Extensible Virtual Call Integrity

Yuancheng Jiang[1(✉)], Gregory J. Duck[1], Roland H. C. Yap[1], Zhenkai Liang[1], and Pinghai Yuan[2]

[1] National University of Singapore, Singapore, Singapore
{yuancheng,gregory,ryap,liangzk}@comp.nus.edu.sg
[2] Anhui Normal University, Wuhu, China
yph@ahnu.edu.cn

Abstract. Virtual calls in C++ are known to be vulnerable to control-flow attacks, and *Virtual Call Control Flow Integrity* (VCFI) is a proposed defense. However, most existing VCFI defenses are incompatible with real-world C++ software that need extensibility in the form of dynamic loading, foreign language interface, etc. In this paper, we propose a novel and *extensible* VCFI mechanism—namely eVCFI—that is flexible enough to handle such software requirements. eVCFI uses *Approximate Membership Query* (AQM) filters, recasting VCFI as an efficient set membership query, giving an $O(1)$ time VCFI check that can be implemented in only a few instructions, all while supporting extensibility and multi-threading. We compare eVCFI with existing VCFIs, showing that we can achieve more accurate policies or extensibility compared with other VCFI mechanisms designed for efficiency or modularity. Evaluation of eVCFI shows small 1.3% overhead with SPEC 2006. Furthermore, we evaluate eVCFI against the FireFox web browser: an example of large/complex C++ software that uses both dynamic loading and a foreign language interface (Rust). We show that eVCFI can protect Firefox with a small overhead of 1.15%. We believe that eVCFI is the first VCFI defense able to protect complex software like Firefox.

1 Introduction

C++ is a popular programming language used to implement large complex software systems such as web browsers [8]. However, C++ is also vulnerable to attacks such as *control flow hijacking*, where the attacker exploits a memory/type error to divert control to an attacker's chosen function/gadget, possibly granting arbitrary code execution. A common control flow hijack attack in C++ programs is to exploit *dynamic dispatch*. Modern C++ Application Binary Interfaces (ABIs), e.g., Itanium C++ ABI [1] (used by x86_64), implement dynamic dispatch using *Virtual Function Pointer Tables* (*vtables*) and *virtual member functions*. During a *virtual call* to a virtual member function, the corresponding function pointer in the *vtable* is called. However, this approach is vulnerable to attack, since pointer to the *vtable* (a.k.a., the *vptr*) is stored within the object itself, making it a potential target for type/memory errors. C++-specific variants of these attacks have also been developed, such as *Counterfeit Object-oriented Programming* (COOP) [18].

Virtual Call Control Flow Integrity (VCFI) is a proposed defense against virtual call control flow attacks [5,13,20]. VCFI defenses must efficiently validate a given object's *vptr* against the set of possible valid values, as determined by the full class hierarchy specified by the program and the `C++` language semantics. A state-of-the-art VCFI defense is shipped with the LLVM/`clang++` compiler (`cfi-vcall` in [14]), which uses *Link-time Optimization* (LTO) to extract the full class hierarchy at compile time. However, in addition to security and performance, *extensibility* is another critical design dimension for VCFI. For example, it is common for real-world software to be *modular*, i.e., interoperating with other modules through dynamic linking/loading. Furthermore, some software includes components with special interfaces, such as foreign language interfaces or *Component Object Models* (COM). Supporting such ad hoc extensions, even with manual intervention, necessitates a VCFI design that can handle "dynamic" class hierarchies—i.e., where the class hierarchy can be extended at runtime, or on-the-fly. We call this *class extensibility*, or simply *extensibility*.

The multithreaded Firefox browser exemplifies complex software architected with extensibility. Firefox uses dynamic loading, XPCOM (similar to COM), and foreign language interfaces, including `C++` and Rust components interacting with each other. Such extensibility is incompatible with many existing VCFI defenses. For example, in Firefox, a `C++` virtual call to an object implemented in Rust will trigger a VCFI violation, even though this is a benign (intentional) usage and not an attack. Our aim is to design an efficient VCFI defense that is compatible with extensibility requirements, expanding the applicability of VCFI to more `C++` software.

In this paper, we identify the requirements for extensible and efficient virtual call integrity in `C++` programs. We introduce a new VCFI defense, eVCFI, which is designed to support extensible `C++` software. To do so, we first cast the problem as an efficient set-membership question on dynamic sets, i.e., is a *vptr* a member of the "allow-set" of the corresponding class? There are many algorithmic tradeoffs with efficient set-membership, with different pros and cons. In this paper, we argue that *Approximate Membership Query* filters (AQM), such as Bloom filters [3], meet the design requirements of extensible VCFI. Specifically, we show that Bloom filters are efficient, i.e., $O(1)$ check that only needs a few instructions, regardless of the class hierarchy size or the number of modules. Furthermore, we show Bloom filters are extensible in that new entries can be added at runtime, without the need for thread locking/synchronization. In effect, our approach supports extensibility using a single underlying mechanism (Bloom filters), without fast/slow path logic. Finally, as our approach is *probabilistic*, we show how to enhance the security of Bloom filters using a combination of randomization and *eXecute Only Memory* (XOM).

We evaluate eVCFI against standard benchmarks (SPEC) and a web browser (Mozilla Firefox). Our results show that the default configuration of eVCFI incurs low-performance overheads on SPEC of 1.3%; and 1.01%, 1.18%, 1.15% overheads on Kraken, Octane, Dromaeo browser benchmarks. We use Firefox to showcase the challenges posed by complex `C++` software requiring non-trivial extensibility. We note that other VCFI defenses tend to fail on Firefox. This shows the importance of extensibility support, since an incompatible defense is as good as no defense, meaning that the software cannot be protected against virtual call control flow attacks. We believe eVCFI is the first approach that can harden the Firefox codebase with VCFI.

2 Overview

We summarize `C++` dynamic dispatch attack and defense, and then describe our attack model. For the rest of this paper, we focus on `C++` under `X86_64`/Linux.

2.1 C++ Dynamic Dispatch

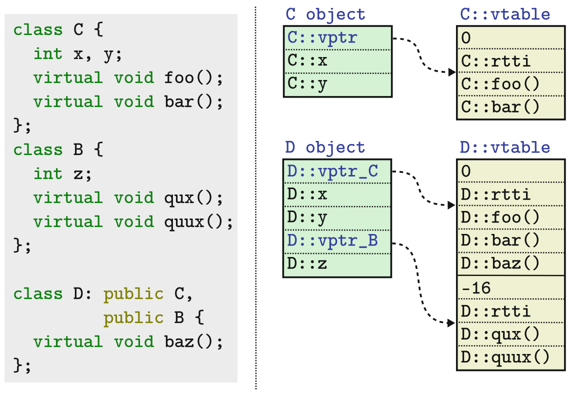
`C++` implements *dynamic dispatch* in the form of *virtual member functions*. A derived class can override the definition of a virtual function they inherit. Virtual calls, abbreviated as *vcall*, use a *Virtual Function Table* (*vtable*) to dynamically decide which virtual function definition is to be invoked. The *vtable* is essentially an array of function pointers, where each virtual function member of a class is mapped to a corresponding index in the function pointer array. The *vtable* is retrieved using a *Virtual Function Table Pointer* (*vptr*) stored in an implicit field (`vptr`) within the object itself. Given an object pointer (*objptr*), the basic template for a virtual call is: (1) read the (*vptr*) value from the implicit (*objptr*`->vptr`); and (2) call the function pointer at the corresponding index (*idx*) as follows:

vptr `=` *objptr*`->vptr;`   *vptr*`[`*idx*`](`...`);`

Here, we say that the *static type* of a virtual call site is given by `decltype(*`*objptr*`)`. Under the `C++` type system, the *dynamic type* of (*objptr*) of class `C` can be any (`D *`), where (`D`=`C`) or (`D`) is derived (possibly indirectly) from (`C`). This implements a form of polymorphism, where a derived class (`D`) can be upcast to the base class (`C`), but a virtual call still uses (`D`)'s definitions. The object and *vtable* layout is defined by the Itanium `C++` ABI [1] (used in `x86_64` Linux). Note the *vtable* is usually protected from modification using read-only memory.

Example 1. *(Class and* vtable *Layout)* An example class and *vtable* layout are illustrated in Fig. 1. Here, we consider a simple hierarchy with two base classes (`C`) and (`B`), and a derived class (`D`) that inherits from both. We consider a (`C`) and (`D`) object. As per the Itanium `C++` ABI, the (`C`) object has a single *vptr* pointing to the first *virtual function entry* in (`C::vtable`). Index (-1) contains the *Run-Time Type Information* (`C::rtti`) entry, and index (-2) contains the *offset-to-top* explained below. Class (`C`) does not inherit from another class, and only has two virtual functions, `C::foo` and `C::bar`.

The layout for the (`D`) object is more complex. Class (`D`) inherits from two base classes (`C`) and (`B`), i.e., *multiple inheritances*. So the (`D`) object has two *vptr*s, one for each proper base class. Furthermore, (`D::vtable`) is a *vtable group* that



```
class C {
  int x, y;
  virtual void foo();
  virtual void bar();
};
class B {
  int z;
  virtual void qux();
  virtual void quux();
};

class D: public C,
         public B {
  virtual void baz();
};
```

Fig. 1. Example class and *vtable* layout.

concatenates: (i) a *primary vtable* for the virtual function entries of (`D`) and the first proper base class of (`D`); and (ii) a *secondary vtable* for each proper base class. For secondary *vtables*, the *offset-to-top* is the pointer difference between the two *vptr* fields in the (`D`) object, in this case (-16). The derived class (`D`) can *override* the implementation of any virtual function from a base class. For example, (`D::foo`) will point to (`D`)'s implementation of inherited virtual function (`foo`). Otherwise, if (`D`) does not override (`foo`), the virtual function entry defaults to (`D::foo`=`C::foo`). □

C++ Dynamic Dispatch Security. The `C++` dynamic dispatch design is primarily intended for efficiency rather than security. Since `C++` is not a memory-safe language, an attacker can exploit a memory error (e.g., buffer overflow or use-after-free) to overwrite *vptr* values inside objects.

Object pointer integrity can also be violated using *type confusion* (e.g., a bad `C++ static_cast`) or *Counterfeit Object-oriented Programming* (COOP) style-attacks [18]. Likewise, *vptr* integrity can be violated using (sub-object) memory, type confusion, or use-after-free errors to directly overwrite the *vptr* value with another value. An attacker can exploit such errors to replace the object or *vptr* value with a new value of choice.

2.2 C++ Dynamic Dispatch Defenses

The mainstream defense to vcall control flow hijack is *Virtual Call Control Flow Integrity* (VCFI) which validates the correctness of the virtual function before invocation. VCFI is essentially a specialization of *Control Flow Integrity* (CFI) [2] to `C++` dynamic dispatch. The basic idea is to associate an *allow-set* of valid values to each virtual call site. There are two main variants: (*direct*) an allow-set of valid *virtual functions*; and (*indirect*) an allow-set of valid *vptr*s.

Under the assumption that *vtables* are read-only, the *direct* and *indirect* variants offer similar protection, with indirect being slightly stronger. Many VCFI implementations, such as LLVM-xDSO [14], use the *indirect* approach and so do we. Given a call to virtual function (`C::f()`), ideally the allow-set contains all *vptr*s allowable under the `C++` type system. This includes all *vptr*s to primary/secondary *vtable*s containing: (1) the entry for (`C::f()`), and (2) the entry for (`D::f()`) for all classes (`D`) derived from (`C`). For example, considering the class hierarchy from Fig. 1, the allow-set for the virtual call (`c->bar()`) is:

$$Allow = \{\texttt{C::vptr}, \texttt{D::vptr\_C}\}$$

VCFI asks if the vcall is in the allow-set at the virtual call site. The allow-set is determined by the strength of the implemented VCFI policy, e.g. Sect. 4.1 shows the effective result from different VCFI systems.

Many implementations of VCFI variants exist, see surveys [5,13,20]. However, most source-based VCFI defenses require that the class hierarchy is fixed at compile time, so the allow-set is also fixed. While monolithic and pure `C++` code can meet this assumption, complex software often goes beyond, sometimes using a wide range of heterogeneous components with diverse dynamic behavior on the class hierarchies and interfacing needs. This requires VCFI solutions that are *extensible* which is the focus of this paper. We highlight that a VCFI defense that is not extensible will simply fail on a codebase without a fixed class hierarchy. There are many tradeoffs needed to support extensibility. Here, we mention some (V)CFI implementations to provide context for the next section. Further details will be discussed in Sect. 5.

MCFI [16] identified the lack of modularity being an impediment for the adoption CFI, proposing a *Modular CFI* (MCFI) supporting separate compilation and dynamic library loading for C. RockJIT [17] extends MCFI with `C++` support. MCFI puts the program in a sandbox to protect its data structures, which can affect performance. A state-of-art VCFI is provided by LLVM [14,15]. LLVM has low overheads, but requires a fixed class hierarchy that is obtained using *Link-Time Optimization* (LTO). LTO allows the full (global) class hierarchy to be known at compile time. Although LLVM does not provide any extensibility, there also exists an experimental mode, namely *LLVM*-xDSO, that also supports dynamic linking and loading. The drawback is a much larger overhead, as we discuss and evaluate later. As can be seen, there are complex tradeoffs for VCFI, and existing solutions solve only a subset of our design goals. This can limit practical adoption, especially for complex/legacy software systems.

2.3 Problem Statement

Using the indirect VCFI check variant, we formalize a VCFI check using the class hierarchy of the program as follows:

$$c.\texttt{vcall}(\ldots); \qquad c.\texttt{vptr} \in Allow_C? \qquad \text{(VCFI-CHECK)}$$

where c is an object of static type C, and `vcall` denotes a virtual member function of C. The VCFI check determines whether *vptr* belongs to the allow-set for type C, where $D\texttt{::vptr} \in Allow_C$ for any D derived from C, or $D{=}C$.

In essence, VCFI has three main components:

– **Algorithms for the VCFI check**. This can also affect multi-threading.
– **Accuracy of the policy check**. We use Eq. (VCFI-Check). When extensibility is not used, the allow-set is determined statically (compile-time) under the `C++` semantics. If the program uses dynamic loading, the class hierarchy may be extended, meaning that the allow-set(s) need to be dynamically updated accordingly. Similarly, ad hoc extensions, such as foreign language interfaces, may also need to be reflected in the allow-set(s).
– **Performance of the check**. Ideally, overheads should be low.

As is common, we model a strong attacker capable of reading from or writing to arbitrary memory, subject to the program's memory protections. We assume the attacker cannot modify page permissions, and that there exists separate mitigation for hardware side channels [12]. We assume that the attacker intends to hijack control flow by compromising `C++` dynamic dispatch. Other kinds of control flow hijack are orthogonal to this paper. We assume *vtable*s reside in *read-only* memory (`.rodata`) and cannot be modified. We also assume that the attacker has not already hijacked control flow, a standard assumption for CFI-like defenses.

In order to enforce the VCFI policy, the *Allow*-set(s) must be constructed from a diverse and dynamic class hierarchy used by complex programs. We support various kinds of extensibility. Modularity by separate compilation allows extensibility by dynamic linking or dynamic loading during execution. This form of extensibility has automated support. Other kinds of extensibility generally used are: component object interfaces such as COM or XPCOM; and foreign language interfaces and language interoperation, e.g. between `C++` and Rust. We classify these under *ad hoc extensibility* and provide a basic extensibility mechanism to validate the custom vcalls.

3 Extensible VCFI Enforcement

In this paper, we seek flexibility in extending the allow-set at runtime either through dynamic loading or ad hoc extensions. In addition, the VCFI check should be secure, constant-time, and support multi-threading (as class hierarchy extensibility involves updates). By casting the VCFI defense as a *secure set membership test*, we can examine the known algorithmic set membership tradeoffs, where it is difficult to simultaneously satisfy design goals such as dynamic sets, constant time, multi-threaded support. Instead, we implement VCFI using an *Approximate Member Query* filter (AMQ), which allows for efficient set membership tests. AMQs are approximate, meaning that there can be false positives, but not false negatives.

Many possible AMQs have been proposed. In this paper, we use Bloom filters [3], which aligns well with our efficiency and extensibility design goals.

3.1 VCFI Based on Bloom Filters

Bloom filters [3] are the most well known form of AMQ. Traditionally, Bloom filters are implemented using a *bit array* B and a set of $k \geq 1$ hash functions

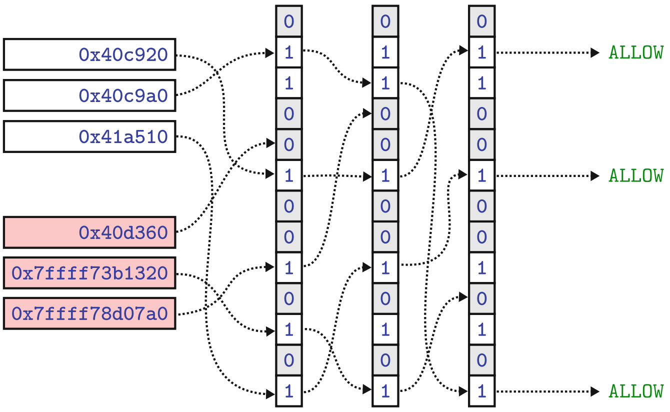


Fig. 2. Example Bloom filter VCFI defense for $k = 3$. Here, we assume that the three pointers `0x40c920`, `0x40c9a0` and `0x41a510` are the only valid members of the *Allow*-set. Each value is mapped $k = 3$ times to the Bloom filter using $k = 3$ *different hash functions*. Here, valid entries map to a non-zero value for each hash function; and the invalid values map to at least one zero value and are "filtered"

$hash_{1..k}$. An element x is considered to be a member of the set if:

$$B[hash_1(x)] \neq 0 \wedge \cdots \wedge B[hash_k(x)] \neq 0$$

Else, if the result is 0 for any $hash_i$, the element x is not a member of the set. Bloom filters are efficient, and testing for membership is constant time.

Figure 2 gives a basic example of a Bloom filter-based VCFI defense. We assume the only valid members of the allow set are: (*vptr*) values `0x40c920`, `0x40c9a0` and `0x41a510`, and that there are 3 hash functions ($k = 3$). All valid *vptr* values map to a non-zero entry and thus will be allowed by the VCFI defense. The invalid value `0x40d360` by the first hash maps to zero and is disallowed. Bloom filters are approximate meaning that collisions are possible, as will be discussed later.

VCFI benefits from the Bloom filter design in multiple ways. Firstly, Bloom filters are inherently *extensible*, meaning that new entries for class hierarchy extensions can be incrementally added at any time. Furthermore, set deletion (for dynamic *unloading*) is also supported using "counting" Bloom filters. Secondly, Bloom filters are inherently *efficient*, with an $O(1)$ set membership test that can be implemented using a few instructions. Finally, our Bloom filter implementation makes exclusive use of *atomic* operations to add/remove entries, thereby achieving *thread safety* without the need for thread synchronization.

3.2 System Design

eVCFI is an implementation of VCFI using Bloom filters, and consists of two main parts: (1) an LLVM-based *program transformation* to insert VCFI checks, and (2) a *runtime support* library.

```
movabs $SALT,%rdi    # Load 64-bit SALT
imul %rax,%rdi       # Multiply
xor %esi,%esi        # Zero accumulator
crc32q %rdi,%rsi     # CRC32
```

Fig. 3. Recipe for the *mulcrc* hash function. This example assumes the input *vptr* is stored in register `%rax`, and the output hash value is stored in `%rsi`.

```
mov (%rdi),%rax      # Load vptr
...                  # Hash into %rsi
movabs $BLOOM,%rdx   # Load Bloom base
testb $0,(%rdx,%rsi)
jnz .LOK             # Entry non-zero?
ud2                  # Invalid vptr
.LOK:
...                  # Repeat for k > 1
...                  # Setup parameters
callq *INDEX(%rax)   # Call virtualFn()
```

Fig. 4. Basic recipe for a hardened virtual call using (salted) Bloom filters

Program Transformation. To enforce the allow-set in a dynamic and extensible manner, we use program transformation to implement the (VCFI-Check) check using Bloom filters. The basic instrumentation schema is shown in Figs. 3 and 4. Here, Fig. 3 implements a single Bloom filter lookup, that is repeated k times, using the following *salted* hash function:

$$hash(salt, vptr) = crc32(salt \times vptr)$$

Note that the choice of the hash function is a tradeoff between performance and security. By design, eVCFI uses the salted hash function, *mulcrc*, which is parameterized by a *salt* constant, allowing for k different hash functions to be readily defined. The Fig. 4 schema implements the hardened vcall. The program transformation is implemented using an LLVM compiler pass.

Compared to the unprotected vcall, our Bloom filter VCFI uses an additional $8{*}k$ instructions (4 for the salted hash and 4 for the remainder of the check, repeated k times). For the minimal $k = 1$, there will be 8 additional instructions for the instrumented vcall, and 7 instructions in the execution path (see Figure 4). In contrast to other VCFI defenses, such as LLVM-xDSO, our solution does not use a fast/slow-path design. Instead, eVCFI uses a single $O(1)$ check uniformly for all vcalls, whether the call site needs dynamic extensibility or not. Dynamic linking/loading is handled by adding entries to the corresponding Bloom filter itself, as handled by the eVCFI runtime.

Runtime Support. A dynamic *Class Hierarchy Analysis* (CHA) is used to build the inheritance relationships between classes at runtime Conceptually, the CHA establishes a mapping between classes C and the the corresponding $Allow_C$ set. The CHA constructs the allow-sets from the (current) set of loaded modules, which may be updated at any time via dynamic (un)loading.

The hash function salts ($\texttt{SALT}_i$, $i \in 1..k$) and the base address of the Bloom filter (`BLOOM`) are encoded as special dynamic symbols, i.e., the eVCFI-symbols. During program initialization (i.e., before `main` is called), and for each dynamically loaded library, the eVCFI-symbols are initialized with suitable randomized values. To handle dynamic linking/loading, the CHA is incrementally applied to all classes in the loaded library. When the loaded library extends an existing class hierarchy with a new class, new entries are incrementally added to the corresponding allow-sets, i.e., by updating the corresponding Bloom filter. Dynamic unloading is handled similarly, by removing entries from allow-sets.

Security. We have introduced a VCFI defense based on Bloom filters. However, the Bloom filters themselves must also be hardened against attack.

- **Bloom Filter Integrity**. We ensure the integrity of the Bloom filter by making it *read-only.* To deal with updates, the most efficient way is to use the `X86_64` *Memory Protection Keys* (MPK) extension. When updating the Bloom filter, we grant the write permission to the thread using MPK, while other threads continue to have read-only access. An additional defense is also to randomize the location of the Bloom filter (`BLOOM`).
- **Missed Detection Mitigation**. The above protects against basic Bloom filter modification by the attacker. Nevertheless, Bloom filters are inherently *approximate*, meaning that *false positives* are possible. In practice, this means that an invalid $vptr'$ value may be accepted as valid by the Bloom filter, if the value happens to *collide* with another valid value.[1] This can be mitigated by increasing k, at the cost of performance. Alternatively, we can also randomize the hash functions by choosing CSPRNG-randomized value(s) for the $\texttt{SALT}_i$ at runtime. The randomized salt(s) make it difficult for the attacker to construct collisions.

  The attacker may also attempt to find a collisions by chance. The missed detection probability can be approximated using the formula: $(1 - e^{-kn/m})^k$, where m is the number of entries in the Bloom filter array, k is the number of hash functions, and n elements have been inserted. By default, eVCFI uses $m = 2^{24}$ and k is user-configurable, allowing for a security versus time trade-off. For example, assuming $n = 1000$, then the *missed detection* probability will be $\sim 5.96 \times 10^{-5}$ for $k = 1$, $\sim 5.96 \times 10^{-15}$ for $k = 3$, etc. Even for $k = 1$, brute force attacks are not practical for most applications, since the program will immediately abort on a single incorrect guess.
- **Runtime Protection**. The randomized $\texttt{SALT}_i$ parameters are encoded as immediate values in the instruction sequence that implement the salted hash function(s) (Fig. 3). In principle, the attacker may also attempt to recover these values by directly reading and interpreting the executable code residing in the program's memory. This can be directly prevented by using *Execute Only Memory* (XOM), which ensures that the instrumentation can only ever be executed, and never read. XOM is supported by Linux, using the standard `mprotect` system call, on all `X86_64` CPUs with MPK support.

Ad hoc Extensibility. Ad hoc extensibility covers cases where a vcall should be allowed (intended by the programmer), but would otherwise be detected as an error. This includes idioms that go beyond the semantics of `C++`, such as COM

[1] Note that, while missed detections are possible, *false detections* are not. That is, a Bloom-filter-based VCFI defense will never flag a valid vcall as invalid.

Table 1. VCFI comparisons

○: unprotected ◔/◕: partially protected ●: fully protected -: not applicable

VCFIs	Policies		Features		Static Overhead		
	Static	Dynamic	non-LTO	Ad Hoc	astar	omnetpp	xalanc
Baseline	○	○	✓	✗	0%	0%	0%
MCFI	◔	◔	✓	✗	35.7%	40.8%	53.6%
VTV	◕	◕	✓	✗	7.4%	4%	55.1%
ShrinkWrap	●	●	✓	✗	7%	6.1%	46.8%
LLVM	●	-	✗	✗	-0.2%	2.5%	2.9%
LLVM-xDSO	●	●	✗	✗	3.8%	4.9%	7.7%
eVCFI	●	●	✓	✓	1.3%	2.6%	8.5%

objects implemented as an opaque wrapper, which can be thought of as a programmatically defined foreign interface. Another example is objects defined in other languages that "inherit" from a base object defined in `C++`. Without any VCFI defense, there is usually no compatibility issue, provided the binary ABI is respected. With a VCFI, then we want to allow such ad hoc extensions if intended by the programmer. To support this, eVCFI supports programmer-specified "extension-lists" that can be used to insert additional entries to the allow-set(s). Although this approach is manual, it allows for arbitrary ad hoc extensions that are necessary for supporting complex software such as Firefox.

4 Evaluation

We compare eVCFI with other VCFI defenses and evaluate the performance on the SPEC2006 `C++` benchmark suite [11] and the Firefox web browser. All experiments run on Ubuntu (kernel version 4.13) with a Xeon Silver 4114 Processor (2.20GHz, 32GB of RAM). Both the processor and kernel support *Memory Protection Keys* (MPK) and *eXecute Only Memory* (XOM).

4.1 Evaluating VCFI Defenses

To give the overview of each (V)CFI implementation, we compare eVCFI against the security policies, features and overheads of: MCFI [16], VTV [19], Shrinkwrap [10] and LLVM. The results are summarized in Table 1. More information is provided in Appendix A.

The *Policies* column in Table 1 summarizes our test results on various vcall attacks using type confusion or memory corruption. The tool either prevents all attacks (●), or some attacks succeed (◔ or ◕), or all attacks succeed (○). We evaluate under several scenarios: (i) a static class hierarchy (the *Static* column); and (ii) a dynamic class hierarchy extension using dynamic loading with `dlopen()` (the *Dynamic* column). The baseline is without any VCFI defense, meaning that all vcall attacks succeed under all use cases.

For the *Static* case, MCFI exhibits the weakest policy under our testing. This is because MCFI implements a type-based CFI-policy, rather than a specialized VCFI policy. VTV implements a stronger policy, but does not detect derived class attacks under our tests. Finally, Shrinkwrap, LLVM and eVCFI all

Table 2. SPEC2006 C++ statistics

SPEC program	Static counts			Dynamic counts
	Lines of code	Number of vtables	Number of vcall-sites	Number of vcalls (Million)
`omnetpp`	26.7k	111	2218	3359.34
`astar`	4.3k	1	1	4996.99
`xalanc`	266.9k	958	21195	9821.91
`namd`	3.9k	4	0	0
`dealII`	94.5k	680	364	164.43
`soplex`	28.3k	29	638	3.18
`povray`	78.7k	28	286	0.15
Total	503.3k	1811	24702	18346

enforce an equivalent (strong) VCFI policy for the *Static* case. For the *Dynamic* case, most results are similar to *Static*, except for LLVM. This is because LLVM requires the (global) class hierarchy to be determined statically, through *Link Time Optimization* (LTO). However, this is not applicable when the class hierarchy is split between libraries.

Under *Features*, the *non-LTO* column indicates whether the VCFI defense is applicable without LTO. The *Ad Hoc* column indicates whether the defense supports ad hoc class hierarchy extensions, such as supporting COM objects or foreign language interfaces.

Finally, a summary of overheads (w.r.t. to *Baseline*) on the SPEC benchmarks is shown under *Static Overhead*. We note that only LLVM, LLVM-xDSO and eVCFI achieve a low performance overhead against the vcall-intensive `xalanc` benchmark. The overhead of eVCFI exceeds LLVM and LLVM-xDSO, but supports non-LTO compilation and ad hoc extensions. The full results are shown in Sect. 4.2 below.

4.2 Evaluation on SPEC Benchmarks

We evaluate the performance of eVCFI on the standard SPEC2006 `C++` benchmark which represents entire programs that are well-understood and extensively analyzed workloads. We run the (`ref`) workloads taking the geometric mean across five runs. Table 2 summarises the SPEC2006 `C++` benchmarks giving *source Lines Of Code* (sLOC), the number of *vtable*s, and the number of virtual call sites. Among the SPEC2006 benchmarks, `xalanc` has the most vcall sites and vcalls during runtime, thereby incurring the most performance overhead for VCFI defenses. eVCFI detects the known type confusion bug in `xalanc`. For details, see Appendix B. It has been patched for the benchmarking.

We evaluate each tool with `-O2` and *Link Time Optimization* (LTO) enabled. Although eVCFI does not require LTO, it is nevertheless compatible with LTO, and LTO is required by LLVM and thus is enabled for a fair comparison. As LTO is enabled with `-O2`, more optimization is enabled, meaning that some virtual calls may be *devirtualized*. For `namd`, this results in no virtual calls at runtime, thus, it is excluded. The experiments use the default eVCFI configuration: a 16MB Bloom filter ($m = 2^{24}$) and *mulcrc* is used as the (salted) hash function. The salt ($\texttt{SALT}_i$) parameters are randomized per run.

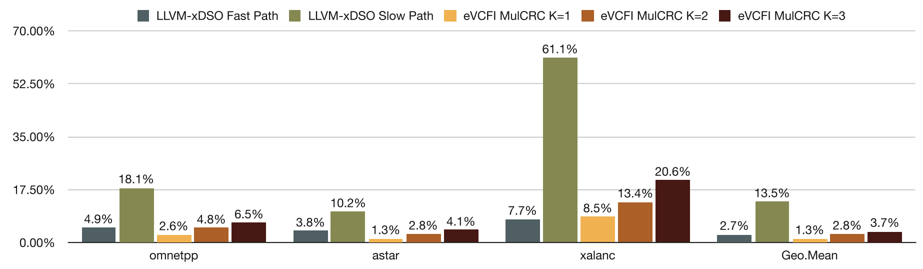


Fig. 5. Relative eVCFI overheads for SPEC `C++` programs

The SPEC2006 runtime performance is shown in Fig. 5. Columns for `dealII`, `soplex` and `povray` are omitted as the overheads are negligible (the number of vcalls is small, see Table 2), but summarized in the geometric mean (*Geo.Mean*) column. Even `dealII`, with 164M virtual calls, has negligible overhead, highlighting that both LLVM-xDSO and eVCFI have minimal overheads for programs that are not virtual call dominant. The overhead of all results is relative to the baseline, which is LLVM (`clang++`) with LTO and `-O2`. We compare the following:

LLVM-xDSO is the LLVM VCFI implementation with experimental "cross-`dso`" support. The implementation uses a fast/slow-path design, where the "fast" path is equivalent to the standard LLVM VCFI check. If the fast check fails, a "slow" path is invoked, which checks for dynamic class hierarchy extensions (e.g., `dlopen()`). The SPEC2006 benchmarks do not use extensibility features, meaning that only the fast-path will normally be invoked.

LLVM-xDSO-Slow is an artificially modified LLVM cross-`dso` that exclusively uses the slow-path VCFI check. This version is intended to represent the potential "worse case" behaviour of a fast/slow path design.

eVCFI is our implementation. We show results for eVCFI with $k = \{1, 2, 3\}$ to demonstrate different performance versus security tradeoffs.

For `omnetpp` and `astar` (with $k = 1, 2$) we see that eVCFI is faster than LLVM-xDSO for `omnetpp` and `astar`, and eVCFI has similar performance for `xalanc` and $k = = 1$. Generally, we see that the overheads of eVCFI increase with k, representing a performance trade-off. We also see that eVCFI is substantially faster than LLVM-xDSO-Slow, which highlights the advantage of single unified check rather than fast/slow-path design. For example, for the vcall-heavy `xalanc` benchmark, we see that LLVM-xDSO-Slow has a 61.1% overhead, compared to 8.5% for eVCFI. In summary, the geometric mean for SPEC2006 is: LLVM-xDSO 2.7%, LLVM-xDSO-Slow 13.5%, eVCFI k={1,2,3}: 1.3%, 2.8%, 3.7%.

4.3 Evaluating Firefox

We also evaluate eVCFI against the Firefox browser [8] version 78.0 ESR. Firefox is designed with different components and modules, including a foreign language

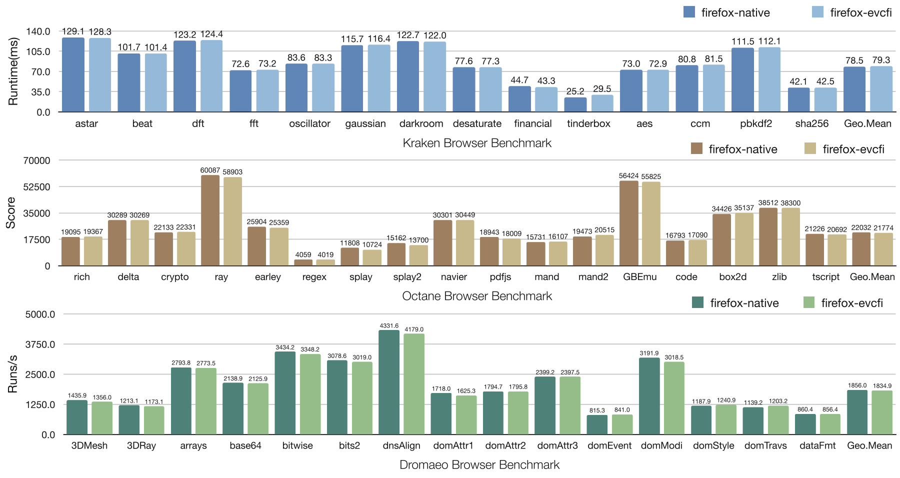


Fig. 6. Browser benchmarks for native Firefox and eVCFI-enhanced Firefox

interface between `C++` and Rust. Firefox is a real-world example of software requiring class hierarchy extensibility. Any VCFI defense that does not account for the extensibility requirements may incorrectly flag some vcalls as attacks, rather than intended behavior. We are also not aware of any existing VCFI defense that works with Firefox (since Rust versions). The Firefox build system currently does not support LLVM-xDSO.

Firefox consists of several binaries (executable and modules), and there are more than 5000 vtables and more than $185K$ virtual call-sites (most in `libxul.so` which is loaded using `dlopen()`). Firefox is also challenging because the code requires extensibility features, namely dynamic loading, foreign language interfaces (Rust), and XPCOM objects. Firefox is also multi-threaded.

In addition to dynamic loading, we use Firefox to test ad hoc extensibility. This involves creating an *extension-list* for the allow-sets to support specific Firefox idioms, such as XPCOM and vtables that were manually implemented in Rust. We remark that such ad hoc extensibility requires manual intervention (i.e., the specification of the extension-list). However, such manual intervention is necessary in the general case, since arbitrary ad hoc extensions cannot necessarily be detected automatically.

To show the practical performance of eVCFI, we evaluate the performance of Firefox using the Kraken, Octane [9], and Dromaeo [6] benchmarks.[2] The benchmark results are in Fig. 6. Overall, eVCFI exhibits low overheads, with the performance overheads on Kraken, Octane, and Dromaeo being 1.01%, 1.18%, and 1.15% respectively. We see that eVCFI has acceptable overheads consistent

[2] Due to insufficient horizontal space for all the Dromaeo results, we show a representative sample that has more differences from the Kraken and Octane results. The results not shown also have low overheads.

with the SPEC2006 `C++` results. We believe eVCFI is the only VCFI defense that has been evaluated against Firefox.

5 Related Work

Some surveys and evaluations of CFI and VCFI defenses are [5,13,20]. Here, we discuss relevant compiler-based VCFI works. ConFIRM [20] show extensibility features such as dynamic linking/loading and component support with interfaces beyond `C++` are common. Both [13,20] also show that very few VCFI defenses support extensibility. In this section, we discuss related work focusing on the tradeoffs of the (V)CFI defenses offering different forms of extensibility: MCFI, LLVM-xDSO, VTV (and ShrinkWrap). There are also binary VCFI systems not discussed, being incomparable to source-based ones, and are usually less accurate with more overhead [5,20].

In Sect. 2.2, we discussed MCFI and LLVM-xDSO. Other (V)CFI implementations with extensibility support include VTV [19] and ShrinkWrap [10]. VTV also highlights the importance of modularity and builds the set of *vtable*s at runtime to validate the *vptr*. The VTV work itself does not fully implement a VCFI policy (see Table 1). ShrinkWrap tightens the VTV policy for VCFI.

MCFI uses a transaction-based framework to update its data structures to support multithreading, and VTV needs to synchronize threads to prevent data races. In contrast, eVCFI uses Bloom filters that naturally support atomic updates, which both simplifies the handling of multi-threaded programs as well as being more efficient.

The underlying data structures need to be updated for extensibility. Thus, any update also needs to be secure against attacks. MCFI secures its data structures using a sandbox design. To do so, MCFI effectively limits the program to a 32-bit address space [16]. However, this can easily introduce incompatibilities, especially with programs that use large amounts of virtual memory. VTV makes its data structures read-only, and updates need to block other threads. In eVCFI, only the thread updating the Bloom filter has write permission using MPK. This avoids the need for synchronization and operating system intervention.

In MCFI, the overhead is a combination of the sandboxing and the cost of the CFI check itself. The results in [17] show that, due to the sandboxing overhead, even programs like `namd`, `soplex` and `povray` incur a performance penalty. In contrast, these benchmarks, with few vcalls, incur negligible cost (near zero) for both LLVM and eVCFI, This is also confirmed by our results in Table 1, see `astar` which has relatively fewer vcalls.

The overheads for VTV on SPEC from [19] are: `omnetpp` 8%, `astar` 2.4%, `xalanc` 19.2%. Similar overheads were reproduced in [5]. Generally, these overheads are much greater than LLVM, which is not surprising as LLVM succeeds VTV. Our timings in Table 1 also give another reference point. However (non-cross-`dso`) LLVM does not support extensibility and being a purely static solution, should have the lowest overhead. Still we see that eVCFI (with extensibility) can compete. We also see that eVCFI is much faster than LLVM-xDSO-Slow.

Specifically, eVCFI has $O(1)$ time guarantees, regardless of the extensibility usage while also adhering to the language semantics VCFI policy.

6 Conclusion

It is common for large/complex software to be broken into different modules, libraries or plugins that may be loaded dynamically. Furthermore, software may need to support ad hoc extensions, such as foreign language interfaces or COM objects. Such extensibility is generally incompatible with most existing VCFI defenses, or the defense is prohibitively slow. As such, no defense will be used, potentially leaving the program vulnerable.

In this paper, we presented a new VCFI defense based on *Approximate Member Query* (AMQ) filters, specifically Bloom filters. We show that Bloom filters can be used to implement an efficient VCFI defense, in the form of the eVCFI tool. Specifically, eVCFI supports $O(1)$ checks that are implemented in a few instructions. Furthermore, eVCFI supports dynamic loading and ad hoc extensions for multi-threaded programs, without relying on a fast/slow path design. We also show how Bloom filters can be hardened using a combination of randomization and *eXecute Only Memory* (XOM). We believe eVCFI is the first VCFI defense which can be used to harden Firefox—a challenging target that uses dynamic linking/loading, component interfaces, and `C++` to Rust interoperation. Our Firefox compatibility testing shows eVCFI can provide greater extensibility support than existing VCFI defenses.

As future work, we believe our underlying design could also be adapted to other CFI-like defenses, beyond VCFI. Essentially, any defense that depends on a set membership query, including both source-based and binary CFI defenses, can likely use our approach.

Acknowledgments. We thank the anonymous reviewers for their valuable comments. This work has been supported in part by the Ministry of Education, Singapore (Grant No. MOE2018-T2-1-142) and by the National Research Foundation, Singapore under its NSoE DeST-SCI programme (Grant No. NSoE_DeST-SCI2019-0006). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

A VCFI Test Programs

To evaluate the security and usability of VCFI defenses, we construct a test program using the following class hierarchy:

```
class A { public: virtual void f() = 0; };    class A1: public A   { public: void f() {...} };
class B { public: virtual void f() = 0; };    class A2: public A   { public: void f() {...} };
class C { public: virtual int  f() = 0; };    class A11: public A1 { public: void f() {...} };
                                              class B1: public B   { public: void f() {...} };
                                              class C1: public C   { public: int  f() {...} };
```

Table 3. VCFI policy test results

	Class hierarchy	Ideal	MCFI	VTV	ShrinkWrap	LLVM	LLVM-xDSO	eVCFI
Static								
TypeConf	Sibling(A2)	✓	✗	✓	✓	✓	✓	✓
TypeConf	Derived(A11)	✓	✗	✗	✓	✓	✓	✓
TypeConf	InterClass(B1/C1)	✓	✓	✓	✓	✓	✓	✓
MemCorr	Sibling(A2)	✓	✗	✓	✓	✓	✓	✓
MemCorr	Derived(A11)	✓	✗	✗	✓	✓	✓	✓
MemCorr	InterClass(B1/C1)	✓	✓	✓	✓	✓	✓	✓
Dynamic								
TypeConf	Sibling(A2)	✓	✗	✓	✓	-	✓	✓
TypeConf	Derived(A11)	✓	✗	✗	✓	-	✓	✓
TypeConf	InterClass(B1/C1)	✓	✓	✓	✓	-	✓	✓
MemCorr	Sibling(A2)	✓	✗	✓	✓	-	✓	✓
MemCorr	InterClass(B1)	✓	✗	✓	✓	-	✓	✓
MemCorr	InterClass(C1)	✓	✓	✓	✓	-	✓	✓

The test program implements common vcall vulnerabilities, including COOP, type confusion and memory corruption. We also test both a static and dynamic class hierarchy, with the latter extended via `dlopen()`. The results are shown in Table 3. Here, the *Ideal* column represents the expected result for a complete VCFI defense. We use the following notation:

“✗”: The defense does not protect against the vcall attack.
“✓”: The defense works correctly and aborts the program preventing the attack.
“-”: This is when the defense is incompatible For example, LLVM (non-cross-`dso`) does not support extensions using `dlopen()`.

B Invalid Virtual Call Detected in `xalanc`

The eVCFI tool detects an invalid virtual call on line 1018 of `SchemaValidator.cpp` from the `xalanc` benchmark:

```
SchemaGrammar& sGrammar =
  (SchemaGrammar&) grammarEnum.nextElement();
sGrammar.getGrammarType();
```

At runtime, the (`grammarEnum.nextElement`) member function may return a reference to an object of type (`DTDGrammar`) that is not derived from the class (`SchemaGrammar`). This bug has been independently detected by other tools, including LLVM-xDSO [14], *vtable* interleaving [4] and type confusion sanitizers such as EffectiveSan [7]. For the performance evaluation, we patched `xalanc` to remove the bad cast and resolve the invalid virtual call.

References

1. Itanium C++ ABI (2022). http://itanium-cxx-abi.github.io/cxx-abi/
2. Abadi, M., Budiu, M., Erlingsson, Z., Ligatti, J.: Control-flow integrity. In: Computer and Communication Security. ACM (2005)
3. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
4. Bounov, D., Kici, R., Lerner, S.: Protecting C++ dynamic dispatch through VTable interleaving. In: Network and Distributed Systems Security. The Internet Society (2016)
5. Burow, N., et al.: Control-flow integrity: precision, security, and performance. ACM Comput. Surv. **50**(1), 1–33 (2017)
6. Dromaeo (2022). https://github.com/jeresig/dromaeo
7. Duck, G., Yap, R.: EffectiveSan: type and memory error detection using dynamically typed C/C++. In: ACM-SIGPLAN Symposium on Programming Language Design and Implementation. ACM (2018)
8. Firefox Web Browser (2022). https://www.mozilla.org/
9. Octane 2.0 (2022). http://chromium.github.io/octane/
10. Haller, I., Goktas, E., Athanasopoulos, E., Portokalidis, G., Bos, H.: ShrinkWrap: VTable protection without loose ends. In: Annual Computer Security Applications Conference. ACM (2015)
11. Henning, J.: SPEC CPU2006 benchmark descriptions. Comput. Arch. News **34**(4), 1–17 (2006)
12. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: Security and Privacy. IEEE (2019)
13. Li, Y., Wang, M., Zhang, C., Chen, X., Yang, S., Liu, Y.: Finding cracks in shields: on the security of control flow integrity mechanisms. In: Computer and Communication Security. ACM (2020)
14. LLVM (2022). https://clang.llvm.org/docs/ControlFlowIntegrity.html
15. LLVM (2022). https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html
16. Niu, B., Tan, G.: Modular control-flow integrity. In: Programming Language Design and Implementation. ACM (2014)
17. Niu, B., Tan, G.: RockJIT: securing just-in-time compilation using modular control-flow integrity. In: Computer and Communication Security. ACM (2014)
18. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A., Holz, T.: Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications. In: Security and Privacy. IEEE (2015)
19. Tice, C., et al.: Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Security Symposium. USENIX (2014)
20. Xu, X., Ghaffarinia, M., Wang, W., Hamlen, K.: CONFIRM: evaluating compatibility and relevance of control-flow integrity protections for modern software. In: Security Symposium. USENIX (2019)