CFIXX: Object Type Integrity for C++

Nathan Burow
Purdue University
nburow@purdue.edu

Derrick McKee
Purdue University
mckee15@purdue.edu

Scott A. Carr
Purdue University
carr27@purdue.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

***Abstract*—C++ relies on object type information for dynamic dispatch and casting. The association of type information to an object is implemented via the virtual table pointer, which is stored in the object itself. As C++ has neither memory nor type safety, adversaries may therefore overwrite an object's type. If the corrupted type is used for dynamic dispatch, the attacker has hijacked the application's control flow. This vulnerability is widespread and commonly exploited. Firefox, Chrome, and other major C++ applications are network facing, commonly attacked, and make significant use of dynamic dispatch. Control-Flow Integrity (CFI) is the state of the art policy for efficient mitigation of control-flow hijacking attacks. CFI mechanisms determine *statically* (i.e., at compile time) the set of functions that are valid at a given call site, based on C++ semantics. We propose an orthogonal policy, *Object Type Integrity* (OTI), that dynamically tracks object types. Consequently, instead of allowing a set of targets for each dynamic dispatch on an object, only the single, correct target for the object's type is allowed.**

To show the efficacy of OTI, we present CFIXX, which enforces OTI. CFIXX enforces OTI by *dynamically* tracking the type of each object and enforcing its integrity against arbitrary writes. CFIXX has minimal overhead on CPU bound applications such as SPEC CPU2006 — 4.98%. On key applications like Chromium, CFIXX has negligible overhead on JavaScript benchmarks: 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream. We show that CFIXX can be deployed in conjunction with CFI, providing a significant security improvement.

I. Introduction

Web browsers are among the most commonly used and most complex applications running on today's systems. Browsers are responsible for correctly processing arbitrary input in the form of web pages, including large and complex web applications with a threat model that includes an adversary with (restricted) code execution through JavaScript. Consequently, browsers are some of the most commonly attacked programs. The major web browsers (Firefox, Chrome, Safari, and Internet Explorer/Edge) are written primarily in C++, which does not enforce type safety and is prone to use after free (UaF) and other memory safety errors. Attackers now increasingly use type safety and UaF vulnerabilities to hijack the program's control flow, frequently by redirecting dynamic dispatch [5]. This is a form of code-reuse, and indeed Return Oriented Programming (ROP) [32], [8] attacks have been specialized for C++ [33]. To the best of our knowledge, no existing work fully explores how vulnerable C++'s dynamic dispatch is to these attacks, or is capable of stopping the full spectrum of attacks on dynamic dispatch.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23279
www.ndss-symposium.org

C++ uses dynamic dispatch to implement polymorphism. A key feature of polymorphism is that objects which inherit from a base class can be up cast to that base class, while still retaining their own implementations of any *virtual* methods. Put another way, an object's underlying type — the class it was allocated as — determines the behavior of dynamic dispatch. Consequently, one call site on a base class can have different behavior depending on the type of the object at runtime. Dynamic dispatch is used to determine which implementation of the virtual method should be invoked. C++ relies on virtual table pointers to implement dynamic dispatch. Virtual table pointers tie an object to its underlying type and thus determine the correct target for the dispatch. For dispatch, *Object Type Integrity* (OTI) requires that the correct object type is used (the one assigned by the constructor when the object was allocated). Dynamic dispatch leverages the virtual table pointer to identify the type of an object, OTI therefore requires integrity of the virtual table pointer.

Violations of OTI are possible because virtual table pointers are fundamentally required to be in writeable memory. Each polymorphic object (i.e., an object with virtual methods) needs a virtual table pointer, and can exist anywhere in memory. Consequently, the virtual table pointer has to be included in the memory representation of the object. As C++ has neither memory nor type safety, having virtual table pointers in the objects exposes them to corruption. Once virtual table pointers have been corrupted — and OTI violated — the attacker has successfully corrupted control flow information. The attacker then forces that object to be used for dynamic dispatch, hijacking the control flow of the program. Given control over the program's control flow, she can mount a code reuse attack, leading to arbitrary execution.

Observe that code reuse attacks operate in three stages: (i) an initial memory or type safety violation, (ii) corruption of control data, and (iii) the control-flow hijack [34]. The current state-of-the art for efficiently mitigating code-reuse attacks is Control-Flow Integrity (CFI) [3], [29], [37], [9], which mitigates the third step of the attack by limiting the control flow to paths that are valid in the program's control flow graph. CFI mechanisms specialized to deal with C++ dynamic dispatch [37], [9] leverage the class hierarchy when computing the set of valid targets for virtual dispatch sites, increasing precision. Despite the precision added by class hierarchy information, the target set can be large enough for the attacker

to find sufficient gadgets to achieve her desired goals [11], [18], [10], [33]. Each overridden virtual function increases imprecision, as all implementations must be in the same target set. For large C++ applications, such as web browsers, these sets are surprisingly large in practice. See Section III-C for details.

CFI's weakness — over-approximate target sets — is not fundamental to the problem of preventing dynamic dispatch from being used for code-reuse attacks. At runtime there is only one correct target for any virtual call. The correct target is dictated by the *dynamic* type of the object used to make the virtual call. The alias analysis problem prevents static analysis from determining the dynamic type of the object at the virtual call site. Indeed, any case where static analysis can determine the exact type, as opposed to a set of types, should be devirtualized as an optimization. Consequently, a security mechanism must leverage runtime information to correctly track an object's type and secure dynamic dispatch on that object, as opposed to the static, compile time information that CFI policies rely on.

We propose a new defense policy, Object Type Integrity (OTI) which guarantees that an object's type cannot be modified by an adversary, i.e., integrity protecting it, thereby guaranteeing the correctness of dynamic dispatch on that object. OTI thus mitigates the second step of a code-reuse attack by preventing key application control flow data from being corrupted. OTI tracks the assigned type for every object at runtime. Consequently, when the object's type is used for a dynamic dispatch, OTI can verify that the type is uncorrupted. Further, OTI requires that each object has a known type, thus preventing the attacker from injecting objects [33], and using them for dynamic dispatch. OTI distinguishes itself from CFI by intervening one stage earlier in the attack, and by being fully precise, instead of relying on target sets. The two can be deployed together, as discussed in Section III-E, achieving even greater security, as they mitigate different stages of code-reuse attacks that utilize dynamic dispatch.

We present CFIXX, a C++ defense mechanism that ensures the integrity of virtual table pointers, thereby enforcing OTI. CFIXX is a practical, deployable defense that removes an entire class of control-flow hijacking targets. On Chromium, it has negligible overhead on JavaScript benchmarks (2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream), unnoticeable to users. For this low cost, it guarantees that every object used for dynamic dispatch and casts has the correct type, removing a significant attack surface for C++ applications. On the CPU bound SPEC CPU2006 benchmarks, CFIXX has 4.98% overhead. This is slightly higher than the best CFI mechanisms, which is to be expected since CFIXX intervenes at runtime, and provides greater security. CFIXX has been used to recompile libc++, and so protects all of user space, leaving no vulnerabilities due to unprotected code. Further, CFIXX shows how to efficiently use the new Intel Memory Protection Extensions (MPX) to integrity protect arbitrary regions of memory, and applies this technique to virtual table pointers.

To support further development and replication of our results, our prototype OTI enforcement mechanism, CFIXX, is open source: https://github.com/HexHive/CFIXX.

Our contributions are:

1) A new defense policy, Object Type Integrity (OTI), which mitigates all known attacks on dynamic dispatch.
2) A defense mechanism that enforces OTI, CFIXX, and an evaluation of the prototype on SPEC CPU2006 — including libc++, and Chromium.
3) A demonstration of how to efficiently use MPX to integrity protect arbitrary regions of memory.

II. C++ Dynamic Dispatch

Dynamic dispatch is a key part of polymorphism in C++, allowing classes to *override* implementations of *virtual* functions that they inherit. Figure 1 contains a simple code example illustrating virtual functions, overriding implementations, and the associated memory layout. C++ implements dynamic dispatch by maintaining a mapping from each object to its underlying type, and thus the true implementation of the object's virtual functions. At each virtual call site, two things occur: (i) the appropriate function is determined from the object's virtual table, and (ii) an indirect call to that function is made. The correctness of dynamic dispatch thus depends on the integrity of the mapping from an object to its underlying type, i.e., Object Type Integrity.

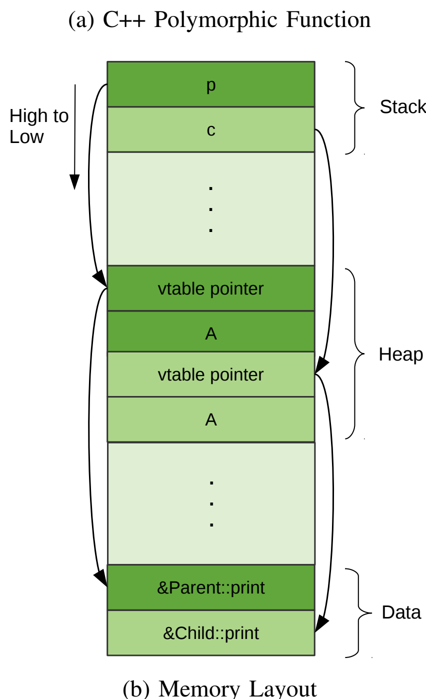
The object type mapping at the core of dynamic dispatch is implemented by creating a *virtual table* for each polymorphic class. The virtual table consists of function pointers to the correct implementation of each of the class's virtual functions. The compiler populates the virtual tables with the correct function pointers, and is responsible for managing the virtual function name to virtual table index mapping. The virtual tables are fixed at compile time, and mapped read only at runtime. Each object of a class is given a *virtual table pointer* which points to the virtual table for the class. Consequently, the virtual table assigned to an object can be thought of as encoding the correct dynamic class of the object. An assignment to the virtual table pointer is added by the compiler in each of the class's constructors.

Virtual function calls are compiled to look ups in the object's virtual table. The virtual table is located by using the virtual table pointer, and the function pointer is retrieved (recall that the compiler manages the virtual function name to index mapping). This function pointer is then called, completing the dispatch.

Figure 1 illustrates how this process works. Figure 1a shows a small C++ program with two classes, `Parent` and `Child`. Both implement a virtual function, `print()`, which prints out the class name. The memory layout for an object of each class is demonstrated in Figure 1b. Note that the virtual table pointer is the first field in the object, as required by the ABI, and is initialized by the object's constructor. To see how the virtual table pointer and virtual tables are used for dynamic dispatch, consider the virtual call at line 20 in Figure 1a. The `print()` function is the only virtual function, and so is at index zero in the virtual table. To find the correct function pointer, indirection through the virtual table pointer is used. This compiles to: `p->vtp[0]()` for `print()`. Regardless of whether `p` is a `Parent` (line 27) or `Child` (line 28) object, this lookup finds the correct implementation of `print()`.

```
#include <iostream>

class Parent {
  public:
    int A = 0;
    virtual void print(void) {
      cout << "Parent" << endl;
    }
};

class Child : public Parent {
  public:
    void print(void) override {
      cout << "Child" << endl;
    }
};

void virtualDispatch(Parent *p)
{
  p->print();
}

int main(void){
  Parent *p = new Parent();
  Child *c = new Child();

  virtualDispatch(p);
  virtualDispatch(c);
}
```

(a) C++ Polymorphic Function



(b) Memory Layout

Fig. 1: Dynamic Dispatch Illustrated.

The description of dynamic dispatch above holds for single inheritance. C++ allows multiple inheritance, which complicates the implementation of dynamic dispatch, however the same concepts hold. Multiple inheritance can result in objects having multiple virtual tables. CFIXX handles multiple inheritance, see Section IV-B for details.

A. *Constructors and Destructors*

In C++, when a new object is created, a special method known as a *Constructor* is called. This method initializes the object, including assigning its virtual table pointer, or pointers in the case of multiple inheritance. When a class is part of an inheritance chain, the constructor for the base class is called first, and then the constructor for its descendant, this continues recursively until the class of the object is reached. These objects are all created at the same memory location, with the result that the virtual table pointer is overwritten for each class in the inheritance chain (for use by the intermediate class's constructor). Analogously, *Destructors* are called when an object is destroyed. The only difference is that the inheritance chain is traversed in the opposite order (from the object's destructor to the base class's destructor). Multiple inheritance complicates the class traversal for constructors and destructors, the details are again ABI specific and not needed here.

III. Object Type Integrity

Object Type Integrity (OTI) is a new security policy which focuses on C++ objects and, at its core, prevents an attacker from changing an object's type, or creating new, synthetic, object types. By protecting objects' types, OTI prevents attackers from hijacking the application's control flow during dynamic dispatch. OTI and Control-Flow Integrity (CFI) thus share a goal — mitigating control-flow hijacking and thus code-reuse attacks. However, OTI's approach is orthogonal, and complementary to, CFI policies as it intervenes earlier in the attack, integrity protecting object types instead of limiting control flow transfers. Additionally, by guaranteeing an object's type, OTI can also protect dynamic casts, which CFI is unable to do. To understand the security added by OTI, we provide background on code-reuse attacks in C++, and define both an attacker model and what is meant by control and data attacks. With these definitions, we discuss the limitations of CFI as a policy in securing forward control flow transfers (e.g. virtual calls), and how OTI mitigates these vulnerabilities. In particular, OTI restricts each forward control flow transfer to one target at runtime, instead of a set of targets. As OTI and CFI are orthogonal, they can be deployed together. We show that using both OTI and CFI together provides significantly stronger security than using either in isolation.

A. *Code-Reuse Attacks*

Code-reuse attacks are designed to bypass deployed defenses: Data Execution Prevention (DEP) [12], stack canaries [13], and Address Space Layout Randomization (ASLR) [35], [7]. DEP prevents code injection, so code-reuse attacks instead leverage existing executable code. Randomization techniques such as ASLR are bypassed by leaking information or using side-channels [17], [21]. Stack canaries [13] prevent sequential overwrites of return addresses, but can be bypassed by corrupting a pointer, and then using the corrupted pointer to write to arbitrary locations.

Code-reuse attacks operate in three steps: (i) an initial memory corruption, (ii) corrupting control data, and (iii) using the corrupted value to hijack a program's control flow. As C++ enforces neither memory nor type safety, any such violation can serve to perform the initial corruption. UaF vulnerabilities

in C++ programs are increasingly popular with attackers, and are currently one of the most common form of memory safety violation for C++ applications [14]. Type safety violations can be used to attack a program's control flow through type confusion attacks, e.g., CVE-2017-2095, CVE-2017-2415, CVE-2017-5023, and Counterfeit Object Oriented Programming (COOP) [33]. Type confusion attacks cast an object to an illegal type where the underlying object is often of different size. This allows the adversary to access unintended memory and, e.g., overwrite virtual table pointers. This is typically done through an attacker forced illegal downcast (e.g., to a sibling class). A more advanced form of type safety violation are COOP attacks. COOP creates (interleaved) objects that use attacker chosen (including synthetic) virtual tables, and thus have invalid types. COOP uses these invalidly typed objects to achieve arbitrary execution. Both type confusion attacks and COOP violate OTI by assigning an object a different type after allocation, or creating a synthetic type.

C++ supports dynamic casts, which execute a runtime check to test if the cast object is compatible with the target type. Dynamic casts leverage C++'s Run Time Type Information (RTTI) to verify that a cast between two classes is valid at runtime. RTTI is associated with virtual tables in C++. Consequently, dynamic casts rely on objects having the correct virtual table pointer to access valid RTTI and correctly verify the type casts. Type confusion attacks either use an object of a different type at a static cast site or an object with a modified vtable pointer at a dynamic cast site. In both cases a cast that should fail completes and an object of the wrong type is used, enabling the type confusion attack.

B. Attacker Model

We assume a strong attacker with arbitrary read and write capabilities. The attacker cannot modify or inject code due to DEP [12]. Her arbitrary write capability can be used to perform control or data attacks. Control attacks are a subset of data attacks, where the data being overwritten directly determines control flow, such as virtual table pointers. A gray area exists between control and data attacks where data (such as `this` pointers) that points to control flow data is modified. Attacks against data that points to control flow data (at any level of indirection) are in scope. Other data attacks are out of scope.

The attacker's goal is a code-reuse attack, which requires hijacking control flow. To hijack control flow, the attacker must corrupt control data, or change the control data that is used. There are two fundamental types of code pointers that are writeable at run time — and thus vulnerable to corruption — in C++ programs: (i) return addresses and (ii) virtual table pointersṄote that backwards control flow transfers (e.g., returns) can be secured by shadows stacks or CET. These defenses can be deployed together with OTI and CFI. Consequently, we only consider attacks which target forward edges, meaning dynamic dispatch through virtual calls for C++ objects. There are four avenues of attack on C++ virtual calls: modifying an existing virtual table, injecting a new virtual table, substituting an existing virtual table, or creating a fake object. Current compilers map virtual tables to read only memory, preventing attackers from modifying existing virtual tables. Injecting a new virtual table is possible, as is substituting an existing virtual table. The final step of either attack is to overwrite the virtual table pointer of an existing object with a pointer to the attacker chosen virtual table. COOP [33] relies on creating "fake" objects, which do not have constructors in the program's source, and so have no official type — virtual table pointer — assigned. All three attacks center on corrupting an object's type via its virtual table pointer, and are included in our attacker model.

A more subtle form of attack involves substituting one valid object for another at a virtual call site. This is accomplished by, e.g., overwriting object pointers in memory. Such attacks that use indirection (by modifying a pointer to a code pointer, instead of the code pointer directly) are also considered in our model.

C. Limitations of CFI - Web Browser Case Study

The CFI policy operates in two phases: analysis and enforcement. At compile time, the analysis phase determines the allowed set of targets for each virtual call site. Since this set is determined *statically*, the dynamic type of the underlying object is unknown. Consequently, the set must include, at a minimum, the implementation of the virtual function in the static class of the object, and any implementation in a descendant class that overrides it. Each such override provides a target that an attacker can divert control flow to without violating the security policy. More coarse grained defenses that do not leverage C++ class hierarchies rely on matching function prototypes (with various degrees of precision), resulting in even larger target sets. Function prototype analysis is also used for C style function pointers. In principle, these sets could be further limited by control and/or data flow analysis. The enforcement component performs a set check at runtime, to verify that the target of the virtual call is in the allowed set. Consequently, even though the check is performed dynamically, it is relying on *static* information — the compile time target sets — for its security properties.

To gain a sense of just how susceptible browsers are to attacks within the minimum allowed target sets for CFI, we quantify the size of minimum target set for virtual calls in Chromium and Firefox. To do so, for each virtual function we count the number of overriding methods. With this information, and the static class of the object used, we can determine the size of the target set at each virtual call site.

In Chromium, we found a total of 13,834 virtual functions, nearly half of which (6,671) have more than one implementation. Out of the 4,679 virtual functions in Firefox, 1,867 (40%) have more than one implementation. 5,828 of the virtual calls in Chromium have more than one call site (including `blink::ExceptionState::rethrowV8Exception` which has 1,559 call sites), and 2,188 virtual functions in Firefox have more than one call site. Taking this data in its totality, it is clear that modern browsers have a large remaining exploitable surface that must be protected even if CFI is deployed. Table I shows the number of virtual functions in Chrome and Firefox with a given number of implementations. The "Max" row is the maximum number of implementations for any virtual function. Each of these virtual functions can potentially be abused without violating the CFI policy, highlighting the severity of the problem.

Impl Count	Chromium		Firefox	
2	3,260	(23.57%)	1,065	(22.76%)
3	1,556	(11.25%)	400	(8.55%)
4	723	(5.23%)	143	(3.06%)
5	705	(5.10%)	88	(1.88%)
$[6 - 9]$	349	(2.52%)	124	(2.65%)
$[10 - 19]$	63	(0.46%)	45	(0.96%)
≥ 20	15	(0.11%)	2	(0.04%)
Max	78		107	

TABLE I: Number and percent of virtual functions with multiple implementations. Max is the maximum number of implementations for any virtual function.

The size of the target sets represents a genuine security risk in light of the prevalence of virtual calls. We instrumented Chromium to count the prevalence of indirect calls through function pointers and virtual calls. Under a normal browsing workload, 42% of indirect calls calls were virtual. This is in line with the results reported by VTrust [37], which found 41% of indirect calls were virtual for Firefox. Of these indirect but non virtual calls, over 50% are due to Chrome's memory management wrapper. Chrome's wrapper intercepts, e.g., `malloc`, `operator new`, dispatching them to its chosen allocator, tcmalloc for Linux[1].

D. OTI Policy

Object Type Integrity guarantees that each C++ object has the correct type, e.g., virtual table pointer, and further that this type was assigned by a valid constructor, thereby preventing "fake" objects. Guaranteeing that every object has the correct type in turn guarantees the correctness of C++ dynamic dispatch and cast on the object (though not necessarily that the correct object is used, see Section III-E). Every virtual call or dynamic cast site has only one *correct* object type at runtime, determined by the dynamic type of the object, determined through the virtual table associated with the object.

OTI enforces that the correct virtual pointer be used for each object during dynamic dispatch and cast, thereby ensuring that the correct virtual table — and thus object type — is used. Further, in order to establish the correct type for each object, OTI requires that every object's type be assigned by compiler generated code in the object's constructor. OTI can be thought of as a fully context and flow sensitive analysis, that takes advantage of runtime information to perfectly resolve the correct virtual table for every object.

OTI mitigates all classes of attacks that can be used to subvert forward control flow transfers for any given object, and all type confusion attacks that target dynamic casts. UaF attacks can be mitigated by invalidating an object's virtual table pointer when its memory is deallocated. This provides only partial safety, as reallocating the memory can assign a new, legitimate virtual table pointer, but it does prevent dynamic dispatch on a free'd object before the memory is re-allocated. Type confusion attacks, or other memory corruptions, that attack control flow by overwriting function / virtual table pointers, are naturally stopped by OTI. Attacks that manufacture "fake" objects, such as Counterfeit Object Oriented Programming, are prevented. These "fake" objects do not have recognized types under OTI. Consequently, our check at the call site fails due to lack of type information for the object, preventing the attack.

Guaranteeing OTI, i.e., the integrity of virtual table pointers, can be used to enforce different security policies. OTI always protects the integrity of virtual table pointers, removing adversary access to them. The OTI policy requires that any mechanism that implements it dispatch on the protected virtual table pointers. By requiring dispatch on protected pointers, the OTI policy already stops control-flow hijack attacks. Implementations can extend the OTI policy by using the virtual table pointer in attacker controllable memory as a canary. Such a policy extension would compare the canary to the protected virtual table pointer. If the canary and the protected pointer are different, a policy violation is reported.

E. Combined Security of OTI and CFI

OTI distinguishes itself from CFI by utilizing *runtime* information to fully protect dynamic dispatch from direct overwrites of virtual table pointers. Such overwrites correspond to control attacks in our attacker model. Recall, however, that data pointers can be used indirectly to change control flow by changing, e.g., which object gets used for dynamic dispatch. OTI on its own cannot mitigate such attacks, just as CFI on its own cannot fully mitigate direct attacks on control data. However, CFI can partially mitigate such indirect attacks on control data because it creates an absolute set of valid targets, thereby limiting the valid object substitutions that can be made. Consequently, OTI is best used complementary with CFI where, fully stopping attacks against code pointers earlier than CFI, and orthogonally leveraging (the potentially imprecise) CFI target sets for any remaining attacks that modify data pointers.

IV. DESIGN

CFIXX enforces OTI by identifying legitimate (compiler inserted) virtual table pointer assignments. These are recorded as the ground truth type of the objects. CFIXX then protects the integrity of these assignments, ensuring that an attacker cannot modify them. The integrity protection is accomplished by recording the original virtual table pointer assignment in a protected metadata table. Dynamic dispatch is then modified to lookup the virtual table pointer for the object in our metadata table instead of in the object itself. Dynamic casts could be protected in the same way, but are not part of the current prototype. CFIXX requires recompilation of the entire program, including libraries (though a compatibility mode with weaker security guarantees is discussed in Section VI). By protecting all of user-space, CFIXX is guaranteed to see the ground truth type for every legitimate object. Consequently, dynamic dispatch on objects without metadata is detected, and the OTI requirement that every object's type be assigned by compiler generated code in a constructor is enforced.

Enforcing OTI, and guaranteeing the integrity of dynamic dispatch per object only requires protecting a small subset of a program's data. Dynamic dispatch only occurs for virtual methods of polymorphic objects, meaning CFIXX only needs to protect the virtual tables and virtual table pointers of these

[1] See https://chromium.googlesource.com/chromium/src/base/+/master/allocator/ for the implementation of Chromium's memory management wrapper.

objects. Virtual tables are mapped to read only memory by current compilers. Consequently, the integrity of the virtual table's themselves is already guaranteed. This leaves only the virtual table pointers in polymorphic objects that need to be integrity protected by CFIXX.

To see how OTI secures dynamic dispatch, consider dynamic dispatch from an attacker's perspective. She wants to change the target of a virtual call, and has an arbitrary read and write per our attacker model. She cannot modify a virtual table, because they are mapped read only. Consequently, she must either change a virtual table pointer in an existing object, or create a "fake" object with a virtual table pointer of her choice. OTI prevents her from executing either attack. She cannot change an existing virtual table pointer, because that would change the object's type, violating OTI. Any object she injects will not have a ground truth type, and thus also violates OTI. A subtle attacker might change the object used at a virtual call site through data-only attacks (i.e., changing object pointers to alternate objects). The effects of such data-only attacks can be limited through CFI.

A. Integrity for Virtual Table Pointers

Guaranteeing OTI is accomplished by protecting the integrity of virtual table pointers, which fundamentally requires allowing only *legitimate* writes to virtual table pointers. Determining what is and is not a legitimate write is a difficult problem in general due to aliasing. However, the problem is simplified for virtual table pointers because only constructors can *legitimately* write virtual table pointers in C++. Any other write is invalid. OTI is guaranteed if a legitimate write of a virtual table pointer is the reaching definition at a virtual call site. Full memory safety would accomplish this, however it is a strictly stronger property than is required. Instead, CFIXX guarantees that only legitimately written virtual table pointers can be used for dynamic dispatch.

CFIXX guarantees virtual table pointer integrity through a two part enforcement mechanism which utilizes a metadata table: (i) recording the correct virtual table pointer in the metadata table, and (ii) using our recorded virtual table pointer for dynamic dispatch on the object. The metadata table is conceptually a key-value store. The object, identified by the `this` pointer serves as the key. The value stored is the correct virtual table pointer for the object. For (i), we note that the initial virtual table pointer assignment to the object cannot be tampered with under DEP as it is in compiler emitted code, and the virtual table pointer is a fixed constant. Consequently, we record the virtual table pointer assigned to the object in our metadata table during this initial assignment. Part (ii) is achieved by looking up the correct virtual table pointer for the object in our metadata table during dynamic dispatch, instead of using the (possibly attacker corrupted) virtual table pointer in the object.

By default, our protection mechanism requires that every object used for a virtual call has an entry in the metadata table. If there is no metadata entry, an attack is detected and execution is terminated. Only objects whose constructors CFIXX did not compile do not have metadata table entries. CFIXX can miss constructors for three reasons: deliberately unprotected code module (e.g., third party shared library), an object created outside of C++ semantics (see Paragraph VI-0c), or an attacker injected object (such as "fake" objects created by COOP). Differentiating between an object from an unprotected module and an attacker injected object is a key challenge for maintaining soundness while still supporting non-protected libraries.

A "compatibility mode" that addresses the challenges of executing with third-party libraries is described in Section VI. CFIXX supports separate compilation and shared libraries. Consequently, there is no technical reason to require a compatibility mode. However, its availability does make it possible for CFIXX to be incrementally deployed, hopefully encouraging its adoption. The rest of this paper assumes that all code is protected by CFIXX.

B. Multiple Inheritance

C++ supports multiple inheritance by allowing objects to have multiple virtual tables. This complicates our metadata scheme, as we can no longer use the `this` pointer (the pointer to the base of the object) as our metadata key, as an object can now require multiple metadata entries. Instead, we use the location of each virtual table pointer in the object as the key. Using the virtual table pointer's address as the key allows multiple values per object in the metadata table, thereby supporting multiple virtual tables and multiple inheritance. CFIXX is ABI agnostic, and relies on the compiler to determine which virtual table for a particular object should be used at a given call-site. Once the compiler has made this decision, CFIXX uses the integrity protected virtual table pointer corresponding to the location in the object.

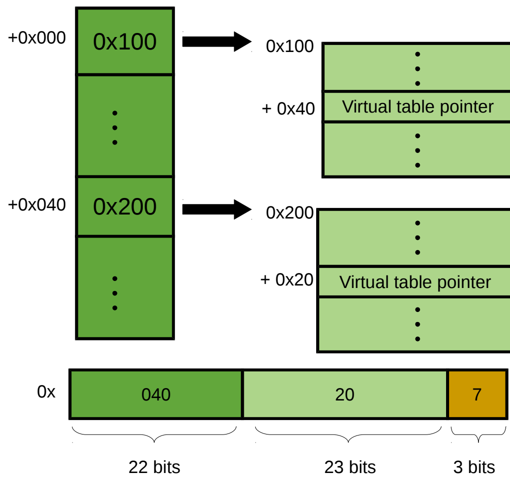
V. Implementation

CFIXX is implemented on top of LLVM 3.9.1, and has three components. The first part is creating, and protecting, the data structure for our metadata. Secondly, we have to create metadata entries each time the compiler assigns a virtual table to a polymorphic object. These compiler generated assignments are the set of all *valid* writes to an object's virtual table pointer. Finally, we alter the dynamic dispatch mechanism to leverage our recorded virtual table pointer for an object, instead of the virtual table pointer contained in the object. The current prototype of CFIXX supports 64-bit x86_64 systems with the Itanium ABI.

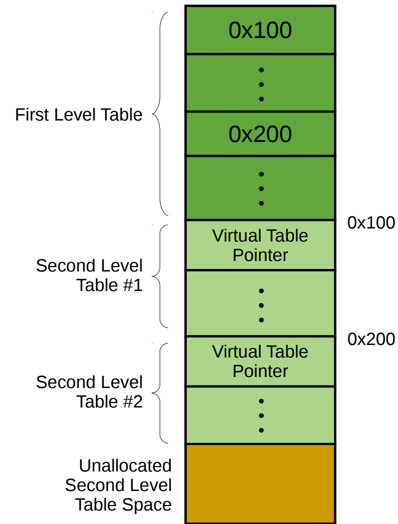
A. Metadata Data Structure

Our metadata is stored in a two level lookup table, see Figure 2a. To find the correct entry, the pointer is split into two parts. The high order bits are the index in the top level table, `040` in the figure. Entries in the top level table are pointers to the second level tables. The low order bits, `20` in the example, are the index in the second level table, which stores the correct virtual table pointer for the object.

Our decision to use a two level lookup table is based on the allocation patterns of polymorphic objects. We found relatively little entropy in the high order bits of object addresses, and insufficient entropy in the middle bits to justify a three level table (see Section VII-B). Consequently, we use a two level table, which requires one less indirection and so has better performance without increasing the size of our metadata table.



(a) Logical Structure of Metadata



(b) CFIXX Metadata Memory Layout

Fig. 2: Metadata Design Illustrated. Figure 2a shows how a pointer is broken into two indexes into the top level and second level table. The entry for the top level table points to the second level table. Figure 2b shows how all the tables are laid out in the protected data region. The top level table is allocated first, then all second level tables are allocated on demand.

While SPEC CPU2006 was used for our design study, the results reflect that objects are either heap or stack allocated and tend to be grouped in memory. The same metadata table structure was used for Chromium, and is not "tuned" for SPEC CPU2006. x86_64 uses 48 bits for virtual addresses. Of these, we chose to use the high order 22 bits for the top level table, and the next 23 bits for the second level tables. All polymorphic objects are at least eight bytes large (for the virtual table pointer), so we can disregard the low order 3 bits.

Our metadata tables are memory mapped, and large enough for the top level table, and a fixed number of second level tables, as shown in Figure 2b. Note that mapped pages are not touched (and thus actually allocated by the OS) until they are needed, so we only pay the memory overhead for touched areas of instantiated tables. Contiguously allocating the metadata tables allows us to only integrity protect one memory region. The location of this region can be determined either at runtime, allowing maximum flexibility for integration with randomization defenses such as ASLR, or at a fixed location, a performance optimization that removes an indirection on each metadata access to lookup the table's location.

B. Metadata Protection

Our attacker model assumes a strong attacker who can perform arbitrary reads and writes. Consequently, we must protect our metadata from modification by an adversary. The lowest overhead option is information hiding through randomization. However, this approach is comparatively easy to defeat [17], [21]. Further, it has a performance cost because it requires an additional redirection at runtime to find the metadata table. This leaves three general approaches: (i) masking, (ii) using MPX to prevent unauthorized writes to our metadata, or (iii) leveraging Intel's forthcoming Memory Protection Keys (PKEYS) to make the metadata read only except when we need to write it (doing this via `mprotect` is prohibitively expensive). Masking is well known from software-based fault isolation, and simply masks each pointer before it is dereferenced, e.g., through an `and` instruction.

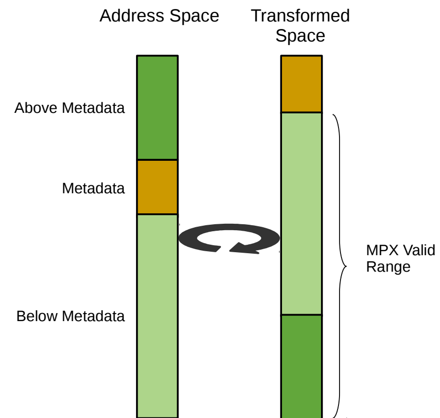


Fig. 3: Address Space Rotation for MPX checks

The MPX hardware extension is designed to provide hardware support for verifying that a pointer is in bounds for memory object accesses. To do so, MPX introduces four new 128 bit registers for storing upper and lower bounds (each register stores two 64 bit pointers — one for the upper bound and one for the lower bound). Additional instructions are provided to make bounds, load and store to the bounds registers, and perform upper and lower bounds checks.

We leverage MPX bounds checks to prevent unauthorized writes to our metadata, by dividing memory writes into two categories: those added by CFIXX to maintain the metadata,

and all other writes. The first category does not require bounds checks. For an attacker to use such writes maliciously, she would have to have already broken CFIXX, or hijacked control through an orthogonal attack. CFIXX adds MPX bounds checks to the second category — all writes that do not touch metadata. Note that reads do not need be checked as CFIXX only requires integrity. There is no need to keep the virtual table pointers confidential. MPX checks ensure that a memory address is within a given range. There is no way to check that an address is not in a given range, or that either the upper or lower bounds check passes, but not both. Consequently, one set of bounds is needed for all non-metadata writes, e.g, all legal memory for the writes needs to be treated as a single, contiguous array. However, our metadata table is mapped somewhere in the middle of the address space, leaving a valid region both below and above our table.

CFIXX uses a novel technique to adapt MPX to perform the required check, illustrated in Figure 3. Our exclusive checks are performed on a rotated version of the address space. In our rotation, the metadata table is at the top of the address space. This allows us to treat everything below it as the valid range. In the rotated address space, the last valid byte of our metadata table is at $2^{64} - 1$. Consequently, the rotation can be accomplished by adding the appropriate offset to a given address. Addresses that are above the metadata table in the normal address space naturally wrap around to the bottom due to arithmetic overflow. The translation is implemented as a `lea, mov, lea`. The first `lea` calculates the address that is being written to, the `mov` loads the offset, and the second `lea` applies it. Note that `lea` is side effect free and does not effect the flag register. CFIXX uses one MPX bounds register, with the bounds set to `0x0` and the rotated base of our metadata table. Consequently, after rotation CFIXX performs an MPX upper bound check `bndcu`. This works because the base of our metadata table is the upper limit of valid addresses in the translated address space.

Our rotation offers two key advantages: (i) the protected region can be anywhere in the address space — unlike for masking, and (ii) it requires half as many bounds checks as a naïve MPX implementation. Masking requires the metadata table to be at the top of the address space. To see why, consider what happens when high order, non-masked bits are changed. For each combination of such bits, a hole is created by the mask for the protected, lower order bits in the address. Consequently, many disparate regions of memory would be protected. Applying MPX to integrity protect a memory region without our rotation is possible. However, MPX only provides inclusive checks, i.e., that an address is in bounds. To provide integrity, the opposite is desired, i.e., that a write is *not* in the protected region. This is not easily accomplished, as it requires checking if the address is in either the area below the protected region, and if that check fails, if it is in the area above the protected area. A violation must only be signalled if both checks fail. MPX does not support this gracefully. However, our rotation provides one valid region for any legal write, which naturally fits the MPX paradigm, and only requires one bounds check.

Intel has announced, but not yet shipped in production, a new hardware feature called Memory Protection Keys (PKEYS). These will allow a process to switch a page (or group of pages) between RW and R with one register write. This feature can naturally be used to protect our metadata. The pages would be set R anytime CFIXX is not performing a write to them. Since CFIXX is implemented in the compiler, it can ensure that only authorized locations use this feature. To maliciously use these locations, an attacker would have to already have diverted the program's control flow. As for MPX, this can only happen if CFIXX has already been bypassed — rendering it irrelevant, or the attacker is utilizing an orthogonal attack. Consequently, PKEYS can be used to protect the metadata. We anticipate that this will have significantly less overhead. Only metadata writes would be protected, instead of all writes. Further, the protection would only require two additional instructions per metadata write — to toggle the metadata to RW and back to R, instead of four instructions per data write (three to rotate the address space, and one to perform the bounds check). Metadata writes are dwarfed by the number of data writes, significantly reducing the number of checks required by the PKEYS protection scheme.

C. *Runtime Instrumentation*

With this metadata scheme in mind, the next questions are how and when metadata is created and referenced. Recall from Section IV that there is only one valid write of an object's virtual table pointer. This is when we want to create metadata for the object. As described in Section II, virtual table pointers are used during dynamic dispatch for virtual calls. CFIXX instruments the virtual calls to use the virtual table pointer from our metadata, as is illustrated with C pseudo code in Figure 4. Figure 4a shows the C equivalent of the original code generated by the compiler for an object constructor, and then virtual code. Figure 4b shows the same code after CFIXX's instrumentation.

Virtual tables are assigned to objects by the object's constructor. The compiler implicitly adds the virtual table pointer field and writes the correct value to it in the constructor. This holds for all constructors in C++11 (and the relevant subset for older versions of C++): default, programmer specified, copy, move. Clang-3.9.1 has a common sub-routine (`CodeGenFunction::InitializeVTablePointer`) that initializes the virtual table pointer for all constructors. CFIXX modifies this sub-routine to also emit instructions to create metadata for the object. The virtual table pointer is still written into the object as well for backward compatibility with unprotected code and to make it possible for CFIXX to prevent *and* detect attacks.

CFIXX's current prototype supports virtual dispatch via the Itanium ABI. In clang-3.9.1, the Itanium ABI relies on `CodeGenFunction::GetVTablePtrCXX` to retrieve the virtual table pointer that is then used for dynamic dispatch. CFIXX reimplements this function to read the virtual table pointer from our metadata instead of the object. As the virtual tables are unchanged, no further changes to dynamic dispatch are required for it to succeed. The detection and logging modes mentioned in Section IV are simple to implement, though not evaluated here. Implementing them requires adding a comparison of the virtual table pointer in the object to the virtual table pointer retrieved from memory. If they are different, an attack would be detected, or a violation logged. This extra `if` check, and potential policy action, would add

```
    // Constructor
    Foo::Foo(){
      this.vtablePointer = 0x200;
    }
    // Virtual Call
    bar(Foo *f){
      f->vtablePointer[1]();
    }

```

(a) Before CFIXX

```
    // Constructor
    Foo::Foo(){
      this.vtablePointer = 0x200;
      secondLevel = metadata[this>>26];
      if (secondLevel == NULL)
        secondLevel = allocate2ndlvl(this);
      secondLevel[(this>>3)&((1UL<<23)-1UL)] =
        this.vtablePointer;
    }
    // Virtual Call
    bar(Foo *f){
      secondLevel = metadata[this>>26];
      vtablePointer = secondLevel[(this>>3)&((1UL<<23)-1UL)];
      vtablePointer[1]();
    }

```

(b) After CFIXX

Fig. 4: Virtual table pointer initialization and dynamic dispatch before and after CFIXX

overhead, and does not affect the security provided by CFIXX, and so was omitted here.

VI. Discussion

Here we elaborate on possible extensions to CFIXX, and highlight some edge cases and low level implementation challenges such as metadata allocation, non-standard objects, leveraging run time type information, compatibility modes, and devirtualization.

a) Metadata Allocation: CFIXX requires its metadata tables to be allocated before any object is allocated and has its constructor called. The most reliable way to enforce early metadata allocation is to add our metadata allocation function to the `.preinit-array` section of the binary. This `ELF` section is supported by the gold linker on both Linux and FreeBSD (as used in our evaluation).

b) Use After Free Detection: The OTI policy can, in principle, provide UaF protection. However, the current CFIXX implementation does not invalidate metadata when objects are deallocated. There are a few challenges to doing so. First, there is no straightforward way to instrument destructors as polymorphic objects can have trivial destructors, which are omitted. Adding destructors would break the C++ ABI. Consequently, we would have to instrument deallocations directly. This means `free` for heap objects, and for stack objects we would have to handle stack frame deallocation explicitly.

c) Non-Standard Objects: It is possible to create an object without calling its constructor. This can be accomplished by copying an existing object with, e.g., `memcpy()`. Such code violates the C++ standard[2], which requires using move or copy constructors, but does often work in practice. We found one instance of this behavior in Chromium, see Section VII-C. Such code should be refactored to use move or copy constructors, as appropriate. CFIXX does not support objects created in this non-standard way, requiring programmer intervention to refactor the code. Note that such behavior is rare, we only found one instance in total, namely in the large Chromium code base and we refactored the code, adding three lines (a call to our metadata create function, and a forward declaration of that function). An alternative solution would refactor the code to use a move constructor.

d) Run Time Type Information: RTTI is used to validate dynamic casts at runtime. To do so, the compiler emits objects that encapsulate an object's type, and can be used to traverse the type hierarchy. The RTTI objects contain virtual pointers and are emitted at compile time. Since RTTI objects are statically created by the compiler, no constructor is ever called for them. To handle these objects, CFIXX adds a new function to the `ELF .ctor` array that creates metadata for every RTTI object.

e) Compatibility Mode: To support non-protected libraries, CFIXX can be extended to provide a compatibility mode. In this mode, the virtual table pointer in the object's memory is used if we do not have a metadata entry for the object. To secure this mode against attacker injected objects, such as created by COOP attacks, this mode is only enabled for virtual call sites where the static type of the object is known to be defined in a third party library. This limits COOP attacks to creating synthetic objects from unprotected classes. As these classes are unprotected, this does not affect the soundness of CFIXX. Note that, if all code is protected through CFIXX then the compatibility mode is disabled, as done for our experiments.

f) Devirtualization: An important optimization for C++ is devirtualization — replacing a virtual call with a direct call. As CFIXX changes the dynamic dispatch mechanism, we investigated the impact on devirtualization. Currently, enabling CFIXX causes about 10% more virtual calls in SPEC CPU2006. We are aware of no fundamental reason why CFIXX should prevent devirtualization and are working on making the optimization passes in the LLVM middle-end aware of CFIXX.

[2]See, e.g., section 12.8 of the C++14 standard which specifies how objects can be copied or moved.

Exploit Type	LLVM CFI	CFIXX	VTrust / VTI†	CPS†
FakeVT	✓	✓	✓	✓
FakeVT-sig	✓	✓	✓	✓
VTxchg	✓	✓	✓	✓
VTxchg-hier	✗	✓	✗	✓
COOP	✗	✓	✗	✗

TABLE II: Exploits caught by vtable protection mechanisms. VTI is Virtual Table Interleaving. See Section VII-A for explanations of exploit types. † inferred from published description.

g) C Style Indirect Function Calls: CFIXX can be adapted to protect C style indirect function calls. There are two challenges: rogue function pointer writes and designing invoke semantics for indirect function calls. First, writes to function pointers are more difficult to detect than writes of virtual table pointers. Additionally, function pointers can be assigned from other variables, leading to the alias analysis problem. Consequently, possible writes of a function pointer would have to be traced throughout the program to correctly maintain our metadata, like SoftBound [27] did for memory objects. Alternately, a different metadata scheme could be used. These difficulties arise because we can no longer rely on the semantics of move and copy constructors. The second challenge would be to refactor indirect function calls. Currently, no lookup is performed. This would have to be changed so that the correct function pointer is looked up in our metadata and then called. Designing an efficient system to address both these challenges is left as future work.

h) Metadata Protection: There are alternatives to protecting the metadata beyond our current MPX implementation, and the forthcoming PKEYS. An alternate MPX implementation is possible, where the metadata table is allocated at the top of user space, eliminating the need to rotate the address space for checks. Allocating the metadata table at the top of user space would require moving the stack down. The principled way to move the stack down would be to change the fixed stack start address (to which ASLR then adds randomness) in the kernel, to guarantee that there would always be room above the stack. Moving the stack via a kernel modification would have no impact on the soundness of CFIXX, and would provide only a performance improvement. Further, it would make CFIXX more difficult to deploy. Consequently, we chose not to implement a kernel modification. Another alternative to using MPX would be storing the metadata in an SGX enclave. However, SGX provides much greater protection than we require. Consequently, we chose not to evaluate it for CFIXX.

VII. EVALUATION

CFIXX is evaluated along two dimensions: security and performance. We show that CFIXX stops more attacks than any existing control-flow hijacking mitigation. On the performance axis, we present our design study of metadata implementation alternatives, and our performance results both on SPEC CPU2006 (4.98%) and on Chromium, using JavaScript benchmarks (2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream). SPEC CPU2006 was run on a machine with a 3.40 GHz Intel Core i7-6700 CPU and 16GB of RAM, under Ubuntu 16.04. Chromium was run on a 4.0 GHz Intel Core i7-6700K with 32GB of RAM, under FreeBSD 11.

A. Exploit Coverage

We created a suite of C++ tests[3] demonstrating various possible scenarios of vtable pointer overwrites. The scenarios we tested for are the following, listed in order of mitigation difficulty:

1) *FakeVT:* Inject a fake virtual table with a pointer to a function that does not share the same signature as the function in the original virtual table.
2) *FakeVT-sig:* Inject a fake virtual table with a pointer to a function that shares the same function signature (same name, number of arguments, types of arguments, and order of arguments) as the function in the original virtual table.
3) *VTxchg:* Overwrite a virtual table pointer with a virtual table pointer of a class outside of the original class hierarchy (i.e., change `Child`'s vtable pointer with the vtable pointer of `Stranger`, and `Child` and `Stranger` are unrelated).
4) *VTxchg-hier:* Overwrite a virtual table pointer with the virtual table pointer of a class within the same class hierarchy (i.e., change `Child1`'s vtable pointer with `Child2`'s and both `Child1` and `Child2` are both direct children of `Parent`).
5) *COOP:* Create a fake object of a class (via, e.g., `malloc`) without calling the class' constructor, and call a virtual function of the fake object. This is the basic COOP attack.

Table II summarizes the results of running the code from our suite with LLVM CFI and CFIXX. We could not evaluate against the state-of-the-art in virtual table protection, VTrust [37] and Virtual Table Interleaving [9], because the source was not yet made available from the authors. Based on our analysis of the papers, we expect that VTrust and Virtual Table Interleaving would detect 1 – 3. They cannot detect 4 and 5 as a valid virtual table pointer is used in the wrong context. The VTrust paper discusses mitigating COOP, but assumes that the control-flow hijack attack violates the class hierarchy constrained set of valid functions. This assumption does not necessarily hold — particularly for web browsers which have large class hierarchies, as shown in Section III-C. The latest Code Pointer Separation [24] code was not able to compile our code suite. However, based on the description in the paper, we expect that CPS mitigates exploit types 1 – 4, but would not protect against exploit type 5.

B. Metadata Design

An efficient implementation of the metadata table is a key component of CFIXX. The metadata table must support an object at every (8 byte aligned) virtual address. At the same time, CFIXX seeks to minimize memory overhead. The classic solution to this situation is a multilevel lookup up table, e.g., as used for virtual memory in the page table. We initially experimented with a three level lookup table. To examine how program's use memory, we logged the address of every polymorphic object created by xalancbmk and omnetpp from the SPEC CPU2006 benchmark suite. We then post processed the log to determine the metadata table that would be built under different lookup schemes.

[3] Source available at https://github.com/HexHive/CFIXX.

Table III shows the results for a three level metadata table under different configurations. Note the extremely low usage rates for all levels. The top level (L1) only had two entries - corresponding to the stack and the heap. The middle level (L2) was also effectively empty. As each level of indirection on metadata reads and writes cost performance, we eliminated the intermediate level.

The remaining design question regards the size of the first and second levels in our two level lookup table. Our experiments on this, shown in Table IV, indicate that using the high order 22 bits is ideal. After 22 bits, density levels off and then declines, increasing the virtual address space footprint of our metadata.

When considering memory overhead for our scheme, it is important to remember that we are predominantly using virtual address space. The vast majority of the underlying pages in any table remains untouched, and so not actually allocated by the kernel. This minimizes our impact on physical memory usage and performance. In the worst case, if every single memory page of the application holds at least one virtual object, the memory of the application doubles. In practice, we measured a 79% increase in memory usage. It should in principle be possible to design a more efficient metadata scheme, as we only need to store eight bytes per polymorphic object.

C. Performance

We evaluated CFIXX on the SPEC CPU2006 compiler benchmarks, as well as on Chromium. The SPEC CPU2006 benchmarks are standard for evaluating compiler based security mechanisms, and included for comparison purposes. Chromium is the open source version of the popular Chrome browser. Web browsers are the most common network facing applications, and so the usability of Chromium with CFIXX is an important indicator of the ability of CFIXX to be deployed in practice. For this evaluation, we used a fixed metadata table location.

Figure 5 shows the performance of the C++ benchmarks in SPEC CPU2006 when compiled with CFIXX. CFIXX has 2.22% overhead without MPX protecting the metadata, and 4.98% with full MPX protection. These averages are geometric means over the full set of benchmarks, as is standard in the literature. LLVM includes a CFI implementation that protects virtual calls, using the same policy as VTrust and VTable Interleaving. As VTrust and VTable Interleaving are not yet open source, we evaluate against the LLVM CFI implementation. This protection is enabled with the `-fsantize=cfi-vcall` flag, and requires LTO. To compare CFIXX against, we reran the baseline with LTO as well to achieve an apples to apples comparison for measuring the performance impact of LLVM-CFI. Despite using LTO and a different baseline, its performance is reported in Figure 5 for comparison. LLVM-CFI fails to run the xalancbmk benchmark, but on the other benchmarks has 0.18% overhead, or an average of 1.25% on benchmarks with virtual calls. This performance is inline with the reported numbers from VTrust and VTable Interleaving, and reinforces the fact that CFIXX achieves almost equivalent performance with much stronger protections.

L1, L2, L3	L1 #(%)		L2 #(%)		L3 #(%)	
16, 16, 13	2	(0.00%)	2,562	(3.91%)	648	(0.99%)
8, 16, 21	2	(0.78%)	14	(0.02%)	123,032	(0.73%)
8, 8, 29	2	(0.78%)	1	(0.39%)	1,660,932	(0.04%)

TABLE III: Three Level Metadata Design Study. Level Sizes are in bits. Sum is 45 — the 48 bits used for virtual addresses less the low order 3 bits (minimum object size is 8 bytes). #(%) is the average number of entries (density of entries). Density is $\frac{entries}{2^{levelsize}}$

L1, L2	L1 #(%)		L2 #(%)	
18, 27	2	(0.00%)	1,660,932	(0.15%)
20, 25	3	(0.00%)	1,107,288	(0.41%)
22, 23	8	(0.00%)	415,233	(0.62%)

TABLE IV: Two Level Metadata Design Study. Level Sizes are in bits. Sum is 45 — the 48 bits used for virtual addresses less the low order 3 bits (minimum object size is 8 bytes). #(%) is the average number of entries (density of entries). Density is $\frac{size}{2^{levelsize}}$

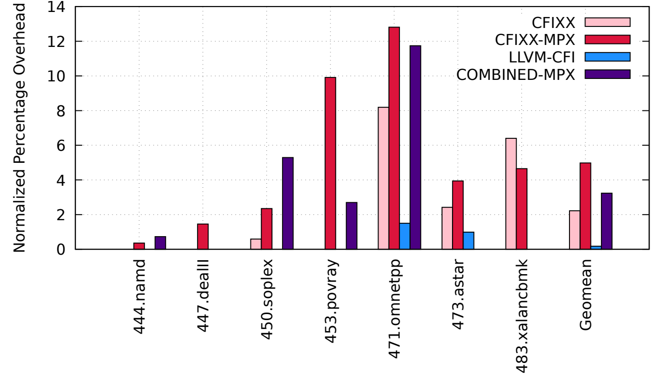


Fig. 5: SPEC CPU2006 Performance Measurements. Results are percentage overhead. LLVM-CFI requires LTO. As a result, the Combined measurement also uses LTO.

To show that OTI and CFI are orthogonal and can work together, we ran SPEC CPU2006 with both LLVM's CFI mechanism and CFIXX enabled. Only one benchmark, dealII, that runs with LLVM-CFI failed under combined protections — without *any* additional engineering effort. We anticipate minimal code changes will be required to make the two mechanisms fully compositional, and ensure that optimizations do not mix their checks. However, running all but one benchmark "out of the box" shows that the two are highly compatible, and that the remaining incompatibility is an engineering issue and not a fundamental design issue. The overhead with both protections enabled is in line with the overhead of only OTI — a benefit of adding the LTO optimizations required by LLVM's CFI mechanism.

To evaluate CFIXX's robustness and impact on real world code, we recompiled the Chromium browser on FreeBSD. FreeBSD was used because its ports system (i) supports clang, and (ii) abstracts out the details of applications' build systems. Consequently, FreeBSD allows for more rapid prototyping of clang-based compilers.

To benchmark Chromium, we ran the Kraken benchmark from Mozilla, the Octane benchmark from Google, and JetStream, which is an independent benchmark. As mentioned in Paragraph VI-0c, Chromium performs a `memcpy` of an allocated object. Running these benchmarks under CFIXX required a manual code change to maintain our metadata in the face of this non-standard code. We added three lines of code (a call to our metadata create function, and a forward declaration of that function) to maintain our metadata. FreeBSD does not yet support MPX, so our evaluation of Chromium is without MPX protection.

Browsers are benchmarked using suites of JavaScript tests. CFIXX has the following overheads: 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream. The slow down is unnoticeable to users. For this low cost, CFIXX provides strong control-flow hijacking guarantees for Chromium. By enforcing OTI, CFIXX guarantees the correctness of dynamic dispatch on a per object basis — something CFI policies cannot do. As shown here, it can easily be combined with CFI to mitigate object swapping attacks. Consequently, we believe that CFIXX should be deployed on all browsers.

VIII. Related Work

Defenses against control-flow hijacking attacks have been studied for many years. Instead of listing all proposed policies, we will focus on control-flow integrity defenses that specifically target virtual dispatch and C++. We refer to a recent survey on CFI defenses for a more complete overview [10].

VTrust [37] presented the current state-of-the-art policy for protecting dynamic dispatch. VTrust protects pointers to virtual tables through a two-layered system. During runtime, it enforces that the virtual function target matches the expected function, as determined statically from source. VTrust ensures that the target function name, argument type list, and class relationship match through the use of hash signatures. The second layer, which is optional and only needed for applications with writable code, encodes legitimate pointers to virtual tables during object creation, and allows only such pointers to be used. VTrust can stop COOP attacks if they cause virtual calls to violate its computed target set, but not all COOP attacks. VTable Interleaving [9] improves the performance of the VTrust policy by rearranging how vtables are organized in memory, efficiently packing the newly organized tables to reduce memory overhead, and reduces virtual call dispatch verification to a simple range check. Neither mechanism is open source, impeding comparative evaluation of performance and security policies. VIP [19] builds on these policies by adding static analysis to reduce the valid target set at virtual call sites.

VTrust, VTable Interleaving, and VIP have less overhead than CFIXX, but provide less security. While VIP leverages extensive analysis of pointers to shrink target sets as much as possible, the authors still report 1,173 call sites in Chrome with more than 64 targets. Determining the object types that can reach a virtual call site is a very difficult problem for static techniques, which are forced to be over-approximate or prevent legitimate execution paths. This over-approximation limits the security that they can provide.

PittyPat [15] performs an online analysis of execution traces in combination with a statically computed control flow graph to determine legitimate targets of indirect control flow transfers. Consequently, PittyPat is fully path sensitive, which makes its static analysis significantly more precise. PittyPat reports that 90%+ of call sites are reduced to one target. However, a subset of call sites with a large number of targets remains. OTI removes this remaining imprecision for C++.

πCFI [30] builds off of MCFI [29] by dynamically activating edges in the control flow graph on demand. This limits the set of targets available to attackers. However, the target set can still grow to be the full statically computed set. Consequently, while πCFI makes attacks more difficult, it is fundamentally no more secure than MCFI.

SafeDispatch [23] provides vtable protections against some control-flow hijacking attacks, through the use of class hierarchy analysis and run-time checks to confirm that the called method is a valid implementation of the called method. SafeDispatch incurs higher overhead (2.1% vs. 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream) and cannot protect against COOP. In addition, SafeDispatch does not support separate compilation. CFIXX supports separate compilation, making its use in large projects practical.

VTint [36] protects against code injection attacks by preventing the use of virtual tables that are writable. In legitimate programs, objects' virtual tables are never changed, and thus any virtual table that is writable should never be used. VTint checks during runtime that any used virtual table is read-only. Recent attack classes, like COOP and vtable reuse, effectively bypass VTint.

vfGuard [31] extracts virtual tables and call sites from COTS binaries to determine a fine-grained CFI. Once all virtual tables and call sites are found, vfGuard then generates a list of targets for each virtual function and enforces this CFI policy through the use of a binary rewriter [26]. T-VIP [20] is another example of binary rewriting that protects against code-reuse attacks in a similar way to VTint. Again, COOP is able to overcome these defenses. Additionally, vfGuard imposes high overhead, making its widespread use unlikely. LLVM provides a virtual call CFI mechanism [1]. However, as shown in Section VII-A, CFIXX blocks more exploits with similar runtime overhead (reported 1% vs. 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream on Chromium). Additionally, LLVM CFI requires Link Time Optimization [2], which greatly increases the resources needed for compilation. CFIXX does not require Link Time Optimization.

An alternate approach to protecting application control flow is Code Pointer Integrity (CPI) and Code Pointer Separation (CPS) [24]. CPI attempts to ensure that pointers to code (return addresses, indirect call sites, etc.) are legitimate through the use of instrumentation and/or runtime checks. CPS is a more relaxed version of CPI, with lower overhead at the expense of some safety guarantees provided by CPI. CPI experienced over 40% performance overhead on some C++ SPEC CPU2006 tests, and over 80% on pybench. CPS had lower overhead, 2% on SPEC CPU2006. The key differences between CPS and CFIXX are that: (i) CFIXX fully protects virtual table pointers, whereas CPS prevents direct overwrites, but not those done through additional indirection, and (ii) CPS

cannot prevent COOP attacks because it cannot detect synthetic objects. Consequently, CFIXX provides greater security for C++ at the same overhead.

Type confusion sanitizers [25], [22] also focus on protecting object types. They detect static cast violations dynamically, and do not protect dynamic dispatch. Sanitizers are orthogonal to defenses as they focus on detecting bugs, not protecting against adversarial attacks.

Full memory safety has been the subject of many years of research [16], [4], [6], [28]. Mechanisms that protect unsafe languages either incur large ($> 100\%$) overhead, or only provide probabilistic protections. If practical, full memory safety would eliminate a class of vulnerabilities used by attackers to corrupt program state, such as object types. For these reasons, whole address space memory protection remains an ongoing area of research.

IX. Conclusion

We have presented Object Type Integrity (OTI). Our new defense policy guarantees that polymorphic objects have the correct type associated with them at run time. By doing so, we protect dynamic dispatch on a per object basis. OTI advances the state of the art in protecting dynamic dispatch by limiting objects to one target per call site, instead of a set of targets like CFI. OTI prevents the second stage of code-reuse attacks — corrupting control flow data — whereas CFI prevents the final stage — control-flow hijacking. Consequently, the two are orthogonal and compatible, and can be deployed together.

Our OTI implementation, CFIXX[4], has minimal overhead on CPU bound applications such as SPEC CPU2006: 4.98%. On key applications like Chromium, CFIXX has negligible overhead on JavaScript benchmarks: 2.03% on Octane, 1.99% on Kraken, and 2.80% on JetStream. CFIXX protects all code in user space (including the libc and any shared libraries), and comprehensively hardens dynamic dispatch. Consequently, CFIXX is compatible with current large C++ applications such as Chromium, readily deployable, and advances the state of the art in protecting dynamic dispatch.

Acknowledgment

We would like to thank the anonymous reviewers and especially our shepherd Brendan Dolan-Gavitt for their insightful feedback that greatly improved the final paper. We thank Abe Clements, Bader AlBassam, James Lembke, and Marion Marschalek for their valuable comments and discussions on this paper. This material is based in part upon work supported by the National Science Foundation under award CNS-1513783, by ONR award N00014-17-1-2513, and by Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

[4]The prototype implementation is open source at https://github.com/HexHive/CFIXX.

References

[1] "Control flow integrity," http://clang.llvm.org/docs/ControlFlowIntegrity.html, 2016.

[2] "Llvm link time optimization," http://llvm.org/docs/LinkTimeOptimization.html, 2017.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[4] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 51–66. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855772

[5] I. Beer, https://googleprojectzero.blogspot.com/2017/04/exception-oriented-exploitation-on-ios.html.

[6] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," *SIGPLAN Not.*, vol. 41, no. 6, pp. 158–168, Jun. 2006. [Online]. Available: http://doi.acm.org/10.1145/1133255.1134000

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits."

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

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

[10] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, 2018.

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 161–176. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831154

[12] M. Corporation, "A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003," https://support.microsoft.com/en-us/kb/875352, 2013.

[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *Usenix Security*, vol. 98, 1998, pp. 63–78.

[14] N. N. V. Database, https://nvd.nist.gov/.

[15] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[16] Duck, Yap, and Cavallaro, "Stack bounds protection with low fat pointers," in *NDSS Symposium 2017*, ser. NDSS 2017, 2017.

[17] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 781–796.

[18] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 901–913.

[19] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for c++," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 329–340.

[20] R. Gawlik and T. Holz, “Towards automated integrity protection of c++ virtual function tables in binary programs,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC ’14. New York, NY, USA: ACM, 2014, pp. 396–405. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664249

[21] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, “Aslr on the line: Practical cache attacks on the mmu,” *NDSS (Feb. 2017)*, 2017.

[22] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, “Typesan: Practical type confusion detection,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 517–528.

[23] D. Jang, Z. Tatlock, and S. Lerner, “Safedispatch: Securing c++ virtual calls from memory corruption attacks.”

[24] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[25] B. Lee, C. Song, T. Kim, and W. Lee, “Type casting verification: Stopping an emerging attack vector,” in *USENIX Security*, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[27] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Softbound: Highly compatible and complete spatial memory safety for c,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’09. New York, NY, USA: ACM, 2009, pp. 245–258. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542504

[28] ——, “Cets: Compiler enforced temporal safety for c,” in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM ’10. New York, NY, USA: ACM, 2010, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1806651.1806657

[29] B. Niu and G. Tan, “Modular control-flow integrity,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 577–587. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594295

[30] ——, “Per-input control-flow integrity,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: ACM, 2015, pp. 914–926. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813644

[31] A. Prakash, X. Hu, and H. Yin, “vfguard: Strict protection for virtual function calls in cots c++ binaries.” 2015.

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

[33] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 745–762.

[34] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 48–62. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.13

[35] P. Team, “Pax address space layout randomization (aslr),” 2003.

[36] C. Zhang, C. Song, K. Chen, Z. Chen, and D. Song, “Vtint: Defending virtual function tables’ integrity,” in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

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