, even if a well-implemented secure scrubbing function is available, developers will forget to use it, instead using the standard `memset` which is removed by the compiler. For example, we found this happened in Crypto++ (Section 2.4.9). This observation makes compiler-based solutions, for example the secure DSE, more attractive because they do not depend on developers correctly calling the right scrubbing function.

Finally, it's important to note that sensitive data may still remain in on the stack even after its primary storage location when it is passed as argument or spilled (in registers) onto the stack. Addressing this type of data leak requires more extensive support from the compiler.

2.8 Related Work

D'Silva *et al.* [44] use the term *correctness-security gap* to describe the gap between the traditional notion of compiler correctness and the correctness notion that a security-conscious developers might have. They found instances of a correctness-security gap in several optimizations, including dead store elimination, function inlining, code motion, common subexpression elimination, and strength reduction.

Lu *et al.* [48] investigated an instance of this gap in which the compiler introduces padding bytes in data structures to improve performance. These padding bytes may remain uninitialized and thus leak data if sent to the outside world. By looking for such data leakage, they found previously undiscovered bugs in the Linux and Android kernels.

Wang *et al.* [60] explored another instance of the correctness-security gap: compilers sometimes remove code that has undefined behavior that, in some cases, includes security checks.

They developed a static checker called STACK that identifies such code in C/C++ programs and they used it to uncover 160 new bugs in commonly deployed systems.

Laurent *et al.* [55] described a few instances of the correctness-security gap with a focus on how compilers may affect constant-time selection code and sensitive data scrubbing operations. The authors argued that adding explicit support to compilers is an effective way to bridge the correctness-security gap. As examples, they implemented compiler supports for preventing constant-time selection and stack sensitive erasure code from being tampered with on the LLVM framwork.

While this chapter examines how programmers handle the correctness-security gap introduced by aggressive dead store elimination, the soundness and security of dead store elimination has been studied formally [64, 49]. However, even a provably-correct implementation of dead store elimination may still undermine software security due to the correctness-security gap.

Deng *et al.* [42] proposed a new secure dead store elimination algorithm on a simple toy programming language and proved its correctness. Variables in this language are categorized into high security variables and low security variables, and with the help of a taint tracking analysis, the algorithm ensures that dead stores to tainted variables will not be removed if doing so leaks information.

2.9 Conclusion

Developers have known that compiler optimizations may remove scrubbing operations for some time. To combat this problem, many implementations of secure memset have been created. In this paper, we surveyed the existing solutions, analyzing the assumptions, advantages and disadvantages of them. Also, our case studies have shown that real world programs still have unscrubbed sensitive data, due to incorrect implementation of secure scrubbing function as well as from developers simply forgetting to use the secure scrubbing function. To solve

the problem, we implemented the secure DSE, a compiler-based solution that keeps scrubbing operations while remove dead stores that have no security implications, and `secure_memzero`, a C implementation that have been tested on various platforms and with different compilers.

Acknowledgements

Chapter 2 is an adapted reprint of the material as it appears in Dead Store Elimination (Still) Considered Harmful in USENIX Security 2017. Yang, Zhaomo; Johannesmeyer, Brian; Olesen, Anders T.; Lerner, Sorin; Levchenko, Kirill., Proceedings of the USENIX Security, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Conclusion

As a crucial part of software development, compilers have significant impact on various qualities of the generated code such as correctness, performance, size, and security. In this dissertation, we examined the security impact of compilers from both the positive and negative directions.

On the one hand, compilers present great opportunities to enhance software security. Compiler-based security schemes have become increasingly popular in the recent years because they can retrofit security policies into potentially vulnerable programs with low required development efforts. In this dissertation, we explored the opportunity of using compilers to secure C++ programs running on resource-constrained systems. Specifically, we pushed the vtable interleaving idea originally from the BKL scheme [39] a step further to fit the space constraint of resource-constrained systems: instead of interleaving whole vtables, we only interleave vtable entries that may be used. In addition, our advanced scheme provides higher protection precision due to our advanced vtable ordering algorithms. Our scheme shows the great potential of compiler-base security schemes to secure software when the special requirements of the targeted software are taken into consideration.

On the other hand, even benign, correctly implemented compilers may harm software security. On this front, we investigated the potential negative impact of Dead Store Elimination (DSE) optimization. Although this phenomenon is considered well-known among developers, our

survey of the existing DSE-circumvention techniques and the case study of open source security programs show that the general understanding of this problem was inadequate. Our results show that the negative security impact of compilers, and the methods for circumventing them, are not sufficiently studied. A good understanding of any negative security effect of compilers normally requires a systematic analysis of the compiler internals, the language specifications, and the nuances of different platforms and tool chains. However, due to the lack of research in this area, when it comes to negative security impact of compilers, many developers have to resort to anecdotes posted at question-and-answer sites like StackOverflow or brief technical blogs, which often have incomplete, or even false, information. Clearly, there is a burgeoning need for additional research in this area.

Appendix A

The current version of `secure_memzero`

Listing A.1. Current version of `secure_memzero`.

```
1  // secure_memzero.h version 1 (October 29, 2016)
2  //
3  // This code is released into the public domain.
4  //
5  // THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
6  // INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
7  // FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
8  // AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
9  // OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
10 // SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
11 // INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
12 // CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
13 // ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
14 // POSSIBILITY OF SUCH DAMAGE.
15
16 // The secure_memzero macro/function attempts to ensure that an optimizing
17 // compiler does not remove the intended operation if cleared memory is not
18 // accessed again by the program. There are several known ways of doing this,
19 // however no single one is both universally available and absolutely guranteed
20 // by the standard. The following code defines secure_memzero as a macro or
21 // function using one of the known alternatives. The choice of implementation
22 // can be controlled by defining a preprocessor macro of the form SMZ_impl,
23 // where <impl> is one of the defined implementation names. SMZ_impl should
24 // expand to an integer indicating the dgeree of preference for the
25 // implementation, where numerically higher values indicate greater preference.
26 // Defining SMZ_impl to be 0 disables the implementation even if it is
```

```
27 // available. Not defining any SMZ_impl will result in default (safe) behavior.
28 //
29 // The following implementations may be used.
30 //
31 // SMZ_SECUREZEROMEMORY
32 // Uses the SecureZeroMemory macro/function on Windows. Requires a Windows
33 // environment (_WIN32 must be defined).
34 //
35 // SMZ_ASM_BARRIER
36 // Uses a compiler memory barrier to force the results of a memset to be
37 // committed to memory. Has been tested to work on:
38 // - Clang 3.9.0 at all optimization levels.
39 // - GCC 6.2 at all optimization levels.
40 //
41 // SMZ_MEMSET_S
42 // Uses the C11 function memset_s. Currently not available on many platforms.
43 // Note that if you want this option, you have to set __STDC_WANT_LIB_EXT1__
44 // to 1 before including string.h or any file that includes string.h in a
45 // compilation unit that includes this header.
46 //
47 // SMZ_VDATAPTR
48 // Uses the volatile data pointer technique to zero one byte at a time. This is
49 // not guaranteed to work by the C standard, which does not require access to
50 // non-volatile objects via a pointer-to-volatile to be treated as a volatile
51 // access. However, it is known to work on the following compilers:
52 // - Clang 3.9.0 at all optimization levels.
53 // - GCC 6.2 at all optimization levels.
54 //
55 // SMZ_VFUNCPTR
56 // Uses the volatile function pointer technique to call memset. This is not
57 // guaranteed to work by the C standard, which does not require the pointed-to
58 // function to be called. However, it is known to work on the following
59 // compilers:
60 // - Clang 3.9.0 at all optimization levels.
61 // - GCC 6.2 at all optimization levels.
62
63 // The remainder of this file implements the selection logic using the
64 // specified compile-time preferences.
65
66 #ifndef _SECURE_MEMZERO_H_
```

```
67  #define _SECURE_MEMZERO_H_
68
69  // STEP 1. Set default preference for all implementations to 1.
70
71  #ifndef SMZ_SECUREZEROMEMORY
72  #define SMZ_SECUREZEROMEMORY 1
73  #endif
74
75  #ifndef SMZ_MEMSET_S
76  #define SMZ_MEMSET_S 1
77  #endif
78
79  #ifndef SMZ_ASM_BARRIER
80  #define SMZ_ASM_BARRIER 1
81  #endif
82
83  #ifndef SMZ_VDATAPTR
84  #define SMZ_VDATAPTR 1
85  #endif
86
87  #ifndef SMZ_VFUNCPTR
88  #define SMZ_VFUNCPTR 1
89  #endif
90
91  // STEP 2. Check which implementations are available and include any necessary
92  // header files.
93
94  #if SMZ_SECUREZEROMEMORY > 0
95  #ifdef _WIN32
96  #include <windows.h>
97  #else
98  #undef SMZ_SECUREZEROMEMORY
99  #define SMZ_SECUREZEROMEMORY 0
100 #endif
101 #endif
102
103 #if SMZ_MEMSET_S > 0
104 #if defined(__STDC_WANT_LIB_EXT1__) && (__STDC_WANT_LIB_EXT1__ != 1)
105 #undef SMZ_MEMSET_S
106 #define SMZ_MEMSET_S 0
```

```
#endif
#if SMZ_MEMSET_S > 0
#ifndef __STDC_WANT_LIB_EXT1__
// Must come before first include of string.h
#define __STDC_WANT_LIB_EXT1__ 1
#endif
#include <string.h>
#ifndef __STDC_LIB_EXT1__
#undef SMZ_MEMSET_S
#define SMZ_MEMSET_S 0
#endif
#endif
#endif

#if !defined(__GNUC__) && !defined(__clang__)
#undef SMZ_ASM_BARRIER
#define SMZ_ASM_BARRIER 0
#endif

#if SMZ_VFUNCPTR > 0
#include <string.h>
#endif

// STEP 3. Calculate highest preference.

#define SMZ_PREFERENCE 0

#if SMZ_PREFERENCE < SMZ_SECUREZEROMEMORY
#undef SMZ_PREFERENCE
#define SMZ_PREFERENCE SMZ_SECUREZEROMEMORY
#endif

#if SMZ_PREFERENCE < SMZ_MEMSET_S
#undef SMZ_PREFERENCE
#define SMZ_PREFERENCE SMZ_MEMSET_S
#endif

#if SMZ_PREFERENCE < SMZ_ASM_BARRIER
#undef SMZ_PREFERENCE
#define SMZ_PREFERENCE SMZ_ASM_BARRIER
```

```
147 #endif
148
149 #if SMZ_PREFERENCE < SMZ_VDATAPTR
150 #undef SMZ_PREFERENCE
151 #define SMZ_PREFERENCE SMZ_VDATAPTR
152 #endif
153
154 #if SMZ_PREFERENCE < SMZ_VFUNCPTR
155 #undef SMZ_PREFERENCE
156 #define SMZ_PREFERENCE SMZ_VFUNCPTR
157 #endif
158
159 // STEP 4. Make sure we have something chosen.
160
161 #if SMZ_PREFERENCE <= 0
162 #error No secure_memzero implementation available
163 #endif
164
165 // STEP 5. Use implementation with highest preference. Ties are broken in
166 // favor of implementations appearing first, below.
167
168 #if SMZ_PREFERENCE == SMZ_SECUREZEROMEMORY
169 #define secure_memzero(ptr,len) SecureZeroMemory((ptr),(len))
170
171 #elif SMZ_PREFERENCE == SMZ_MEMSET_S
172 #define secure_memzero(ptr,len) memset_s((ptr),(len),0,(len))
173
174 #elif SMZ_PREFERENCE == SMZ_ASM_BARRIER
175 #define secure_memzero(ptr,len) do { \
176         memset((ptr),0,(len)); \
177         __asm__ __volatile__("" ::"r"(ptr): "memory"); \
178 } while (0)
179
180 #elif SMZ_PREFERENCE == SMZ_VDATAPTR
181 static void secure_memzero(void * ptr, size_t len) {
182         volatile char * p = ptr;
183         while (len--) *p++ = 0;
184 }
185
186 #elif SMZ_PREFERENCE == SMZ_VFUNCPTR
```

```
187 static void * (* volatile _smz_memset_fptr)(void*,int,size_t) = &memset;
188 static void secure_memzero(void * ptr, size_t len) {
189         _smz_memset_fptr(ptr, 0, len);
190 }
191
192 #endif
193
194 #endif // _SECURE_MEMZERO_H_
```

Bibliography

[1] 6.45.2 Extended Asm - Assembler Instructions with C Expression Operands. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

[2] 8537 – Optimizer Removes Code Necessary for Security. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537.

[3] A glance at compiler internals: Keep my memset. http://blog.quarkslab.com/a-glance-at-compiler-internals-keep-my-memset.html.

[4] Bitcoin: Open source P2P money. https://bitcoin.org/.

[5] Bug 15495 - dead store pass ignores memory clobbering asm statement. https://bugs.llvm.org/show_bug.cgi?id=15495.

[6] Changeset 009521a. https://community.openvpn.net/openvpn/changeset/009521ac8ae613084b23b9e3e5dc4ebeccd4c6c8/.

[7] Crypto++ library. https://www.cryptopp.com/.

[8] CVE-2016-0777. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0777.

[9] GCC Bugzilla - Bug 26587. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=26587.

[10] Issue 464797: Deploy -fsanitize=cfi-vcall on linux. https://bugs.chromium.org/p/chromium/issues/detail?id=464797. Accessed: 2019-08-22.

[11] Kerberos - The Network Authentication Protocol. https://web.mit.edu/kerberos/.

[12] Libgcrypt. https://www.gnu.org/software/libgcrypt/.

[13] Libressl. https://www.libressl.org/.

[14] libsodium - A modern and easy-to-use crypto library. https://github.com/jedisct1/libsodium.

[15] The linux kernel archives. https://www.kernel.org/.

[16] Llvm control flow integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html. Accessed: 2019-07-24.

[17] Memory protection technologies. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457155(v=technet.10). Accessed: 2019-08-22.

[18] Millions of binaries later: a look into linux hardening in the wild. https://capsule8.com/blog/millions-of-binaries-later-a-look-into-linux-hardening-in-the-wild/. Accessed: 2019-08-22.

[19] musl libc. https://www.musl-libc.org/.

[20] Network Security Services - Mozilla. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[21] Octane 2.0 javascript benchmark. http://chromium.github.io/octane. Accessed: 2019-08-22.

[22] Omnibus f4 sd. https://docs.px4.io/v1.9.0/en/flight_controller/omnibus_f4_sd.html. Accessed: 2019-09-01.

[23] OpenSSH. http://www.openssh.com/.

[24] OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[25] OpenVPN - Open Source VPN. https://openvpn.net/.

[26] The pax team's address space layout randomization. https://pax.grsecurity.net/docs/aslr.txt. Accessed: 2019-08-22.

[27] Pixhawk 3 pro. https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk3_pro.html. Accessed: 2019-09-01.

[28] Px4. https://px4.io/. Accessed: 2019-09-01.

[29] Reimplement non-asm OPENSSL_cleanse(). https://github.com/openssl/openssl/pull/455.

[30] Secure an android device: Control flow integrity. https://source.android.com/devices/tech/debug/cfi. Accessed: 2019-08-22.

[31] Sudo. https://www.sudo.ws/.

[32] Tarsnap - Online backups for the truly paranoid. http://www.tarsnap.com/.

[33] Tor Project: Anonymity Online. https://www.torproject.org.

[34] Vendicator. http://www.angelfire.com/sk/stackshield/info.html. Accessed: 2019-08-22.

[35] When erasing secrets, use a memset() that's not optimized away. https://community.openvpn.net/openvpn/ticket/751.

[36] WolfSSL - Embedded SSL Library for Applications, Devices, IoT, and the Cloud. https://www.wolfssl.com.

[37] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of CCS 2005*, New York, NY, USA, 2005.

[38] G. Barthe, B. Grgoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic constant-time. In *Proc. of CSF 2018*, Oxford, UK, 2018.

[39] D. Bounov, R. G. Kici, and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *Proc. of NDSS 2016*, San Diego, CA, USA, 2016.

[40] N. Burow, D. McKee, S. A. Carr, and M. Payer. Cfixx: Object type integrity for c++ virtual dispatch. In *Proc. of NDSS 2018*, San Diego, CA, USA, 2018.

[41] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of USENIX Security 1998*, San Antonio, TX, USA, 1998.

[42] Chaoqiang Deng and Kedar S. Namjoshi. Securing a compiler transformation. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 170–188, 2016.

[43] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee. Efficient protection of path-sensitive control security. In *Proc. of USENIX Security 2017*, Vancouver, BC, Canada, 2017.

[44] Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optimization. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 73–87, 2015.

[45] X. Fan, Y. Sui, X. Liao, and J. Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for c++. In *Proc. of SIGSOFT 2017*, Santa Barbara, CA, USA, 2017.

[46] D. Jang, Z. Tatlock, and S. Lerner. Safedispatch: Securing c++ virtual callsfrom memory corruption attacks. In *Proc. of NDSS 2014*, San Diego, CA, USA, 2014.

[47] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proc. of OSDI 2014*, Broomfield, CO, USA, 2014.

[48] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 920–932, New York, NY, USA, 2016. ACM.

[49] William Mansky, Dmitri Garbuzov, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proc. of CAV 2015*, San Francisco, CA, USA, 2015.

[50] Carlos O'Donell and Martin Sebor. Updated Field Experience With Annex K — Bounds Checking Interfaces. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm, September 2015.

[51] Colin Percival. Erratum. http://www.daemonology.net/blog/2014-09-05-erratum.html.

[52] Colin Percival. How to zero a buffer. http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html.

[53] A. Prakash, X. Hu, and H. Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Proc. of NDSS 2015*, San Diego, CA, USA, 2015.

[54] Robert Seacord. *The CERT C Secure Coding Standard*. Addison Wesley, 2009.

[55] L. Simon, D. Chisnall, and R. Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *Proc. of EuroS P 2018*, London, United Kingdom, 2018.

[56] Cryptographic Coding Standard. Coding rules. https://cryptocoding.net/index.php/Coding_rules#Clean_memory_of_secret_data.

[57] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8).

[58] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proc. of USENIX Security 2014*, San Diego, CA, USA.

[59] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proc. of SOSP 2013*, Farminton, Pennsylvania, USA.

[60] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, USA, 2013. ACM.

[61] David Wheeler. Specially protect secrets (passwords and keys) in user memory. https://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/protect-secrets.html.

[62] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. Vtrust: Regaining trust on virtual calls. In *Proc. of NDSS 2016*, San Diego, CA, USA, 2016.

[63] C. Zhang, C. Song, K. Chen, Z. Chen, and D. Song. Vtint: Defending virtual function tables integrity. In *Proc. of NDSS 2015*, San Diego, CA, USA, 2015.

[64] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proc. of POPL 2012*, Philadelphia, PA, USA, 2012.