Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM

Caroline Tice *Google, Inc.*
Tom Roeder *Google, Inc.*
Peter Collingbourne *Google, Inc.*
Stephen Checkoway *Johns Hopkins University*
Úlfar Erlingsson *Google, Inc.*
Luis Lozano *Google, Inc.*
Geoff Pike *Google, Inc.*

Abstract

Constraining dynamic control transfers is a common technique for mitigating software vulnerabilities. This defense has been widely and successfully used to protect return addresses and stack data; hence, current attacks instead typically corrupt vtable and function pointers to subvert a forward edge (an indirect jump or call) in the control-flow graph. Forward edges can be protected using Control-Flow Integrity (CFI) but, to date, CFI implementations have been research prototypes, based on impractical assumptions or ad hoc, heuristic techniques. To be widely adoptable, CFI mechanisms must be integrated into production compilers and be compatible with software-engineering aspects such as incremental compilation and dynamic libraries.

This paper presents implementations of fine-grained, forward-edge CFI enforcement and analysis for GCC and LLVM that meet the above requirements. An analysis and evaluation of the security, performance, and resource consumption of these mechanisms applied to the SPEC CPU2006 benchmarks and common benchmarks for the Chromium web browser show the practicality of our approach: these fine-grained CFI mechanisms have significantly lower overhead than recent academic CFI prototypes. Implementing CFI in industrial compiler frameworks has also led to insights into design tradeoffs and practical challenges, such as dynamic loading.

1 Introduction

The computer security research community has developed several widely-adopted techniques that successfully protect return addresses and other critical stack data [13, 20]. So, in recent years, attackers have changed their focus to non-stack-based exploits. Taking advantage of heap-based memory corruption bugs can allow an attacker to overwrite a function-pointer value, so that arbitrary machine code gets executed when that value is used in an indirect function call [6]. Such exploits are referred to as *forward-edge* attacks, as they change forward edges in the program's control-flow graph (CFG).

To make these attacks more concrete, consider a C++ program that makes virtual calls and has a use-after-free bug involving some object. After the object is freed, an attacker can reallocate the memory formerly occupied by the object, overwriting its vtable pointer. Later virtual calls through this object get the attacker's vtable pointer and jump to a function from the attacker's vtable. Such exploits are becoming commonplace, especially for web browsers where the attacker can partially control executed JavaScript code [14, 23].

Control-Flow Integrity (CFI) [1] guards against these control-flow attacks by verifying that indirect control-flow instructions target only functions in the program's CFG. However, although CFI was first developed over a decade ago, practical CFI enforcement has not yet been adopted by mainstream compilers. Instead, CFI implementations to date are either ad-hoc mechanisms, such as heuristic-driven, custom binary rewriting frameworks, or experimental, academic prototypes based on simplifying assumptions that prevent their use in production compilers [1, 9, 12, 29, 31–34].

In this paper, we present implementations of two mechanisms that provide forward-edge CFI protection, one in LLVM and one in GCC. We also provide a dynamic CFI analysis tool for LLVM which can help find forward-edge control-flow vulnerabilities. These CFI implementations are fully integrated into their respective compilers and were developed in collaboration with their open source communities. They do not restrict compiler optimizations, operation modes, or features, such as Position-Independent Code (PIC) or C++ exceptions. Nor do they restrict the execution environment of their output binaries, such as its use of dynamically-loaded libraries or Address Space Layout Randomization (ASLR).

The main contributions of this paper are:

- We present the first CFI implementations that are fully integrated into production compilers without restrictions or simplifying assumptions.
- We show that our CFI enforcement is practical and highly efficient by applying it to standard benchmarks and the Chromium web browser.
- We identify, discuss, and resolve the main challenges in the development of a real-world CFI implementation that is compatible with common software-engineering practices.

All our mechanisms verify targets of forward-edge indirect control transfers but at different levels of precision, depending on the type of target and the analysis applied. For example, C++ indirect-control transfers consist mostly of virtual calls, so one of our approaches focuses entirely on verifying calls through vtables. Our security analyses show that our CFI mechanisms protect from 95% to 99.8% of all indirect function calls. They are also highly efficient, with a performance penalty (after optimizations) ranging from 1% to 8.7%, as measured on the SPEC CPU2006 benchmark suite and on web browser benchmarks.

Most notably, our CFI mechanisms compare favorably to recent CFI research prototypes. The security guarantees of our work differ from these prototypes, but a comparison is nonetheless instructive, since our attack model is realistic, and our defenses give strong guarantees.

MIP [28] and CCFIR [34] state efficiency as their main innovation. In particular, on the SPEC Perl benchmarks (where they both were slowest), CCFIR reports 8.6% overhead, and MIP reports 14.9% to 31.3% overhead. Our mechanisms have a corresponding overhead of less than 2%, as we report in Section 7.2. For C++ benchmarks, which perform indirect calls more frequently, the performance differences can be even greater. Another recent CFI implementation, bin-CFI [35], reports overheads of 12% on the SPEC Perl benchmarks, but 45% for the C++ benchmark omnetpp, while the overhead is between -1% and 6.5% for omnetpp compiled using our mechanisms. Even the most recent CFI implementation, SAFEDISPATCH [19], must sacrifice software-engineering practicality and use profile-driven, whole-program optimization to achieve overheads comparable to ours (roughly 2% for all three of their Chromium benchmarks).

2 Attacks and Compiler-based Defenses

Software is often vulnerable to attacks that aim to subvert program control flow in order to control the software's behavior and assume its privileges. Typically, successful attacks exploit software mistakes or vulnerabilities that allow corruption of low-level state.

To thwart these low-level attacks, modern compilers, operating systems, and runtime libraries protect software integrity using a variety of techniques, ranging from coarse-level memory protection, through address-space layout randomization, to the fine-grained type-safety guarantees of a high-level language. In particular, machine-code memory is commonly write protected, and thread execution stacks are protected by placing them at random, secret locations, and by checking that secret values (a.k.a. canary values) remain unmodified. This guards return addresses and other stack-based control data against unintended overwriting [7, 13].

As a result of such compiler-based defenses becoming widely used, corruption of the execution stack or machine code has become a far less common means of successful attack in well-maintained, carefully-written software such as web browsers [20]. However, there has been a corresponding increase in attacks that corrupt program-control data stored on the heap, such as C++ vtable pointers (inside objects) [14], or function pointers embedded in data structures; these attacks subvert indirect control transfers and are known as "return-to-libc" or return-oriented programming [3, 8, 22, 25–27, 30].

In this paper we present three compiler-based mechanisms for further protecting the integrity of program control data. Focusing on the integrity of control-transfer data stored on the heap, two of our mechanisms enforce forward-edge CFI by restricting the permitted function pointer targets and vtables at indirect call sites to a set that the compiler, linker, and runtime have determined to be possibly valid. The third mechanism is a runtime analysis tool designed to catch CFI violations early in the software development life-cycle. Our mechanisms are efficient and practical and have been implemented as components of the GCC and LLVM production compiler toolchains. While they differ in their details, and in their security — such as in how precisely the program's CFG is enforced — all three of our implementations:

- add new, read-only metadata to compilation modules to represent aspects of the program's static CFG;
- add machine code for fast integrity checks before indirect forward-edge, control-flow instructions;
- optionally divert execution to code that performs slower integrity checks in certain complex cases;
- call out to error handlers in the case of failures; and
- may employ runtime library routines to handle important exceptional events such as dynamic loading.

Like all defenses, we aim to prevent certain threats and not others, according to an attack model. As in the original work on CFI, our model pessimistically assumes that an adversary can arbitrarily perturb most writable program data at any time [1]. The program code, read-only data, and thread register state cannot be modified. While pessimistic, this attack model has stood the test of time, and is both conceptually simple and realistic.

Similar to recent independent CFI work done concurrently with ours [19], and motivated by attackers' increasing focus on heap-based exploits, our mechanisms protect only forward-edge control transfers. Our attack model does not contain many types of stack corruption, since, as stated previously, effective defenses against such corruption are already in common use. Thus, our choice of attack model differs from most earlier work on CFI, except for work on mechanisms like XFI [12], which place the stack outside of pointer-accessible memory.

In our attack model we also depart from most previous CFI work by choosing to trust the compiler toolchain. For the integration of general-purpose defenses in production compilers, we find relying only on stand-alone verification of the final, output binaries to be impractical — although well-suited to custom compilers for specific scenarios, such as in Google's Native Client [17, 21]. While eliminating trust in the compiler is a laudable goal [24], doing so increases complexity, reduces portability, and prevents optimizations, while providing only uncertain benefits. In particular, we know of no exploits on previous compiler-based defenses that justify the software engineering costs of eliminating trust in the compiler.

By adopting the above attack model, our mechanisms are practical as well as efficient. While many experimental CFI mechanisms have been constructed and described in the literature, none have been able to efficiently provide strong, precise integrity guarantees with the full support that programmers demand from a production compiler. In particular, incremental compilation and dynamic libraries have remained primary challenges for CFI implementations, as has achieving low performance overheads; these challenges have only recently started to be addressed in experimental prototypes that enforce more coarse-grained CFI policies [28, 34]. However, CFI enforcement that is too coarse grained may provide only limited protection [4, 10, 15, 16] against modern attackers.

In summary, by focusing on forward-edge CFI, and fully integrating into compilers, our mechanisms can enforce fine-grained CFI at a runtime overhead that improves on that of the best previous work.

Related Work. Following the original 2005 work on CFI, later revised as Abadi et al. [1], there have been a number of implementations that have extended or built-upon CFI: XFI by Erlingsson et al. [12], BGI by Castro et al. [5], HyperSafe by Wang and Jiang [31], CFI+Sandboxing by Zeng et al. [32], MoCFI by Davi et al. [9], CCFIR by Zhang et al. [34], Strato by Zeng et al. [33], bin-CFI by Zhang et al. [35], MIP by Niu et al. [28], and SAFEDISPATCH by Jang et al. [19].

These CFI-based mechanisms vary widely in their goals, tradeoffs and implementation details. To achieve low overhead, many enforce only coarse-grained CFI, which may be a weak defense [15].

XFI, Strato, HyperSafe, and BGI use control-flow integrity primarily as a building block for higher-level functionality, such as enforcing software-based fault isolation (SFI), or fine-grained memory-access controls. Some, like XFI, focus on statically verifying untrusted binary modules, to establish that CFI will be correctly enforced during their execution, and thus that they can be used safely within different address spaces, such as the OS kernel.

Many implementations of CFI are based on binary rewriting. XFI and the original work on CFI used the sound, production-quality Windows binary rewriter, Vulcan [11], as well as debug information in PDB files. These implementations construct precise control-flow graphs (CFGs) to ensure that all indirect control transfers are constrained in a sound manner. Other implementations — including MoCFI, CCFIR, and bin-CFI — are based on more ad hoc and fragile mechanisms. For example, MoCFI produces an imprecise CFG for ARM applications running on an iPhone based on runtime code dumping, disassembly, and heuristics. CCFIR relies on relocation tables and recursive disassembly with heuristics to identify code that needs to be protected. The code is then rewritten to use a special randomized springboard section through which all indirect control transfers happen. Bin-CFI also uses heuristic disassembly; however, unlike CCFIR, bin-CFI injects a CFI-protected copy of the text section into the original binary and uses dynamic binary translation to convert pointers between the two code copies at runtime.

Other implementations, including HyperSafe, MIP, CFI+Sandboxing, and SAFEDISPATCH are implemented as modifications to the compiler toolchain and compile source code to binaries with CFI protection. The first three are implemented as rewriting either assembly or the compiler's Intermediate Representation (IR) of machine code — essentially a more precise form of binary rewriting.

Our vtable verification (see Section 3) is most similar to SAFEDISPATCH: work done independently and concurrently with ours that adds passes to LLVM for instrumenting code with runtime CFI checks, and relies on profile-driven, whole-program optimization to reduce enforcement overhead. At a call site, SAFEDISPATCH checks that either (1) the vtable pointer is to a valid vtable, for the call site, or (2) the address in the vtable points to a valid method for the call site. However, SAFEDISPATCH disallows separate compilation, dynamic libraries, etc., and relies on profile-driven, whole-program optimization, which is not very practical.

The overhead of all these various CFI mechanisms ranges from 6% to 200%, with some significant variations. In particular, the recent papers on MIP, CCFIR, and Strato state their low CFI enforcement overhead as a main contribution; Strato also highlights its support for compiler optimizations. However, as mentioned previously, their overheads on comparable benchmarks are several times larger than those of our CFI mechanisms — and they are likely to perform even worse on C++ benchmarks. This is due in part to the different properties our work enforces: we limit our scope to protecting control data that is not already well-protected by other mechanisms.

3 VTV: Virtual-Table Verification

Vtable Verification (VTV) is a CFI transformation implemented in GCC 4.9 for C++ programs. VTV protects only virtual calls and does not attempt to verify other types of indirect control flow. However, most indirect calls in C++ are virtual calls (e.g., 91.8% in Chrome), making them attractive targets for attackers. VTV is our most precise CFI approach: it guarantees that the vtable used by each protected virtual call is both valid for the program and also correct for the call site.

3.1 Problem Description

Virtual calls are made through objects, which are instances of a particular class, where one class (e.g., `rectangle`) inherits from another class (e.g., `shape`), and both classes can define the same function (e.g., `draw()`), and declare it to be virtual. Any class that has a virtual function is known as a *polymorphic class*. Such a class has an associated virtual function table (*vtable*), which contains pointers to the code for all the virtual functions for the class. During execution, a pointer to an object declared to have the type of a parent class (its *static type*, e.g., `shape`) may actually point to an object of one of the child classes (its *dynamic type*, e.g., `rectangle`). At runtime, the object contains a pointer (the *vtable pointer*) to the appropriate vtable for its dynamic type. When it makes a call to a virtual function, the vtable pointer in the object is dereferenced to find the vtable, then the offset appropriate for the function is used to find the correct function pointer within the vtable, and that pointer is used for the actual indirect call. Though somewhat simplified, this explanation is generally accurate.

The vtables themselves are placed in read-only memory, so they cannot be easily attacked. However, the objects making the calls are allocated on the heap. An attacker can make use of existing errors in the program, such as use-after-free, to overwrite the vtable pointer in the object and make it point to a vtable created by the attacker. The next time a virtual call is made through the object, it uses the attacker's vtable and executes the attacker's code.

3.2 Overview of VTV

To prevent attacks that hijack virtual calls through bogus vtables, VTV verifies the validity, at each call site, of the vtable pointer being used for the virtual call, before allowing the call to execute. In particular, it verifies that the vtable pointer about to be used is correct for the call site, i.e., that it points either to the vtable for the static type of the object, or to a vtable for one of its descendant classes. VTV does this by rewriting the IR code for making the virtual call: a verification call is inserted after getting the vtable pointer value out of the object (ensuring the value cannot be attacked between its verification and its use) and before dereferencing the vtable pointer. The compiler passes to the verifier function the vtable pointer from the object and the set of valid vtable pointers for the call site. If the pointer from the object is in the valid set, then it gets returned and used. Otherwise, the verification function calls a failure function, which normally reports an error and aborts execution immediately.

3.3 More Details

VTV differs from all previous compiler-based CFI implementations in that it allows incremental compilation rather than requiring that all files be recompiled if a single one is changed. VTV also does not forbid or restrict dynamic library loading. Such requirements and restrictions are common in research prototypes but impractical for real world systems.

Because VTV allows incremental compilation and dynamic loading, it must assume that its knowledge of the class hierarchy is incomplete during any particular compilation. Therefore, VTV has two pieces: the main compiler part, and a runtime library (*libvtv*), both of which are part of GCC. In addition to inserting verification calls at each call site, the compiler collects class hierarchy and vtable information during compilation, and uses it to generate function calls into libvtv, which will (at runtime) build the complete sets of valid vtable pointers for each polymorphic class in the program.

To keep track of static types of objects and to find sets of vtable pointers, VTV creates a special set of variables called *vtable-map variables*, one for each polymorphic class. At runtime, a vtable-map variable will point to the set of valid vtable pointers for its associated class. When VTV inserts a verification call, it passes in the appropriate vtable-map variable for the static type of the object, which points to the set to use for verification.

Because our vtable-pointer sets need to be built before any virtual calls execute, VTV creates special constructor init functions and gives them a high priority. The compiler inserts into these special functions the calls for building vtable-pointer sets. These functions run before standard initialization functions, which run before `main`, ensuring the data is in place before any virtual calls are made.

Vtable-map variables and vtable-pointer sets need to be read only to avoid introducing new vectors for attack. However, they must be writable when they are first initialized and whenever a dynamic library is loaded, since the dynamic library may need to add to the vtable-pointer sets. So, we need to be able to find all our data quickly. To keep track of the vtable-pointer sets, we wrote our own memory allocation scheme based on mmap. VTV uses this scheme when creating the sets; this lets it find all such sets in memory.

To find vtable-map variables, VTV writes them into a special named section in the executable, which is page-aligned and padded with a page-sized amount of zeros to prevent any other data from residing on the same pages. Before updating its data, VTV finds all memory pages that contain vtable-map variables and vtable-pointer sets and makes them writable. When it finishes the update, it finds all appropriate pages and makes them read-only. Thus, the only times these VTV data structures are vulnerable to attack are during program initialization, and possibly during `dlopen` calls. The VTV code that updates our data structures uses pthread mutexes to prevent races between multiple threads.

3.4 Practical Experience with VTV

We encountered some challenges while developing VTV.

Declaring global variables. Originally, VTV declared vtable-map variables as COMDAT and as having global visibility, because incremental compilation can result in multiple compilation units defining the same vtable-map variable, and COMDAT sections can be coalesced by the linker. However, this did not work reliably because there are many ways in which programmers can override the visibility of a symbol: linker version scripts can explicitly declare which symbols have global visibility; the rest of the symbols become hidden by default. Also, if global symbols are not added to the dynamic symbol table, then vtable-map variables might not be global. Finally, calls to `dlopen` with the RTLD_LOCAL flag have similar effects. We found all these techniques at work in Chromium.

When some vtable-map variables do not have global visibility, there can be multiple live instances of a vtable-map variable for a particular class, each pointing to a different vtable-pointer set containing only part of the full vtable-pointer set for the class. If the verification function is passed a variable pointing to the wrong part of the set, execution aborts incorrectly. We call this the *split-set problem*.

We finally concluded that there is no existing mechanism for the compiler to ensure a symbol will always be globally visible. The only way to eliminate the split-set problem was to accept that there would be multiple live versions of some vtable-map variables. To handle the consequences of this new assumption, VTV keeps track of the first instance of a vtable-map variable for each class. When initializing any vtable-map variable, it first checks if it has already seen a version of that variable. If not, then it allocates a vtable-pointer set for the variable, makes the variable point to the vtable-pointer set, and registers the variable in its variable registry. All subsequent vtable-map variables for that class are then initialized to point to the same vtable-pointer set as the first one.

Mixing verified and non-verified code. VTV causes execution to halt for one of three reasons: (1) a vtable pointer has been corrupted; (2) the C++ code contains an incorrect cast between two class types (programmer error); or (3) the set of valid vtable pointers used for verification is incomplete. The split-set problem is an example of the last case. This can also occur if some files that define or extend classes are instrumented with vtable verification, and other files that define or extend part of the class hierarchy are not. A similar effect also can occur with libraries or plugins that pass objects in and out, if one is instrumented and the other is not.

lib.cc

```
#include "lib.h"
struct Derived_Priv : public Base {
  virtual ~Derived_Priv() {}
};

Base *GetPrivate() {
  return new Derived_Priv;
}
void Destroy(Base *pb) {
  delete pb; // virtual call #1
}
```

main.cc

```
#include "lib.h"
struct Derived : public Base {
        virtual ~Derived() {}
};

int main() {
  Derived *d = new Derived;
  Destroy(d);
  Base *pp = GetPrivate();
  delete pp; // virtual call #2
}
```

main.cc w/ VTV	lib.cc w/ VTV	vcall #1 OK	vcall #2 OK	Missing from vtable pointer set
Yes	Yes	Yes	Yes	Nothing
No	No	Yes	Yes	Nothing
Yes	No	Yes	No	`Derived_Priv`
No	Yes	No	Yes	`Derived`

Figure 1: Example of problems resulting from mixing verified and unverified code. If only main.cc is compiled with verification, the vtable pointer for `Derived_Priv` does not get added to the valid set for Base, so virtual call #2 fails to verify. If only lib.cc is compiled with verification, the vtable pointer for `Derived` does not get added to the valid set for Base, so virtual call #1 fails.

Figure 1 shows this problem: a header file, lib.h, declares a base class `Base`. The class `Base` contains one virtual function, its destructor. There are two source files, lib.cc and main.cc, that each includes lib.h and contains classes that inherit from `Base`, as shown in the upper part of Figure 1. The table in the lower part of Figure 1 shows the effects on the two marked virtual calls of compiling lib.cc and main.cc with and without VTV. Note that the only cases where both virtual calls pass verification are when everything is built with VTV or nothing is.

We encountered this problem in ChromeOS with the Chrome browser. There are two third-party libraries which are built without VTV and are distributed to ChromeOS as stripped, obfuscated binaries (these binaries are not part of the open-source Chromium project). To make matters worse, when we built the rest of Chrome and ChromeOS with VTV and ran tests that exercised those libraries, we encountered verification failures.

To deal with the mixed-code problem in general, VTV was designed and written with a replaceable failure function. This function gets called if the verification function fails to find a vtable pointer in a valid set. To replace the default failure function, a programmer writes a replacement function (using the same signature, provided in a header file in libvtv), compiles it, and links it into the final binary. For Chrome we replaced the default failure function with a whitelist failure function. The whitelist function maintains an array with one record for each whitelisted library. The record contains the memory address range where the readonly data section for the library (which contains all of the library's vtables) is loaded. If a vtable pointer fails normal verification, it gets passed to the whitelist failure function. The function goes through the array, checking to see if the pointer is in any of the address ranges. If so, it assumes the pointer is valid and execution continues (verification succeeded). Otherwise, it reports an error and aborts.

Because the obfuscated libraries are dynamically loaded, the whitelist array records do not initially contain any addresses. If any records are empty when the whitelist failure function is called, then the function checks to see if the corresponding library has been loaded, and if so, it fills in the addresses before verification. For ChromeOS, our whitelist consists of the two third-party libraries mentioned above. This secondary verification, while not as accurate as normal VTV verification, still severely limits what an attacker can do, and with it we were able to execute all our tests on Chrome with no verification failures. Our secondary failure function only gets called in those cases where the main verification function fails. In that case it usually performs at most one alignment check and four pointer comparisons. Therefore, its overall impact on performance is small.

3.5 Alternatives & Enhancements for VTV

Since our performance overhead is reasonably good (ranging from 2.0% to 8.7% in the worst case, as we discuss in Section 7.1), we have not spent much time improving the performance of VTV. However, there are some things that could be done to improve these numbers. For various reasons, Chrome/ChromeOS currently cannot be compiled with devirtualization[1] enabled; we could enable devirtualization in Chrome and tune the devirtualizer to be more aggressive when combined with VTV. Partial inlining of the verification call sequences is another avenue we could explore, since call overhead accounts for a significant portion of our overall performance penalties. We could also implement secure methods for caching and reusing frequently verified values.

When we started implementing VTV, we decided that we did not want to modify any element of the toolchain except the compiler, especially because GCC can be used with a variety of different assemblers and linkers, and we did not want to modify all of them. An alternative approach would have been to have the compiler store the vtable-pointer sets as data in the assembly files. This data would be passed through the assembler to the linker. At link time the linker would see the whole program and could efficiently combine the vtable-pointer sets from the various object files into the appropriate final vtable-pointer sets. The dynamic loader, when loading the program, could load the pointers into our data sets and mark them read-only. This approach would eliminate ordering issues between functions that build vtable-pointer sets and functions that make virtual calls. The dynamic loader would also need to update the data, as appropriate, whenever it loaded a dynamic library that contained additional vtable pointers. A disadvantage of this alternative approach is that instead of requiring modifications only to the compiler, it would modify the entire toolchain: the compiler, the assembler, the linker, and the dynamic loader.

4 IFCC: Indirect Function-Call Checks

Indirect Function-Call Checks (IFCC) is a CFI transformation implemented over LLVM 3.4. It operates on LLVM IR during link-time optimization (LTO). IFCC does not depend on the details of C++ or other high-level languages; instead, it protects indirect calls by generating *jump tables* for indirect-call targets and adding code at indirect-call sites to transform function pointers, ensuring that they point to a jump-table entry. Any function pointer that does not point into the appropriate table is considered a CFI violation and will be forced into the right table by IFCC. IFCC collects function-pointers into jump tables based on function-pointer sets, like VTV's vtable-pointer sets, with one table per set.

IFCC forces all indirect-calls to go through its jump tables. This significantly reduces the set of possible indirect-call targets, and severely limits attacker options, preventing attacks that do not jump to a function entry point of the right type.

Each entry in a jump table consists solely of an aligned jump instruction to a function. The table is written to the read-only text area of the executable and is padded with trap instructions to a power-of-two size so that any aligned jump to the padding will cause the program to crash. Since the size of the table is a power of two, IFCC can compute a mask that can be used at call sites to force

[1]Devirtualization is an optimization that replaces virtual calls with a fast-path/slow-path mechanism: the fast path uses a direct call to the most common target, with a conditional check to make sure this is right; the slow path falls back on the normal virtual call mechanism. Devirtualization reduces the number of indirect calls and verifications and improves the overall performance of VTV.

a function pointer into the right function-pointer set. For example, if each jump-table entry takes up 8 bytes, and the table is 512 bytes in size, then there are 64 entries in the table, and the mask would be 111111000 in binary, which is 504 in decimal.

IFCC rewrites IR for functions that have their address taken; we call these *address-taken* functions. The main transformation replaces the address of each such function with the address of the corresponding entry in a jump table. Additionally, indirect calls are replaced with a sequence of instructions that use a mask and a base address to check the function pointer against the function-pointer set corresponding to the call site. The simplest transformation subtracts the base address from the pointer, masks the result, and adds the masked value back to the base. If the pointer was in the right function-pointer set before this transformation, then it remains unchanged. If not, then a call through the resulting pointer is still guaranteed to transfer control only to addresses in the function-pointer set or to trap instructions. Note that every pointer in a jump table is a valid function pointer (although some of them immediately hit trap instructions when they are called), so they can correctly be passed to external code.

IFCC can support many kinds of function-pointer sets, each with different levels of precision. For example, the most precise version would have one function-pointer set per function type signature. However, real world code does not always respect the type of function pointers, so this can fail for function-pointer casts. We will focus on two simple ways of constructing function-pointer sets: (1) *Single* puts all the functions into a single set; and (2) *Arity* assigns functions to a set according to the number of arguments passed to the indirect call at the call site (ignoring the return value).

Note that although we implemented only two simple types of tables, any disjoint partitioning of the function-pointer types in the program will work, as long as each call site can be uniquely associated with a single table. This is true no matter what analysis is used to generate these tables — be it static, dynamic, or manual. So, for example, the compiler could perform a detailed analysis of escaping pointers and use it to separate these pointers into their own tables.

The following example demonstrates the IFCC technique for a simple program. Consider a function `int f(char a) { return (int)a; }` and a main function that makes an indirect call to `f` through a function-pointer variable `g`. In LLVM IR, the symbol `@f` will refer to function defined above; IFCC adds a `@baseptr` symbol that stores a pointer to the first function pointer in the generated jump table. Before IFCC, the LLVM IR contains an indirect call instruction `%call = call i32 %2(i8 signext 0)` to the function pointer stored in variable `%2`.

IFCC generates a new symbol `@f_JT` and defines it in the IR as an external function. It finds each instance where the program uses the address of `@f` and makes it use the address of `@f_JT` instead. It also creates a jump table of the form:

```
  .align 8
  .globl f_JT
f_JT:
  jmp f
```

This defines the symbol `@f_JT` and satisfies the linker. IFCC instruments the code before the indirect call with instructions that transform the pointer. There are several ways to perform this transformation. We show two techniques, one that requires large alignments, and another for when large alignments are not supported. The requirement for large alignment in one scheme is because the base pointer must be aligned to the size of its table. This makes the base a prefix of each entry in its table.

When the object format and the kernel support large table alignments (e.g., greater than one page), IFCC can use a compact set of instructions to transform a pointer. The following IR assumes integer representations of `@baseptr` in `%1` and `@mask` in `%2`, and a pointer to `@f` in `%3`.

```
%4 = and i64 %2, %3
%5 = add i64 %1, %4
%6 = inttoptr i64 %5 to i32 (i8)*
%7 = call i32 %6(i8 signext 0)
```

In x86-64 assembly, this becomes:

```
and     $mask, %rax
add     $baseptr, %rax
callq   *%rax
```

The ELF format supports arbitrary alignments but the Linux kernel does not (as of version 3.2.5, under ASLR with Position-Independent Executables (PIE)). Under ASLR, the kernel treats the beginning of each ELF segment as an offset and generates a random base to add to the offset. The base is guaranteed to be aligned to a page boundary (2^{12}) but the resulting address is not guaranteed to have larger alignment.

Under these circumstances, IFCC changes the way it operates on function pointers; instead of adding a masked pointer to a base, it computes the difference between the base address and the function pointer, masks this value with the same mask as before, and adds the result to the base. This ends up generating 3 instructions for pointer manipulation in x86-64: a `sub`, then an `and`, then an `add`.

4.1 Practical Experience with IFCC

We modified the Chromium build scripts to build under LTO as much of the code as possible. It built 128 files as x86-64 ELF objects, and 11,012 files as LLVM IR. We then applied, separately, the Single and Arity versions of IFCC in this configuration.

Like VTV, IFCC suffers from false positives due to external code. In particular, any function that was not defined or declared during link-time optimization will trigger a CFI violation. This can happen for several reasons. First, JIT code (like JavaScript engines) can generate functions dynamically. Second, some external functions (like `dlsym` or `sigaction`) can return external function pointers. Finally, some functions can be passed to external libraries and can be called there with external function pointers; this is common in graphics libraries like `gtk`. The number of false positives varies greatly depending on the external code, ranging from extremely frequent in the case of JIT-generated code to extremely infrequent in the case of signal handlers.

To handle function pointers returned by external code, we added a fixed-size set of special functions to the beginning of each table. These functions perform indirect jumps through function pointers stored in an array. IFCC rewrites all calls to external functions (including `dlsym`) that return function pointers and inserts a function call that takes the pointer and writes it to the array if it is not already present. It returns a pointer to the table entry that corresponds to the array entry used to store the pointer. If the array has no more space, then it halts the program with an error. This converts function pointers from external code into table entries at the expense of adding a small number of writable function pointers to the code. This memory can be protected using the techniques described in Section 3.4, though the prototype in this paper does not perform this protection.

To handle functions passed to external functions, IFCC must find all cases in which functions are passed to external code and must rewrite the functions to not test their function pointers against the jump tables generated by IFCC. We added a flag to the IFCC plugin that takes the name of a function to skip in rewriting. To discover these function pointers, we added a warning mode to the IFCC transformation that prints at run time the names of functions that make indirect calls to functions outside the function-pointer sets. We found 255 such functions in Chromium, mostly associated with graphics libraries.

4.2 Annotations

The version of IFCC described in this section provides automatic methods for discovering and handling false positives. To improve maintainability of software with IFCC, however, we have implemented a different version that uses annotations instead of compile-time flags and uses custom failure functions like VTV. This is the version that we are working on upstreaming into LLVM.

Instead of functions being forced into the appropriate jump table, they are checked using the same code sequences as above, and any pointer that fails the check is passed to a custom failure function. IFCC's default failure function prints out the name of the function in which the failure occurred, and the value of the pointer that failed the check. This version of IFCC adds a comparison, a jump, and function call to the inserted instruction sequence. However, it gives greater flexibility to the resulting code in handling false positives, as discussed for VTV.

This new implementation provides annotations and a simple interprocedural dataflow analysis to help detect and handle these problems automatically. We provide two annotations that programmers can add using attribute notation: `__attribute__((annotate()))`.

- `cfi-maybe-external` is applied to local variables/parameters as well as to pointers in data structures.
- `cfi-no-rewrite` is applied to functions.

The dataflow analysis in IFCC finds external function pointers and traces their flow into indirect calls and into store instructions. It also traces the flow from `cfi-maybe-external`-annotated pointers and other variables into indirect calls and store instructions. It produces compile-time warnings if it finds a store instruction for an external function pointer and the pointer in the store instruction did not come from a location annotated by `cfi-maybe-external`. The annotations then can be used as a kind of whitelist in the CFI failure function, or these indirect calls can be skipped in rewriting.

The annotation `cfi-no-rewrite` means that all indirect calls in the annotated function might use external function pointers. The information from this annotation can be used either to build a whitelist or to skip rewriting. Our implementation currently skips rewriting for these indirect calls.

These annotations are also useful for cases that IFCC cannot detect, like callbacks buried deep inside data structures passed to external code. Calls to these functions will generate CFI violations at run time; these violations are false positives, and the locations of these indirect calls can be annotated with, e.g., `cfi-maybe-external` to indicate this to the CFI failure function.

It might seem like all an adversary has to do is to find one of the locations that has been annotated with `cfi-maybe-external` and overwrite a pointer that flows into it, and this will defeat IFCC. However, these annotations merely convey information to the CFI failure function; this function can perform arbitrarily complex checks make sure that function pointers that violate CFI are still valid. For the purposes of our evaluation, we implemented a simple failure function, as described above.

5 FSan: Indirect-Call-Check Analysis

The more precise the control-flow graph used in a CFI implementation, the harder it becomes for an attacker to exploit a program while staying within CFI-enforced bounds: a more precise CFG leads to fewer choices in targets for indirect control-transfer instructions. However, building a precise CFG is a hard problem, and programming techniques like function-pointer type punning and runtime polymorphism exacerbate this problem. Every time a CFI mechanism faces uncertainty, it is forced to be conservative to preserve correctness. Although this strategy guarantees correctness, it may result in a loss of security. Thus, techniques that reduce uncertainty about the CFG can increase security. We can achieve the best practical results by combining knowledge of programming language constructs (such as vtables), static analysis (such as we do for IFCC), and dynamic analysis. To that end, we designed FSan—an optional indirect call checker—and integrated it into Clang, LLVM's front end.

Kind	Offset	Data	Interpretation
Signature	0	0xeb	`jmp .+0x08/0x0c`
	1	0x06/0x0a	
	2	0x46	'F'
	3	0x54	'T'
RTTI	4	Pointer to `std::type_info` for the function's type (4/8 bytes)	

Table 1: Function prefix data layout for the optional function type checker.

Clang's undefined behavior sanitizer (UBSan) instruments C and C++ programs with checks to identify instances of undefined behavior. FSan was added to UBSan in LLVM 3.4. FSan detects CFI violations at runtime for indirect function calls.[2] FSan operates during the translation from the Clang AST to LLVM IR, so it has full access to type information, allowing it to make more accurate checks than IFCC, which uses IR alone.

FSan is a developer tool designed to perform optional type checking. In particular, it is not designed to defend against attacks. Instead, it is designed to be used by developers to identify CFI violations that may lead to security issues. As a fully accurate checker (it checks definedness exactly according to the definition in the C++ standard), it can also be used to help guide the development of control-flow integrity techniques by identifying properties of interest to be checked in the field.

FSan prefixes each function emitted by the compiler with 8 (on x86-32) or 12 (on x86-64) bytes of metadata. Table 1 shows the layout of these bytes; they are executable and cost little in performance, since the first two bytes encode a relative branch instruction which skips the rest of the metadata. The next two bytes encode the instructions `rex.RX push %rsp` (on x86-64) or `incl %esi ; pushl %esp` (on x86-32); this sequence of instructions is unlikely to appear at the start of a non-instrumented function body, and we observed no false positives in Chromium due to this choice of prefix.

Each indirect call site first loads the first four bytes from the function pointer, and compares it to the expected signature — the optionality of FSan arises from selecting a signature unlikely but permitted to appear at the start of an uninstrumented function. Because GCC at optimization level -O2 and higher and Clang at any optimization level will align functions to 16 bytes, this initial read succeeds for each function compiled with these compilers, regardless of the length of the function. This assumes GCC-compiled system libraries are compiled with -O2 or higher.

If the signature matches, then the next 4 or 8 bytes are loaded and compared against the expected function Run-Time Type Information (RTTI) pointer, which is simply the RTTI pointer for the function type of the callee expression used at the call site. If the pointers are unequal, then a runtime function is called to print an appropriate error message. A pointer equality test is sufficient because the function RTTI pointer for a particular function type is normally unique. This is because the linker will normally (but not always) coalesce RTTI objects for the same type, as they have the same mangled name.

The condition for undefined behavior as specified by the C++ standard is that the function types do not match (see C++11 [18, expr.reinterpret.cast], "The effect of calling a function through a pointer to a function type [...] that is not the same as the type used in the definition of the function is undefined"), so FSan is precise (no false positives or false negatives) with respect to this paragraph of the standard when both the caller and callee are compiled with the checker (provided that the linker coalesces RTTI symbols). However, FSan has not been implemented for C, and indeed would not work in its present form, mainly because the C rules relating to the definedness of calls to functions without a prototype are more complex.

Note that the RTTI pointer for a vtable function call is less precise than the vtable-pointer set check in VTV. FSan checks that each function has the correct type but not whether it was in the original program's CFG.

5.1 Practical Experience with FSan

We evaluated FSan by applying it to Chromium: we ran an instrumented version of the main browser executable, and FSan produced a variety of undefined-behavior reports. Two of the main categories of reports we observed were:

- Template functions whose parameters are of a templated pointer type, which are cast to functions whose parameters are of void pointer type so that they can be used as untyped callbacks;

[2]The undefined behavior sanitizer also includes a vtable-pointer checker which is not described here.

- Functions that take context parameters as typed pointers in the function declaration but void pointers or pointers to a base class at the call site.

The fix for these types of bugs is simple in principle: give the parameters a void pointer type and move the casts into the function body. One instance of each type of problem was found and fixed in the Skia graphics library that Chromium uses. This eliminated much of the low-hanging fruit reported by FSan; most of the remaining problems were more widespread in the codebase and thus will take more effort to deal with. For example, V8 uses callbacks to implement a feature known as "revivable objects" which Blink (née WebKit) relies on heavily; in many cases these callbacks were implemented using the derived types expected by the object implementation, rather than V8's base value type.

6 Security Analysis

In order to evaluate the efficacy of security techniques it is important to apply them to the real-world code they are expected to protect and measure the impact. To this end, we analyzed Chromium compiled with VTV and GCC, and with IFCC and Clang.

One consequence of implementing security mechanisms in the compiler is that it is important to evaluate the final output rather than simply the output of the particular compiler pass. The reasons for this are twofold: (1) later optimization passes may transform the security mechanism in unpredictable ways; and (2) the final linking step adds additional binary code to the final executable that the security pass never sees.

Basic optimizations such as common-subexpression elimination and loop-invariant code motion can eliminate redundant checks or hoist checks out of loops. Although such optimizations are generally acceptable from a correctness point of view, they may be impermissible from a security standpoint. For example, consider two consecutive calls to C++ member functions `obj.foo(); obj.bar();`. A security mechanism that protects virtual function calls, such as VTV, can load a vtable pointer into a callee-saved register, perform the verification, and then perform the two calls:

```
movq    (%r12), %rbx  ; set rbx to the vptr
movq    %rbx, %rdi
callq   verify_vtable ; verify_vtable(vptr)
movq    %r12, %rdi
callq   *16(%rbx)     ; obj.foo()
movq    %r12, %rdi
callq   *24(%rbx)     ; obj.bar()
```

This is perfectly correct behavior since the first function call is guaranteed by the platform ABI to preserve the value of the register. However, if `rbx` is spilled to the stack in `foo()` and is later overwritten, e.g., via a buffer overflow on the stack, then the call to `bar()` will be to an attacker-controlled location. An alternative to using a callee-saved register is to explicitly spill/reload the register to/from the stack, which has similar security concerns.

Since protecting the stack is outside the scope of this work, the compiler has significantly more freedom to eliminate this sort of redundant check.

Although it is difficult to meaningfully quantify the security provided by a mitigation measure, recent work by Zhang and Sekar [35, Definition 1] introduced the Average Indirect-target Reduction (AIR) metric

$$AIR = \frac{1}{n}\sum_{i=1}^{n}\left(1-\frac{T_i}{S}\right)$$

where n is the number of indirect control-transfer instructions (indirect calls, jumps, and returns), T_i is the number of instructions the ith indirect control transfer instruction could target after applying a CFI technique, and S is the size of the binary.

It's clear that for any reasonable CFI technique and a large binary, $T_i \ll S$ for all indirect control-transfer instructions transformed by the technique. Similarly, $T_i \approx S$ for all other indirect control-transfer instructions. So, AIR reduces to the fraction of indirect control transfer instructions that are protected by the technique.[3]

Since we are focused on protecting forward edges, we consider the related metric forward-edge AIR, or fAIR, which performs the same computation as AIR, but the average is taken only over the forward-edge indirect control transfer instructions: indirect calls and jumps.

To compute the statistics reported in the rest of this section, we modified LLVM's object-file disassembler to perform a hybrid recursive and linear scan through the Chromium binary, reconstructing functions and basic blocks on which we performed our analysis. This process was aided by ensuring that Chromium was compiled with debugging information including symbols (cf. Bao et al. [2]). This disassembler was used as part of a stand-alone tool to find all indirect control transfer instructions. For each such instruction, the tool walks backward through the CFG, looking for the specific protection mechanism. It also attempts to find constants which are inserted into registers used for the call or jump. See Table 2 for a break down of forward-edge indirect control transfer (fICT) instructions in Chromium.

VTV. Compiling a recent version of Chromium using GCC with vtable verification leads to a final binary containing 124,325 indirect calls and 18,453 indirect jumps for a total of 142,778 fICT instructions. Of these, 6,855 are neither constant nor protected by vtable verification, giving $fAIR_{\text{VTV}} = 95.2\%$. The majority of the unprotected

[3]This demonstrates that AIR — and our related fAIR — is at best a weak proxy for measuring security. Unfortunately, an actual metric for the security a CFI technique provides has thus far remained elusive.

fICT	VTV	IFCC
Constant	7,410	5,957
Constant, spilled†	7,334	315
Protected	113,617	154,244
Protected, spilled†	7,562	33,914
Unprotected	6,855	908
Total	142,778	195,338

Table 2: Forward-edge indirect control transfer (fICT) instructions in Chromium. Their arguments may be placed in three classes: (a) a type of constant, (b) an indirect address protected by CFI, and (c) an unprotected address. Constant-argument instructions include indirect jumps in the PLT which target a read-only GOT section and indirect jump instructions implementing switch statements, as well as indirect call instructions with constant targets.
† The targets for these indirect control transfer instructions are either spilled to the stack explicitly or are in callee-saved registers which are potentially spilled by intervening function calls.

fICTs come from C libraries, function-pointer adapter classes, and C-style callbacks.

Although more than 89% of the protected or constant fICTs are used almost immediately after being verified or loaded from read-only memory, in 14,896 instances (about 10%), the indirect target is potentially spilled to the stack. But this is not a flaw in our protection (see below).

IFCC. Compiling the same version of Chromium using Clang with IFCC (Single) produces a different binary containing 175,396 indirect calls and 19,942 indirect jumps for a total of 195,338 fICT instructions. Having more calls and jumps is what we would expect since the link-time optimizer has more inlining opportunities than is the case when optimizing one translation unit at a time.[4]

Since IFCC is designed to protect all fICT instructions, not just C++ virtual member function calls, only 908 fICT instructions are left unprotected. This gives $fAIR_{\text{IFCC}} = 99.5\%$. In fact, this is an over estimate of the number of unprotected fICT instructions. Of the 908 unprotected instructions, 512 correspond to the special functions created for function pointers returned from non-instrumented functions. Discounting those gives the more accurate value of $fAIR_{\text{IFCC}} = 99.8\%$. Most of the remaining unprotected fICT instructions correspond to functions which are explicitly not instrumented. (See Section 4.1 for a discussion of both of these.) The remaining handful come from the C run time statically linked into every binary.

With IFCC, about 18% of the constant or protected fICTs have targets which are potentially spilled to the stack. However this is not a fatal flaw, as discussed immediately below, since we are assuming that the stack is protected by some other means.

Stack spilling implications. For the purpose of our techniques, spilling target values to the stack introduces no additional security risk, since an attacker who can overwrite one value on the stack can easily overwrite a saved return address. This does have serious implications for CFI schemes that attempt to protect backward edges.

Our experience shows the importance of verifying a protection mechanism's intended invariants on the final binary output after all optimizations, including architecture-dependent optimization in the compiler backend, have taken place and the language runtime has been linked in.

Counting ROP gadgets. It is common in CFI papers to count the number of return-oriented programming gadgets that remain after applying the protection mechanism. Since we are explicitly not protecting return instructions, it does not make sense to count gadgets.

7 Performance Measurements and Results

We measured the performance of our approaches both on the C++ tests from the SPEC CPU2006 benchmark suite and on the Chromium browser running Dromaeo, SunSpider, and Octane. Except where otherwise specified, the VTV tests were run all on an HP Z620 Xeon E52690 2.9GHz machine, running Ubuntu Linux 12.04.2, and the IFCC and FSan tests were run on an HP Z620 Xeon E5550 2.67GHz machine, running the same OS. We turned off turbo mode and ASLR on these machines, as doing so significantly reduced the variation in our results.

The Chromium web browser is a large, complex, real-world application, comprising over 15 million lines of C++ code in over 50,000 source files, and containing hundreds of thousands of virtual calls. It links in many third-party libraries and makes extensive use of dynamic library loading. It is also representative of the type of target attackers are interested in. For all these reasons, Chromium makes an excellent test for measuring the effects of our CFI approaches on real-world systems. Both VTV and IFCC were able to successfully build fully-functional, protected versions of Chromium.

7.1 VTV Performance

Since verification adds instructions to the execution, some performance penalty is unavoidable. Initially, we ran SPEC CPU2006 C++ benchmarks with and without VTV to get a baseline. Table 3 shows that omnetpp, astar, and xalancbmk suffer a noticeable performance penalty in this naive implementation, ranging from 2.4% to 19.2%. We improve on this later. The other four benchmarks (povray, namd, soplex, and dealII) showed no significant performance effects. To determine why, we collected statistics on those benchmarks, for both compile-time

[4]There is a corresponding decrease in the number of return instructions for the same reason.

Test	no VTV (seconds)	VTV (seconds)	% slowdown
omnetpp	320.70	346.42	8.0
astar	440.61	450.95	2.4
xalanc.	248.86	296.53	19.2
namd	445.19	445.32	*
dealII	344.89	348.39	*
soplex	235.20	236.46	*
povray	181.18	181.87	*

Table 3: Untuned SPEC run-time numbers, at -O2. The asterisks indicate changes that are too small to be of any significance. These numbers are the minimum out of three runs (standard deviation is very close to zero).

Test	virtual calls (static)	verified calls (dynamic)	run time (secs)	verified calls per second
namd	2	0	445.32	0
dealII	2,118	201,867,094	348.39	579,428
soplex	720	4,846,399	236.46	20,496
povray	159	159,186	181.87	875
omnetpp	1,312	1,029,110,532	346.42	2,970,702
astar	2	2,780,359,179	450.95	6,165,560
xalanc.	15,753	2,629,817,426	296.53	8,868,639
dromaeo	NA	6,705,708,649	2379.34	2,818,303
octane	NA	113,037,194	66.41	1,702,214
sunspi.	NA	27,068,246	16.36	1,654,943

Table 4: Verifications per second when running SPEC CPU2006 C++ benchmarks and Chrome with VTV.

(static) and run-time (dynamic) numbers of verified virtual calls. From this we calculated the number of verified calls per second. As shown at the top of Table 4, those tests do not perform enough calls per second to noticeably affect performance, so we did not include them in any further analyses of VTV.

Next, we looked at reducing VTV's performance penalty. To determine the minimum lower bound penalty, we considered two sources of performance overhead: (1) the cost of making the verification function calls; and (2) the (potential) cost due to the overall increase in code size (code bloat). We measured these by doing two experiments on the three SPEC benchmarks of interest. First, we replaced the bodies of the functions in libvtv with stubs. Second, we inserted an unreachable region of code preceded by an unconditional jump over the region just before each virtual call instruction (the unreachable region represented the code that would be inserted by VTV). Our results can be seen in Table 5. Note that making calls with stubs must increase the number of instructions, just as with the code bloat test. Therefore the stubs penalty automatically includes the code bloat penalty. This shows that even if we could reduce to zero the time spent executing inside the verification function, the minimum lower bound VTV penalty for these tests ranges from 1.0% to 4.7%. Note that the lower bound is test-specific and depends on the number of virtual calls a test executes.

	Code Bloat % Slowdown	VTV Stubs % Slowdown
omnetpp	0.0	2.4
astar	0.2	1.0
xalancbmk	0.8	4.7

Table 5: Results of lower bound experiments for VTV.

We then tried various options to reduce VTV's performance penalties. The two most effective options were: using profile guided optimizations (PGO) to improve devirtualization, via GCC's -ripa flag, thus reducing the overall number of virtual calls; and statically linking libvtv itself, to reduce the level of indirection at each verification call. We re-ran the SPEC benchmarks using these various options, and the results are shown at the top of Figure 2. The xalancbmk test had the worst performance with VTV, so it is instructive to consider its results under optimization: devirtualization brought the performance penalty from 19.2% down to 10.8%, and static linking reduced it further to 8.7%. The lower bound of VTV is 4.7% for xalancbmk (see Table 5).

Chrome interacts with many system libraries, so to avoid the problems of mixing verified and unverified code with VTV we built and ran a verified version of Chrome in a verified version of ChromeOS on a Chromebook (thus building all the libraries with verification as well). We built ChromeOS version 28 with VTV, and ran the Dromaeo, SunSpider 1.0.2, and Octane 2.0 benchmarks with it. For these tests, we loaded our images onto a Chromebook with an Intel Celeron 867 chip, pinned at 1.3GHz, with ASLR turned off, and we ran the tests there, with and without VTV. The bottom of Figure 2 shows the performance costs. We were not able to build Chrome with PGO and devirtualization, nor with the statically linked libvtv, so for these measurements we only have the naive, untuned VTV numbers. For Octane we saw a 2.6% penalty with VTV. SunSpider had a 1.6% penalty. Dromaeo had a fair amount of variation across the full set of micro-benchmarks, but the overall performance penalty across all of them was 8.4%. We expect that adding devirtualization would significantly improve these numbers. As with the SPEC benchmarks, the performance penalty varies depending on the number of verified calls made at runtime. We measured this for each of these tests (see Table 4). As expected, Dromaeo, which had the largest penalty, makes significantly more verified calls/second than the other two.

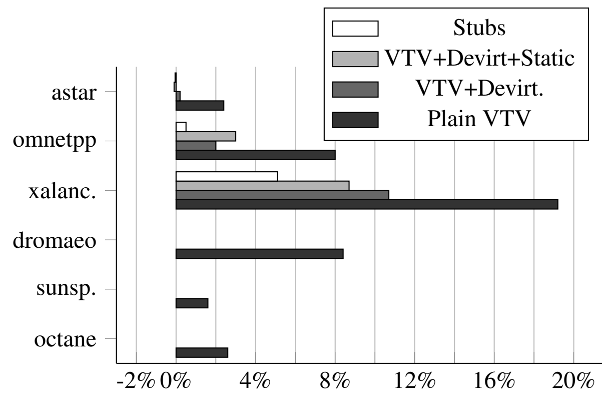


Figure 2: Relative performance overhead of VTV, with various tuning options, for the SPEC 2006 C++ benchmarks and for Chrome browser.

7.2 IFCC and FSan Performance

The performance of code compiled under IFCC depends on how often it makes indirect calls. IFCC adds code to every indirect call site that is not explicitly skipped by a command-line directive. The amount of code it adds depends on the version of IFCC: if tables are small enough to fit in a page in memory, then it can use the transformation that adds only two instructions (comprising 14 bytes) to each site. Otherwise, it uses the subtraction version, which adds 4 instructions (which become 20 bytes). Each indirect call has additional extra overhead from the `jmp` in the indirect call table(s); and jumps through a table might have effects on instruction-cache usage. Finally, when rewritten code receives a pointer from `dlsym` or from a dynamically-linked library, this pointer is wrapped using linear search through a fixed-length array; this is a slow operation but should not happen often.

The exact instructions added by FSan depend on the specific optimization level used, but we found that it usually adds about 12 instructions to each call site.

Figure 3 shows results from running C++ programs from the SPEC benchmark suite under IFCC and FSan and provides relative performance overhead compared to an optimized version compiled using Clang; each running time is the minimum of 10 executions. As expected, LTO outperforms both IFCC transformations in most cases. This is because IFCC adds instructions to the base LTO-compiled binary, and these instructions reduce performance of the executable. The cases in which IFCC outperforms LTO involve only small differences in performance and are likely due to effects similar to the noise discussed by Mytkowicz et al. [24], so we do not analyze them further here.

We ran the Dromaeo benchmark on Chromium 31.0.1650.41 built with Clang LTO, a version built with IFCC Single, and a version built with IFCC Arity. Single got 96.6% of the LTO score, and Arity got 96.1%; higher is better in Dromaeo, so this is about a 4% overhead, as shown in Figure 3a. We also built a version of Chromium and the SPEC CPU2006 benchmarks with the annotation version of IFCC, and we saw similar results.

IFCC had nearly the same performance as LTO for both Single and Arity versions of the SPEC CPU2006 benchmarks, except for xalancbmk. FSan had effects similar to IFCC. The xalancbmk benchmark suffers the most performance degradation from IFCC; this is expected due to it having the most dynamic virtual calls, as shown in Table 4. Similarly, Dromaeo has a large number of virtual calls and has the second highest overhead. So, as with VTV, the performance overhead is directly related to the number of indirect calls.

7.3 Comparison to Prior Work

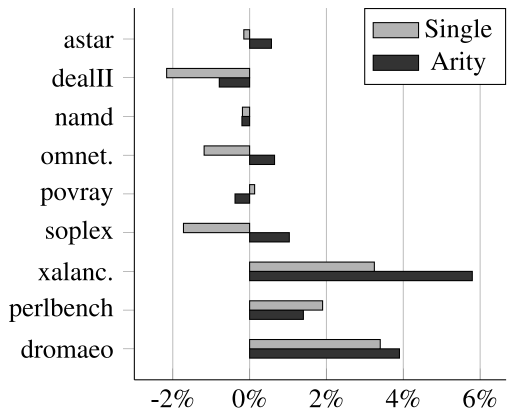
The SPEC Perl benchmark is worth highlighting. As Niu and Tan [28] point out, Perl is, in some sense, a worst case for CFI techniques for C — whereas C++ code can be even worse. Perl operates by translating the source code into bytecode, then sits in a tight loop, interpreting each instruction by making an indirect call. This worst case behavior is apparent in the performance of four recent CFI implementations: CCFIR by Zhang et al. [34], bin-CFI by Zhang and Sekar [35], Strato by Zeng et al. [33], and MIP by Niu and Tan [28]. The overheads reported for CCFIR, bin-CFI, Strato, and MIP are 8.6%, 12%, 15%–25%, and 14.9%–31.3%, respectively.

In contrast, our own work has less than 2% overhead (see Figure 3a). We are able to achieve this significant speed up over prior work by focusing only on forward edges as well as leveraging the compiler to apply optimizations. This gives different security guarantees, but we believe our attack model comports well with reality.

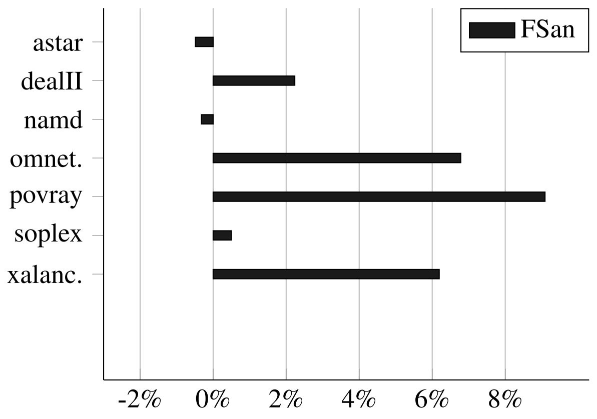
8 Conclusions

This paper advances the techniques of Control-Flow Integrity, moving them from research prototypes to being firmly in the domain of the practical. We have described two different principled, compiler-based CFI solutions for enforcing control-flow integrity for indirect jumps: vtable verification for virtual calls (VTV) guarantees that the vtable being used for a virtual call is not only a valid vtable for the program but is semantically correct for the call site; and indirect function-call checking (IFCC) guarantees that the target of an indirect call is one of the address-taken functions in the program. We also present FSan, an optional indirect call checking tool which verifies at runtime that the target of an indirect call has the correct function signature, based on the call site.

We have demonstrated that each of these approaches is feasible by implementing each one in a production compiler (GCC or LLVM). We have shown via security analysis that VTV and IFCC both maintain a very high level



(a) Relative overhead of IFCC enforcement (baseline LLVM LTO) for SPEC CPU2006 benchmarks and the Dromaeo benchmark.



(b) Relative overhead of the FSan optional indirect-call checking (baseline Clang) for the C++ benchmarks in SPEC CPU2006.

Figure 3: Performance measurements for IFCC and FSan.

of security, with VTV protecting 95.2% of all possible indirect jumps in our test, and IFCC protecting 99.8%. We have also measured the performance of these approaches and shown that while there is some degradation, averaging in the range of 1%–4%, and in the worst case getting up to 8.7% for VTV (the most precise approach), this penalty is fairly low and seems well within the range of what is acceptable, particularly in exchange for increased security.

Due to our relaxed, yet realistic, attack model coupled with compiler optimizations, we achieve significant performance gains over other CFI implementations while defending against real attacks.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Trans. Info. & System Security*, 13(1):4:1–4:40, Oct. 2009.

[2] T. Bao, J. Burket, and M. Woo. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[3] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of ISSTA 2012*, July 2012.

[4] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[5] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of SOSP 2009*, Oct. 2009.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of CCS 2010*, pages 559–572. ACM Press, Oct. 2010. URL `https://cs.jhu.edu/~s/papers/noret_ccs2010.html`.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of USENIX Security 1998*, Jan. 1998.

[8] "d0c_s4vage". Insecticides don't kill bugs, Patch Tuesdays do. Online: `http://d0cs4vage.blogspot.com/2011/06/insecticides-dont-kill-bugs-patch.html`, June 2013.

[9] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of NDSS 2012*, Feb. 2012.

[10] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[11] A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, Apr. 2001.

[12] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address

spaces. In *Proceedings of OSDI 2006*, pages 75–88, Nov. 2006.

[13] Ú. Erlingsson, Y. Younan, and F. Piessens. Low-level software security by example. In P. Stavroulakis and M. Stamp, editors, *Handbook of Information and Communication Security*, pages 633–658. Springer Berlin Heidelberg, 2010.

[14] C. Evans. Exploiting 64-bit linux like a boss. Online: `http://scarybeastsecurity.blogspot.com/search?q=Exploiting+64-bit+linux`, 2013.

[15] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, May 2014.

[16] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of USENIX Security 2014*, Aug. 2014.

[17] Google Developers. Native client. Online: `https://developers.google.com/native-client/`, 2013.

[18] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[19] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of NDSS 2014*. Internet Society, Feb. 2014. To appear.

[20] K. Kortchinsky. 10 years later, which vulnerabilities still matter? Online: `http://ensiwiki.ensimag.fr/images/e/e8/GreHack-2012-talk-Kostya_Kortchinsky_Crypt0ad_-10_years_later_which_in_memory_vulnerabilities_still_matter.pdf`, 2012.

[21] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: better, faster, stronger SFI for the x86. In *Proceedings of PLDI 2012*, pages 395–404, June 2012.

[22] Mozilla Foundation. Mozilla Foundation security advisory 2013-29. Online: `https://www.mozilla.org/security/announce/2013/mfsa2013-29.html`, 2013.

[23] MWR InfoSecurity. Pwn2Own at CanSecWest 2013. Online: `https://labs.mwrinfosecurity.com/blog/2013/03/06/pwn2own-at-cansecwest-2013`, 2013.

[24] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of ASPLOS 2009*, Mar. 2009.

[25] NIST. CVE-2010-0249. Online: `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0249`, 2010.

[26] NIST. CVE-2010-3971. Online: `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3971`, 2010.

[27] NIST. CVE-2011-1255. Online: `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1255`, 2011.

[28] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of CCS 2013*, Nov. 2013.

[29] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *Proceedings of ACSAC 2013*, Dec. 2013.

[30] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security*, 15(1), Mar. 2012.

[31] Z. Wang and X. Jiang. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of IEEE Symposium on Security and Privacy ("Oakland") 2011*, May 2011.

[32] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of CCS 2011*, Oct. 2011.

[33] B. Zeng, G. Tan, and Ú. Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *Proceedings of USENIX Security 2013*, Aug. 2013.

[34] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, L. Szekeres, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, May 2013.

[35] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of USENIX Security 2013*, Aug. 2013.