Compile-Time Function Call Interception to Mock Functions in C/C++

Gábor Márton   Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics
Dept. of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C Budapest, Hungary
martongabesz@gmail.com   gsd@elte.hu

Abstract

In C/C++, test code is often interwoven with the production code we want to test. During the test development process we often have to modify the public interface of a class to replace existing dependencies; e.g. a supplementary setter or constructor function is added for dependency injection. In many cases, extra template parameters are used for the same purpose. These solutions may have serious detrimental effects on the code structure and sometimes on the run-time performance as well. We introduce a new technique that makes dependency replacement possible without the modification of the production code, thus it provides an alternative way to add unit tests. Our new compile-time instrumentation technique enables us to intercept function calls and replace them in runtime. Contrary to existing function call interception (FCI) methods, we instrument the call expression instead of the callee, thus we can avoid the modification and recompilation of the function in order to intercept the call. This has a clear advantage in case of system libraries and third party shared libraries, thus it provides an alternative way to automatize tests for legacy software. We created a prototype implementation based on the LLVM compiler infrastructure which is publicly available for testing.

Keywords C++ programming language, unit testing, function call interception, compiler instrumentation

1. Introduction

In legacy code bases often there are few or no unit tests. Refactoring such code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle. We can break the circle with non-intrusive tests, i.e. without actually modifying the production code. Function call interception (FCI) is often the only tool which enables non-intrusive testing by making it possible to replace function bodies. By replacing functions we can eliminate the unwanted dependencies in tests.

Eliminate unwanted dependencies is necessary during testing of new software systems as well. There may be cases when we do not want to widen the public interface of a class or when we do not want to promote the concrete type to a class template just because of testing.

With FCI we are able to intercept function calls at runtime and we can execute actions before and/or after the original function body or even completely replace it. The different FCI methods have different advantages and disadvantages. Dynamic methods are not always applicable (think about intercepting inline functions of C++ member functions). In the same time some dynamic solutions have minimal runtime overhead. Static methods vary from preprocessor techniques to modifying the generated binary. Working before runtime, they have much more freedom for intercession.

Compared to languages like Java, the C and C++ languages offer less mature solutions for FCI. Java runtime reflection allows us both introspection and intercession. Aspect-oriented programming defines non-intrusive joint points (which could be function call sites), advices (like before, after and around) and pointcuts (predicates matching the joint points) grant us to replace functions [23]. Early attempts have been made to implement aspect-oriented frameworks for C++ [8, 50], but they are less popular.

In this paper, we investigate a new compile-time instrumentation based FCI technique for C/C++ programs which enables the replacement of functions and methods. Our approach complements the existing FCI methods, with it we can synthesise tests for C/C++ code in a non-intrusive way. By applying the instrumentation, the generated binary code will be different than the original binary program code, but the high-level C/C++ source code remains untouched. Contrary to other instrumentation methods, we instrument the call expression instead of the callee, thus we can avoid the necessity of recompilation of the function we would like to intercept. This has a clear advantage in case of system libraries, third party shared libraries and security critical applications where we have to evade library interposing. We implemented a prototype based on the LLVM/Clang compiler infrastructure. We demonstrate how the use of our method simplifies dependency replacement for the use-case of writing unit tests for legacy systems. We also evaluate the performance of the prototype by using various benchmarks.

This paper is organized as follows. In Section 2, we show the existing dynamic and static FCI methods. In Section 3, we present general test automation patterns and concepts for testing legacy code. We present how our method simplifies writing unit tests for legacy systems in Section 4. We describe our new interception technique in details in Section 5 in details. Section 6 contains the evaluation of the performance of our prototype. In Section 7, we describe the current limitations and possible future work. We have an overview of the related works in Section 8. Our paper concludes in Section 9.

[Copyright notice will appear here once 'preprint' option is removed.]

2. Function Call Interception Techniques

We differentiate the FCI techniques based on the time FCI is applied [22]. *Dynamic techniques* perform the interception at *program load-time* or at *runtime*. Contrary to dynamic approaches, *static techniques* achieve FCI by modifying the *source files* (e.g. with the help of the preprocessor), by changing the *linkage order*, by generating object files which contains the *instrumentation* or by modifying the application *binary image*; all these modifications happen before runtime.

2.1 Load-time FCI

Most modern operating systems provide the possibility to specify shared objects to be loaded before all others. This can be used to selectively override functions in other shared objects. On Linux this behavior is controlled by the `LD_PRELOAD` environment variable [33]. With this technique, calling the original function is cumbersome. We have to use `dlsym` auxiliary function with the `RTLD_NEXT` argument [32]. In case of C++ functions we have to provide the mangled names. Furthermore, this mechanism is unreliable with member functions, because the member function pointer is not expected to have the same size as a void pointer on some platforms [21].

2.2 Run-time FCI

In Unix like systems, runtime dynamic interception is implemented with the help of the `ptrace` system call [35, 44, 45]. If ptrace is used with the `PEEKTEXT` or `POKETEXT` argument then it is possible to attach to a running process and to read or write different segments of its memory. For instance, the GNU debugger (gdb) [16] and Intel Pin [31] both use this approach. A disadvantage of these tools is that they rely on a specific kernel functionality; thus porting these implementations to other operating systems may be hard. E.g. Intel Pin currently does not support function replacement on macOS [20]. Another property of this technique is that we cannot instrument inline functions.

2.3 Pre-compilation-time FCI

We consider some use of the C/C++ preprocessor as pre-compilation-time interception. A typical use case is to replace the `malloc` and the `free` functions from the standard C library [47]:

```
void *my_malloc(size_t size) {
  //...
  return malloc(size);
}

void my_free(void *p) {
  //...
  return free(p);
}

#define free my_free
#define malloc my_malloc

void SystemUnderTest() {
  int *array = (int *)malloc(4 * sizeof(int));
  // do something with array
  free(array);
}
```

This approach can be applied conveniently in C, but not in C++. As soon as we use namespaces, the preprocessor might generate code which cannot be compiled because of the ambiguous use of names. Hazardous side effects of macros are also well known.

2.4 Link-time FCI

One example for the link-time static interception is the `wrap` command line option of the GNU linker (`ld`) [15]. When this program option is applied then the linker uses a wrapper function for the specified `symbol`, any undefined reference to `symbol` will be resolved to `__wrap_symbol` and any undefined reference to `__real_symbol` will be resolved to `symbol`. This approach makes it possible to replace a function and call the original. However, in case of C++ we have to specify the mangled names as symbols. We cannot use this approach if the `symbol` is defined within the very same translation unit where it is referenced.

2.5 Post-compilation-time FCI

There exist tools to modify the compiled binary code for interception. As an example, in [3] the authors describe a method which is a mixture of Link-time and Post-compilation-time techniques used to avoid typical security vulnerabilities, like buffer overflow. A modified compiler can be applied on a binary executable (or shared library) to extract type information from the debugging data and reinsert it in the same binary which is then available at runtime in a special data structure. At runtime a pre-loaded shared library intercepts the possibly dangerous calls and validates them using the data structure stored in the first step.

2.6 Compile-time FCI

Perhaps the most widely used static FCI technique is to configure the compiler to emit instrumented code in a way that interception is possible. The GNU/GCC and LLVM/Clang compilers both provides the `-finstrument-function` program option to instrument each and every function call in a way to execute code before and after the body of the functions [9, 14]. Actually, when this instrumentation is enabled then the compiler emits two extra calls for each function body. The prototypes of these two called functions are the following:

```
void __cyg_profile_func_enter(void *this_fn,
                              void *call_site);
void __cyg_profile_func_exit(void *this_fn,
                             void *call_site);
```

The arguments for these functions represent the address of the original function and the address of the instruction from where it was called. A serious drawback of this technique is that we cannot replace an intercepted function with another function; the original function will be called anyway.

2.7 Hybrid FCI techniques

XRay [4, 26], a function call tracing system allows engineers to get accurate function call traces with negligible overhead when off and moderate overhead when on, suitable for services deployed in production. XRay enables efficient function call entry/exit logging with high accuracy timestamps, and can be dynamically enabled and disabled.

XRay uses a mixture of compile-time and run-time FCI techniques. It relies on compiler changes to insert no-op sleds in function entry/exits, and record those locations in tables encoded in the object files. At runtime, if XRay is disabled, these no-op sleds are executed as-is and add minimal execution overhead. However, if XRay is enabled, the XRay runtime library overwrites these no-ops with calls to instrumentation code that logs function entry/exit information to in-memory buffers. XRay is a special purpose instrumentation, it is not a general FCI framework; therefore is it not possible to replace functions with it.

3. Test Automation Conventions

The above presented FCI techniques maybe used in the process of creating automated tests. Thus, in this section we discuss the general test automation patterns and we show the more specialized concepts about testing legacy code. Note, the presented general

test automation patterns maybe used during writing tests for legacy code.

3.1 General Test Automation Patterns

The *four-phase* test pattern is driven by the observation that each test require some sort of setup and tear down routines. This pattern splits each test into four phases [39]:

- In the first phase, we set up everything that is required for the system under test (SUT) to exhibit the expected behavior.
- In the second phase, we interact with the SUT.
- In the third phase, we do whatever is necessary to determine whether the expected outcome has been obtained.
- In the fourth phase, we tear down the test to put the world back into the state in which we found it.

This pattern is also know as the build-operate-check-clear pattern [51].

The *given-when-then* pattern of representing tests is originated from behaviour-driven development [43]. The essential idea is to break down writing a test into three sections [11]:

- The *given* part describes the pre-conditions to the test. In these pre-conditions we present the state of the world before we begin the behavior we specify in the test.
- The *when* section represents the behaviour we specify.
- The *then* section describes the changes we expect due to the specified behavior.

We can also look at this pattern as a reformulation of the four-phase test pattern. Essentially these three states are equal to the first three states of the four-phase pattern.

In the context of the four-phase pattern, Robert C. Martin states that anyone who reads the tests should be able to work out what they do very quickly, without being misled or overwhelmed by details [36]. Consequently, both the four-phase and the given-when-then patterns imply that the test setup should be strictly part of the visible test code and should not be separated from the rest of the test code. For instance, using load-time FCI to set up a test separates the "given" phase from the rest of the test code, thus it violates both patterns and makes the test hard to understand.

3.2 Testing Legacy Code

Unwanted dependencies embody a critical problem in software development; we often have to break existing dependencies before we can change some piece of code [47]. Breaking existing dependencies is also an important prerequisite to introduce unit tests for legacy code [10].

A *seam* is an abstract concept introduced by Feathers to identify points where we can break dependencies [10]. The goal is to have a place where we can alter the behaviour of a program without modifying it in that place; this is important because editing the source code is often not an option [47]. Feathers, Rüegg and Sommerlad define four different kinds of seams for C++ [10, 40, 47]:

1. Link seam: Change the definition of a function via some linker specific setup.
2. Preprocessor seam: With the help of the preprocessor, redefine function names to use an alternative implementation.
3. Object seam: Based on inheritance to inject a subclass with an alternative implementation.
4. Compile seam: Inject dependencies at compile-time through template parameters.

```
// Turtle.hpp

class Turtle {
    int x = 0, y = 0;
public:
    void PenUp() { /* ... */ }
    void PenDown() { /* ... */ }
    void Forward(int distance) { /* ... */ }
    void Turn(int degrees) { /* ... */ }
    void GoTo(int x, int y) { /* ... */ }
    int GetX() const { return x; }
    int GetY() const { return y; }
};

class Painter {
    Turtle turtle;
public:
    void DrawLine(int x0, int y0, int x1, int y1) {
        turtle.GoTo(x0, y0);
        turtle.PenDown();
        turtle.GoTo(x1, y1);
        turtle.PenUp();
    }
    // ...
};
```

Figure 1. A hypothetical legacy graphics program

The *enabling point* of a seam is the place where we can make the decision to use one behaviour or another. Different seams have different enabling points. For example, replacing the constructor argument for the implementation of an interface with a mock implementation when a unit test is set up is an object seam with the constructor as an enabling point.

Link and preprocessor seams can be used to write non-intrusive tests. However, object and compile seams may be used for such purpose only if the unit under test already has the proper architecture. For example, in case of object seams the unit must have a constructor (or setter) function to setup a different implementation for the dependency. In case of compile seams, the unit must be a template and it must have a template parameter via which we can mock the dependency. Often, these architectural requirements are not satisfied, therefore the use of object and compile seams ofttimes demand that we intrusively change the source code of the unit.

Some seams are realized with FCI techniques. For instance, preprocessor seams are implemented with pre-compilation-time FCI. Link seams are realized with load-time and link-time FCI. The existence of compile-time, post-compile-time and run-time FCI drives us to further extend the list of existing seams. We define a new class of seams, the *FCI seams*. More specifically we introduce three new seams for each FCI technique: *compile-time FCI seam*, *post-compile-time FCI seam* and *run-time FCI seam*.

4. Compile-time FCI Seam

In Figure 1 we present a hypothetical legacy graphics program that relies on a LOGO-like API for drawing. The API is realized as a class named the `Turtle`. Also, there is `Painter` class which is responsible for drawing lines and shapes. This class has a hard-wired dependency on the concrete `Turtle` class. Still, we would like to write a test which checks the `DrawLine()` function. In this hypothetical example let us suppose that the turtle functions are quite expensive to use. Generally speaking, a dependency may represent a database, or a network connection, whose usage can be hard, or very expensive. Therefore, in our test we want to mock the `Turtle` class (or at least its member functions).

Our new instrumentation technique makes it possible to write non-intrusive tests easily. Figure 2 lists the test which uses our new instrumentation method. We define our mock class (`MockTurtle`)

```
1
2   #include "Turtle.hpp"
3   #include <gmock/gmock.h>
4   #include <access_private.hpp>
5   #include <hook.hpp> // for SUBSTITUTE
6
7   class MockTurtle {
8   public:
9     MOCK_METHOD0(PenUp, void());
10    // PenDown, Forward, ...
11  };
12
13  MockTurtle& GetMockObject(Turtle*) {
14      static MockTurtle m;
15      return m;
16  }
17
18  namespace proxy {
19      void PenUp(Turtle* self) {
20          return GetMockObject(self).PenUp();
21      }
22      // Similarly to PenDown, Forward, ...
23  }
24
25  struct TurtleTest : ::testing::Test {
26      TurtleTest() {
27          SUBSTITUTE(Turtle::PenUp, proxy::PenUp);
28          // Similarly to PenDown, Forward, ...
29      }
30  };
31
32  ACCESS_PRIVATE_FIELD(Painter, Turtle, turtle)
33
34  TEST_F(TurtleTest, TestDrawLine) {
35      using ::testing::AtLeast;
36
37      Painter painter;
38      Turtle& turtle = access_private::turtle(painter);
39      MockTurtle& mockTurtle = GetMockObject(&turtle);
40
41      EXPECT_CALL(mockTurtle, PenDown())
42          .Times(AtLeast(1));
43      painter.DrawLine(0, 0, 10, 10);
44  }
45
46  int main(int argc, char **argv) {
47      ::testing::InitGoogleTest(&argc, argv);
48      return RUN_ALL_TESTS();
49  }
```

Figure 2. Testing the legacy graphics program with compile-time FCI

with the help of the gmock macros (lines 7-11). Our test-case is defined from line 34 to 44. In the test-case we create an instance of the `Painter` class, then we get a reference to its private `turtle` member (line 38). Note that there are several different techniques to access a private member, we use a method which relies on explicit template instantiations [37]. Then we get a reference to an instance of the `MockTurtle` class which acts as a test double for the `Turtle` instance (line 39). We state our expectations as we would do with any other regular mock objects (lines 41-42). In line 43 we exercise our unit under test by calling the `DrawLine()` method.

With the help of our tool we setup replacement functions for each member function of the `Turtle` class (lines 27-28). These replacement functions behave as a proxy; they forward each function call on a given `Turtle` instance to a corresponding test double (lines 18-23). The way we get the reference for a relevant test double is pretty simple in this test: we return a reference to a static instance of the `MockTurtle` class (lines 13-16). We can use this simplification because we know that there is only one `Turtle` object over the lifetime of our test-case. If there were several `Turtle` objects then we should solve the mapping differently, perhaps with the help of a static hash map. Lines 46-49 contains the definition for the `main()` function which uses the functions and macros from googletest to initialize and run the test.

The most important property of this test is that the test setup is included in the test application itself. During the compilation of our test binary we have to include a header file from our auxiliary runtime library which provides the `SUBSTITUTE` macro, and we have to enable the mentioned instrumentation with a compiler switch. Also, during linking we have to link with our given runtime library.

Our method has clear advantages compared to the `LD_PRELOAD` approach where we can substitute functions only if they are defined in shared libraries. With our technique it is possible to write non-intrusive tests and replace even inline functions. However, this new method requires rebuilding the application (or unit) we want to test with the specific compiler option which will disable inlining. Our technique has the following advantages:

- The test setup is part of the test application and clearly visible together with the rest of the test code. This way it does not violate the given-when-then test automation pattern and best practices.
- It does not introduce a new tool into the existing build chain. The functionality is embedded into the compiler.
- On platforms where the compiler is supported, the new instrumentation could be supported as well.
- There is no need to use mangled names.
- We can use the ordinary unit test building tools and we can group unit tests into the same test application.

5. FCI with Call Expression Instrumentation

Our new interception technique and the prototype [1] consists of two parts: a compiler instrumentation module and a runtime library. The instrumentation module modifies the code to check whether a function has to be replaced or not. The runtime library provides functions to setup the replacements.

5.1 Instrumentation

During the code generation we modify each and every function call expression to call an auxiliary function. Let us consider the following function call expression of `foo`:

```
foo(args...);
```

When our instrumentation is in action, the emitted code is equal to the following pseudo code:

```
char* funptr = __fake_hook(&foo);
if (funptr) {
    funptr(args...);
} else {
    foo(args...);
}
```

The call to `__fake_hook` resolves at runtime if we should replace the callee with another function or not. We replace a function if the returned value of `__fake_hook` is not zero, in this case the returned value is a pointer to the function we call as a substitution. If the return type of the callee function is not `void` then we create an additional storage for the return value:

```
char* funptr = __fake_hook(&foo);
using ReturnType = decltype(foo(args...);
ReturnType ret;
if (funptr) {
    ret = funptr(args...);
} else {
    ret = foo(args...);
```

[1] https://github.com/martong/finstrument_mock

```
define i32 @_Z3bari(i32 %p) #0 {
entry:
  %fake_hook_result = tail call i8*
     @__fake_hook(i8* bitcast (i32 (i32)* @_Z3fooi to i8*))
  %0 = icmp eq i8* %fake_hook_result, null
  br i1 %0, label %else, label %then

then:                                   ; preds = %entry
  %1 = bitcast i8* %fake_hook_result to i32 (i32)*
  %subst_fun_result = tail call i32 %1(i32 %p)
  br label %cont

else:                                   ; preds = %entry
  %call = tail call i32 @_Z3fooi(i32 %p)
  br label %cont

cont:                                   ; preds = %else, %then
  %call_res.0 = phi i32 [ %subst_fun_result, %then ], [ %call, %else ]
  ret i32 %call_res.0
}
```

Figure 3. LLVM IR for function replacement

```
}
return ret;
```

Our prototype is based on LLVM/Clang [24, 27]. The implementation modifies the emitted LLVM Intermediate Representation (IR) [28] code. For instance, let us consider the following definition of the `bar` C++ function:

```
int foo(int);

int bar(int p) {
    return foo(p);
}
```

The LLVM IR of `bar` after optimization looks like this:

```
define i32 @_Z3bari(i32 %p) #0 {
entry:
  %call = tail call i32 @_Z3fooi(i32 %p)
  ret i32 %call
}
```

The generated code is very straightforward: there is only one basic block (`entry`) which stores the return value from the call of `foo` and then it returns with it. Note that the function names are mangled thus we see the `_Z3` prefix for the function names.

When we enable our instrumentation and optimization, then the IR has the form presented in Figure 3. Now we have four different basic blocks. The first block (`entry`) evaluates the return value of the `__fake_hook` function, compares it to zero and emits a branch based on the comparison. The `then` block is executed if the callee shall be replaced. We call the substituting function pointer, then we jump to the last basic block(`cont`). The `else` block is executed if the callee shall not be substituted; we just simply call the original function then jump to the `cont` block. At last, in the `cont` block, we store the result of either the callee or the replaced function, and we return with that.

Clang's internal architecture is built in such a way that the code generation for all kind of call expressions are eventually handled in one common routine. For example, in the case of virtual function calls the adjustment of the `this` pointer happens before calling that routine. We placed the emission of our instrumentation code inside that routine. As a result, special cases such as the `this` adjustment are automatically handled; we do not have to manually adjust the `this` pointer when we substitute a virtual function.

Contradictory to `-finstrument-functions`, by instrumenting the call expressions (and not the function body) we have the convenience that we do not have to recompile dependant libraries if the call expression is in a code outside of the library. This has a clear advantage in case of system libraries, third party shared libraries and security critical applications where we have to evade library interposing.

5.2 Runtime Library

The main purpose of the runtime library is to implement the `__fake_hook` function which is referenced from the instrumented code. The realization of this hook function has to find the related function pointer in case of an active substitution. Essentially, it is a simple pointer to pointer mapping which may be implemented with a simple hash function. However, in order to make the lookup as fast as possible, we chose to implement the mapping with a simple offsetting into the virtual memory (shadow memory). During program startup – more precisely, when our shared object is loaded – we initialize the shadow memory with the help of the `mmap` [34] system call.

We assume that a size of a function definition is at least 1 byte, since it has to contain at least a return instruction. Let N denote the size of a pointer in bytes of a specific architecture. Since we have to store a function pointer for every function, we have to reserve a shadow memory which is N times bigger than the normal virtual address space which holds the function definitions. Modern compilers on x64 systems with `-O2` optimization generate function definitions to be aligned to a 16 byte boundary; they achieve this usually by emitting additional `nop` instructions. Therefore, on such systems where all function addresses are aligned, it might be feasible to reserve a smaller shadow memory. If the `mmap` system call is called with the `MAP_ANONYMOUS` argument then it guarantees that the reserved memory is initialized to zero. Note that in practice the OS does not zero out the mapped region during the mapping, only at the moment when a virtual addressed is being accessed the first time. We divide the user-space virtual memory into two different regions. Low memory and high memory. We handle the memory mapping differently for each region. For instance, on macOS the memory is partitioned as follows:

```
[0x7f0000000000, 0x7fffffffffff] || HighMem
[0x120000000000, 0x19ffffffffff] || HighShadow
[0x020000000000, 0x11ffffffffff] || LowShadow
[0x000000000000, 0x01ffffffffff] || LowMem
```

Let $addr$ denote the original address ,$shadowAddr$ the address of the corresponding shadow and $shadowOffset$ the offset for a region. We calculate the shadow address with this formula: $shadowAddr = addr * N + shadowOffset(region(addr))$. By using the shadow memory instead of a simple hash map we trade execution time for space. The program occupies terabytes in virtual memory, however the resident (physical) memory usage is equal to the number of used substitutions multiplied with N. More specifically, operating systems do not reserve the specific physical pages to the process until there is no write to that memory area. Consequently, those memory pages which contain the shadow values of substituted functions will be resident physical pages registered in the process page table. In practice, this means only a few kilobytes of additional physical memory usage (given a page has 4kb size and not taking into account the Linux specific huge pages).

During program startup we must make sure that our shared object gets initialized before the first function call. Our prototype achieve this by setting the `constructor` attribute [13] on the initializer function of the shared object. If there are other shared libraries linked to the final executable with such initializer functions, then it is the user's responsibility to ensure that our library is initialized first.

Another purpose of the runtime library is to provide the user interface to setup the function substitutions. Replacing a function in C is pretty simple, the shared object defines a function for that:

```
_substitute_function((const char*)&foo,
                     (const char*)&fake_foo);
```

We may use the `SUBSTITUTE` macro in case of C++ to replace functions; this construct is more generic because it also supports

```
1  template <typename Class, typename MemPtr>
2  const char *
3  address_of_virtual_fun(const Class *aClass,
4                         MemPtr memptr) {
5    const char **vtable = *(const char ***)aClass;
6
7    struct pointerToMember {
8      size_t pointerOrOffset;
9      ptrdiff_t thisAdjustment;
10   };
11
12   pointerToMember p;
13   memcpy(&p, &memptr, sizeof(p));
14
15   static const size_t pfnAdjustment = 1;
16   size_t offset =
17       (p.pointerOrOffset - pfnAdjustment) /
18       sizeof(char *);
19
20   return vtable[offset];
21 }
```

Figure 4. Get the address of a virtual function (Itanium C++ ABI)

```
struct B {
    virtual void foo();
};
struct D : B {
    void foo() override;
};
```

Figure 5. A simple class hierarchy with a virtual function

member functions. Note that we have to include the header file attached to the runtime library, also we have to link with it. Our implementation is thread safe if there are multiple threads calling the very same function. Although, there is a race condition if one thread is calling the specified function while another thread is setting up the substitution; in such cases, the user code must ensure thread safety.

5.3 Virtual Functions

A pointer-to-member function may have a different layout in case of virtual functions than in case of regular member functions. Therefore, we cannot just simply cast a virtual function pointer to a `void` pointer.

5.3.1 ABI dependent substitution

Without compiler support, we can get the address of a virtual function in an architecture dependent way. On Figure 4 we present how we can get the address in case of the Itanium C++ ABI [21]. First, we receive the vtable from an object by dereferencing its vpointer (line 5). The vpointer is the first element in the object. We interpret the bits of the pointer to member (`memptr`) as an instance of the aggregate class `pointerToMember` (lines 7-13). Next, we setup the architecture dependent function pointer adjustment (line 15). Then, we get the offset and return with the appropriate element in the vtable (lines 16-20).

The API in our runtime library provides a function template, which we could use to replace virtual functions by exploiting the above presented technique. Let us consider the class hierarchy in Fig 5. If we wanted to replace the `foo()` function when the dynamic type was `D` then we had to get a pointer to such an instance:

```
B* dummy = new D;
SUBSTITUTE_VIRTUAL(&D::foo, dummy, &D_fake_foo);
```

However, to replace the function in the base class as well, we had to have a pointer to an instance whose dynamic type was `B`:

```
B* dummy = new B;
SUBSTITUTE_VIRTUAL(&B::foo, dummy, &B_fake_foo);
```

5.3.2 New compiler intrinsic

The `SUBSTITUTE_VIRTUAL` template is ABI dependent and it also requires a reference to an existing object. Therefore, we further investigated our options to find a better alternative without those restrictions.

Generally speaking, in order to replace functions we just need an identifier for each function – virtual or not – which is unique in the program. Actually, each function has such a unique identifier, and it is its own address in the program's virtual memory. Unfortunately, there is no valid C++ language construct to get this unique identifier. Nevertheless, GCC has implemented this feature [12], but sadly Clang did not. Clang developers claim that this feature is fundamentally broken, because when we use it then the proper adjustment of the `this` pointer may be elided [25]. Still, our technique could use this feature since our compiler instrumentation intervenes after the this adjustment thunk is emitted. Thus, we implemented this functionality in the Clang compiler, so we are able to use it within our implementation, hidden from the users and enabled only in test code.

Considering the previous example in Fig 5, the replacement of the `foo()` function when the dynamic type is `D` has the following form:

```
SUBSTITUTE(D::foo, D_fake_foo);
```

This is the very same form which we can use to replace free functions or non-virtual member functions.

Internally, the `SUBSTITUTE` macro expands to a call to `_substitute_function` and the arguments of that function are generated by our new compiler intrinsic:

```
#define SUBSTITUTE(src, dst)                    \
  do {                                          \
    _substitute_function(                       \
        (const char *)__function_id src,        \
        (const char *)__function_id dst);       \
  } while (0)
```

We modified the compiler to parse a new kind of unary expression when the `__function_id` literal is given and the test specific instrumentation is enabled. In case of free functions and static member functions this unary expression has the very same type which we would get in case of the "address of" unary expression:

```
void foo();
void bar() {
  auto p = & foo; // void (*)()
  auto q = __function_id foo; // void (*)()
}
```

However in case of non-static member functions the two expressions yield different types:

```
struct X { void foo(); virtual void bar(); };
void bar() {
  auto p = & X::foo; // void (X::*)()
  auto q = __function_id X::foo; // void (*)()
  auto r = __function_id X::bar; // void (*)()
}
```

At runtime the value of these expressions are evaluated to hold the address of the specific raw function which can be identified by the corresponding mangled name in the compiled binary's text section.

5.4 Overload Resolution

We may have several functions with the same name but with different parameters. Let us consider the below code:

```
1  #define SUBSTITUTE_BASE(src, dst)                    \
2    do {                                               \
3      _substitute_function(                            \
4          (const char *)__function_id src,             \
5          (const char *)__function_id dst);            \
6    } while (0)
7
8  #define SUBSTITUTE_OVERLOAD(signature, src, dst) \
9    do {                                               \
10     using SignAlias = signature;                     \
11     using FunPtr = SignAlias *;                      \
12     FunPtr funptr = __function_id src;               \
13     _substitute_function(                            \
14         (const char *)funptr,                        \
15         (const char *)__function_id dst);            \
16   } while (0)
17
18 #define GET_MACRO(_1, _2, _3, NAME, ...) NAME
19 #define SUBSTITUTE(...)                              \
20   GET_MACRO(__VA_ARGS__, SUBSTITUTE_OVERLOAD,        \
21             SUBSTITUTE_BASE)(__VA_ARGS__)
```

Figure 6. The SUBSTITUTE macro as a dispatcher

```
struct X {
  int foo(int) { return 1; }
  int foo(double) { return 2; }
}
int X_fake_foo_i(X*, int);
```

Normally, if we would like to get the address of `X::foo(int)` we have to explicitly cast a function pointer to the appropriate type:

```
int(X::*mfp)(int) = & X::foo;
```

Here, we define a pointer variable with the name `mfp` which has the type `int(X::*)(int)` and it holds the address of `X::foo`. In case of the `__function_id` intrinsic we have to do the same, but the type will be different:

```
int(*mfid)(int) = __function_id X::foo;
```

For safety reasons, the users of our instrumentation must not use the `__function _id` directly, but they can use the three parameter form of the provided `SUBSTITUTE` macro to replace an overloaded function. For example, to replace `X::foo` with the `X_fake_foo_i` free function we have to write:

```
SUBSTITUTE(int(int), X::foo, X_fake_foo_i);
```

Actually, the `SUBSTITUTE` macro is implemented in terms of three other macros. In Fig 6, the `GET_MACRO` dispatches between the two different flavors of the `SUBSTITUTE` macro. The `SUBSTITUTE_BASE` macro has two parameters while `SUBSTITUTE_OVERLOAD` has three parameters. In the `SUBSTITUTE_OVERLOAD` macro we handle the case where an additional signature is passed in as the first parameter. First we create a type alias to the signature (line 10) then we create yet an other alias to the type of the function pointer (line 11). By using these two type aliases we can conveniently handle both the free functions and the member functions with one macro.

5.5 Other Special Cases

A few standard library functions, such as `abort` and `exit`, cannot return. Some programs define their own functions that never return. We can declare them `noreturn` to tell the compiler this fact The compiler can optimize without regard to what would happen if a `noreturn` function ever did return. This makes slightly better code.[13]. Our prototype supports the substitution of functions with the `noreturn` attribute with functions which do return. We achieve this by generating such code for the call expression which we would generate in case of normal functions on the branch where the substitution is active.

Generally, during compilation, functions are not inlined unless optimization is specified. For functions declared inline, the `always_inline` attribute [13] inlines the function even if no optimization level was specified. Our implementation makes it possible to replace always-inline functions, if a special program option is passed for the compiler. Naturally, the given function definition will not be inlined and it will be emitted as a standalone function with an address. However there are special cases with always-inline function declarations. For instance, in the case of the libcxx library – which is one standard C++ library implementation – most of the getters and setters have the `always_inline` attribute. For example, the `basic_string` class template declares `c_str()` as always inline but there is an `extern` template declaration in the `<string>` header for `basic_string<char>`. Also, there is an implicit template instantiation of `basic_string<char>` which does not expose `c_str()`, as that is declared to be always-inline. When we turn on our instrumentation, the generated object file has an undefined reference to `c_str()`, since the code is not emitted because of the `extern` template declaration. To make the instrumentation work either we have to recompile libcxx with our instrumentation enabled and we have to link against the instrumented libcxx, or we have to eliminate somehow the `extern` template declaration. The latter is possible in one of our branch of the libcxx repository. Note that we did not experience this interesting case with the GNU standard C++ library implementation on Linux (GCC/6.2 version).

A C++ `constexpr` function cannot be replaced when it is used in a compile-time expression. However, it can be replaced whenever it is used within a runtime context:

```
constexpr int foo(int p) { return p * p; }
int fake_foo(int p) { return p * p * p; }

TEST_F(FooFixture, Constexpr) {
    SUBSTITUTE(foo, fake_foo);

    // compile-time evaluation
    static_assert(foo(2) == 4, "");

    // runtime evaluation
    EXPECT_EQ(foo(2), 8);
}
```

The expression inside the `static_assert` is forced to be evaluated during the compilation.

6. Performance Evaluation

We did not notice any degradation in the run time and in the memory usage of the compilation process itself when our instrumentation was turned on. We measured the runtime performance of the compiled instrumented code with the help of the Adobe C++ Performance Benchmark [1]. Adobe's primary goals with their benchmark are

- to help compiler vendors identify places where they may be able to improve the performance of the code they generate
- to help developers understand the performance impact of using different data types, operations, and C++ language features with their target compilers and OSes.

We customized their benchmark to our needs. We removed test suites about loop unrolling, constant folding and loop invariants. We use exactly 3 different test suites, they measure overhead about different abstractions:

- The function objects test suite compares the performance of function pointers, functors, inline functors, standard functors, and native comparison operators. (This test suite contains only one test case.)

- The Stepanov abstraction test suite examines any change in performance when adding abstraction to simple data types. For instance, a value wrapped in a class may perform worse than a raw value. Also, a value recursively wrapped in a struct or class may perform worse than a raw value. There are several different test cases in this test suite. For instance, there is a test case for measuring the performance of an insertion sort when several layers of abstractions are applied. One other test case measures this abstraction penalty for heap sort.
- The Stepanov vector test suite examines any change in performance when moving from pointers to vector iterators. Vector iterators may perform worse than raw pointers. There is only one test case in this test suite, which measure the performance in case of applying quicksort on a `std::vector`.

Note that a good optimizing compiler with `-O2` or `-O3` optimization level should not produce any performance penalty on behalf of the abstractions. For instance, the MSVC 2008 compiler has an abstraction penalty ratio not greater than 1.82 [2].

We compiled the test suites with different compiler flags and we compared the absolute total time of run time of each test cases. On Figure 7 we show the total time for the function objects test case with the different compiler setups. Our instrumentation is turned on with the `-fsanitize=mock` option. By providing the `-DSUB` switch we define a macro. If that macro is present then we do substitute functions in the busy loop of each test cases. With the `-finstrument-functions` setup we define the `__cyg_profile` functions in a standalone, separate translation unit. Figure 8 presents the total absolute time for the vector use case from the Stepanov vector test suite. Similarly, Figure 9 represents the total time in one test case of the Stepanov abstraction suite.

On Figure 10 we can see the normalized total times. We display for each compiler setup the execution times of all the test cases (normalized by the longest execution time per test case). The benchmark computes the penalty of an abstraction by dividing the run time of a test execution by the time it takes to run the same algorithm but without any abstractions applied. We present the abstraction penalties on Figure 11. We can see that our instrumentation causes similar penalty as `-finstrument-functions` when the `-O2` optimization is enabled.

Both our instrumentation and `-finstrument-functions` causes performance degradation. In most cases our technique performs similarly to `-finstrument-functions`. Our method exchanges two extra function calls (`__cyg_profile_func_enter` and `__cyg_profile_func_exit`) with one extra function call to `__fake_hook` followed by an efficient lookup.

When we enable our instrumentation and we do replace some functions, then we may experience some performance degradation compared to when we just simply enable the new instrumentation (up to 15%). The reason behind this is that when we replace a function we always take that branch which calls a function via a pointer and in some cases we may loose important hardware optimization opportunities (e.g. prefetching of instructions). Note that there is an important optimization possibility in case of the non-replacing branch. If we could include the definition of the `__fake_hook` function when we emit the LLVM IR for a call expression than the upcoming optimizer passes of the compiler might be able to inline it. Also, by this inlining, on the non-replacing branch, further inlining optimization opportunities might be exploited. We plan to investigate the feasibility of this inlining in the future.

We experienced that with our instrumentation, the size of the binary may grow bigger. In the case of a simple C program we measured around 15% (bzip2). In the case of a template heavy program (Stepanov abstractions test suite) we measured that the

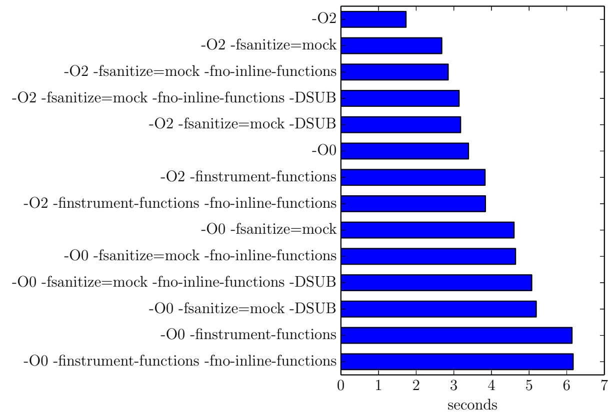

Figure 7. Total absolute time for function objects

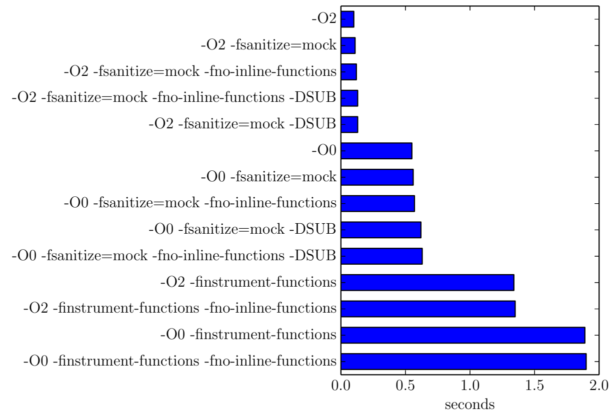

Figure 8. Total absolute time for vector quicksort

executable could be up to 3 times bigger compared to an `-O2` optimized binary. We measured very similar size growth in case of the `-finstrument-functions` feature.

We performed the measurements on a Linux machine with an Intel(R) Core(TM) i7-4610M CPU @ 3.00GHz processor and with 16GB RAM. The given CPU is a laptop-class hardware that scales the frequency dynamically from 0.8Ghz to 3.7Ghz, therefore we turned off turbo boost and frequency scaling by using the appropriate ACPI kernel driver.

Our measurement scripts are publicly available [38] and the measurements can be reproduced easily in other machines.

7. Limitations And Future Work

Our prototype implementation works only on 64 bit x86 systems. More specifically, we tested it on macOS El Capitan and on Ubuntu Linux 16.04 and 16.10. Since we have to map a relatively big shadow memory, supporting 32 bit systems might require a different lookup algorithm which might not be as efficient than the simple offsetting; for instance, we could use a simple hash map implementation instead.

Replace the `operator()` of a lambda is not supported unless we can take the address of the lambda. Similarly, member functions

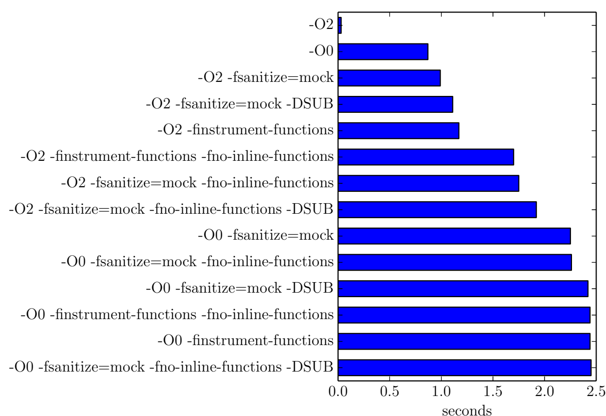

Figure 9. Total absolute time for abstraction insertion sort

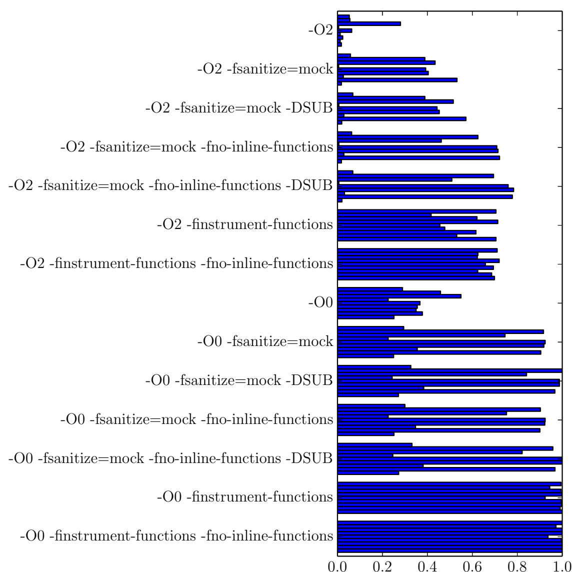

Figure 10. Normalized total absolute times

of structs/classes which are defined inside a function cannot be replaced, because there is no valid expression to get their address. Our technique relies on that we should be able to get the address of the function we want to substitute. In case of constructors and destructors we cannot get their address with any standard C++ expression. Still, replacing constructors or destructors would be a valuable contribution in the domain of testing, thus this is an important area for further research.

There is an important optimization possibility in case of the non-replacing branch as we described in Section 6. Our prototype uses 8 bytes for each function address in the shadow memory. However we support only replacing of user-space functions, therefore it would be enough to use 6 bytes. Also, on systems where the function definitions are all aligned it might be possible to allocate smaller shadow memory.

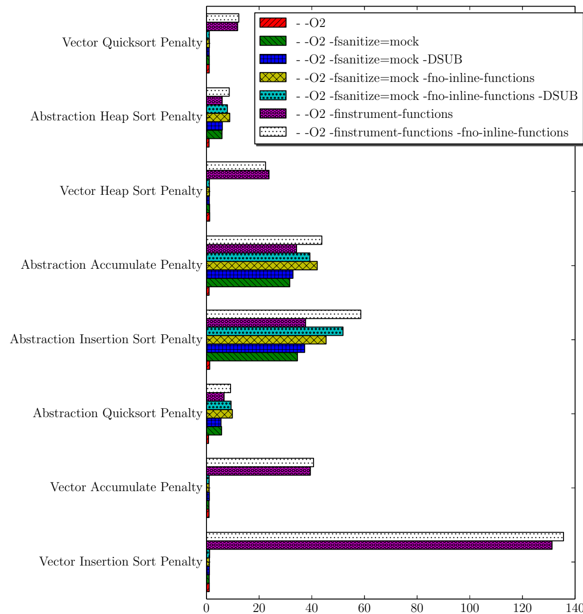

Figure 11. Abstraction penalty

It is worth the effort to investigate another optimization possibility: to use a hybrid run-time and compile-time FCI technique, similarly as XRay does. We could insert no-op sleds before the call expression of the original function instead of explicitly emitting the two basic blocks (`entry` and `then` in Fig 3) which handle the call of the mock function before the `cont` block. It is a hard open question how could we handle the variable length of these basic blocks. During runtime if the function is not replaced, these no-op sleds would be executed, however if the function is replaced then the runtime library could overwrite these no-ops with the appropriate additional basic blocks. Note, we should also pad with no-ops the generated code for the `phi` node to be able to overwrite the corresponding parts. We plan to investigate the feasibility of these performance opportunities in the future.

Currently we do not have any check to enforce that the original function and its replacement have the same signature. In the future we plan to create a checking function template for the substitutions.

8. Related Work

The different function call interception techniques are explained in details by Kang [22]. The author also discusses aspect-oriented programming implementation techniques for intercepting method calls.

The four-phase test automation pattern is introduced by Meszaros [39] and the given-when-then pattern is described by North [43]. Feathers describes different techniques about testing legacy code in his book [10]. He introduces the concept of seams via we can alter behaviour without changing the original unit. Rüegg and Sommerlad elaborate this concept in C++ [47].

There are many software error checking tools which are based on some kind of instrumentation. A large number of memory error detectors are based on binary instrumentation. For example, Valgrind (Memcheck) [42], Dr. Memory [5], Purify [18], Intel Inspector [19]. The most popular compiler instrumentation based error checker tools are the AddressSanitizer [49], ThreadSanitizer [48], XRay instrumentation [4, 26] and other different sanitizers sup-

ported by the LLVM/Clang infrastructure [29, 30]. Our instrumentation technique was inspired by the AddressSanitizer, we reused many ideas from its implementation (e.g shadow memory).

Shadow memory is often used by different error checker software. The above mentioned AddressSanitizer and ThreadSanitizer both use shadow memory to store metadata for a specific piece of memory. AddressSanitizer uses a shadow space scaled down to one eight of the normal address space and can be easily used on 32 bit systems. However, ThreadSanitizer uses 8 times larger shadow memory than the normal address range, therefore support for 32-bit platforms is problematic and is not planned by the maintainers. We can find users of the shadow memory mapping outside of the LLVM universe as well. For instance Valgrind [41] TaintTrace [7], LIFT [46], Bound-Less [6], Umbra [52, 53] and LBC [17].

9. Conclusion

Function call interception is a technique of intercepting function calls at program runtime. Without directly modifying the original code, FCI enables to undertake certain operations before and/or after the called function or even to replace the intercepted call.

FCI dynamic techniques perform instrumentation at load time or at runtime. Static techniques are applied either at the application binary level or during the compilation proper (compiler instrumentation). With existing compiler instrumentation techniques for languages like C and C++ we can provide hooks which are called before and after a function, but we cannot replace an intercepted function call.

Test seams are used to create non-intrusive tests for legacy systems, some of these seams are often realized via an FCI technique. We introduced our new compiler instrumentation for C and C++ programs, which makes it possible to replace the intercepted function call. While most of the existing instrumentation methods modify the function to call we instrument the caller side. We substitute the actual call with a small code snippet in compilation time, which decides at runtime whether the original or a replacement function is about to call. The decision is made using shadow memory and an offset to minimize runtime overhead.

In contrast to other seams, our new instrumentation seam keeps the test setup code close to the other phases of the test. The technique makes it feasible to write non-intrusive tests which follow the given-when-then test pattern. This way, our method could help to implement high-quality tests for legacy software systems.

Compared to existing compile-time instrumentation solutions, our technique does not require the modification or even the recompilation of the intercepted functions, which is a possible advantage in case of legacy code, system libraries, third party shared libraries or in situations when we have to avoid library interposing. Our technique is also capable to substitute both C/C++ free functions as well as C++ member functions.

We have created a prototype implementation using the LLVM/-Clang compiler infrastructure. The modified C++ compiler replaces the call expressions and a runtime library looks up the substitutions. We have evaluated the prototype using various benchmarks. We measured the runtime overhead is similar to the overhead caused by the other compile-time instrumentation, `-finstrument-functions`.

References

[1] Adobe. C++ performance benchmarks, 2017. URL `https://stlab.adobe.com/performance`.

[2] Adobe. Abstraction penalty with msvc 2008, 2017. URL `https://stlab.adobe.com/wiki/index.php/Performance/Analysis/Example3`.

[3] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Binary rewriting and call interception for efficient runtime protection against buffer overflows: Research articles. *Softw. Pract. Exper.*, 36(9):971–998, July 2006. ISSN 0038-0644. doi: 10.1002/spe.v36:9. URL `http://dx.doi.org/10.1002/spe.v36:9`.

[4] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. Xray: A function call tracing system. 2016.

[5] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL `http://dl.acm.org/citation.cfm?id=2190025.2190067`.

[6] Mark Brünink, Martin Süßkraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 13–24, June 2011. doi: 10.1109/DSN.2011.5958203.

[7] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2588-1. doi: 10.1109/ISCC.2006.158. URL `http://dx.doi.org/10.1109/ISCC.2006.158`.

[8] Shigeru Chiba. A metaobject protocol for c++. *SIGPLAN Not.*, 30 (10):285–299, October 1995. ISSN 0362-1340. doi: 10.1145/217839.217868. URL `http://doi.acm.org/10.1145/217839.217868`.

[9] clang.llvm.org. Clang command line argument reference, 2017. URL `https://clang.llvm.org/docs/ClangCommandLineReference.html`.

[10] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. ISBN 0131177052.

[11] Martin Fowler. Givenwhenthen. URL `https://martinfowler.com/bliki/GivenWhenThen.html`.

[12] gcc.gnu.org. Extracting the function pointer from a bound pointer to member function, 2017. URL `https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/Bound-member-functions.html`.

[13] gcc.gnu.org. Declaring attributes of functions, 2017. URL `https://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Function-Attributes.html`.

[14] gcc.gnu.org. Program instrumentation options, 2017. URL `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`.

[15] gnu.org. Using gnu ld, 2017. URL `ftp://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html`.

[16] gnu.org. Gdb: The gnu project debugger, 2017. URL `https://www.gnu.org/software/gdb/`.

[17] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259034. URL `http://doi.acm.org/10.1145/2259016.2259034`.

[18] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[19] Intel. Intel inspector, 2017. URL `https://software.intel.com/en-us/intel-inspector-xe`.

[20] Intel. Pintool api reference - rtn: Routine object, 2017. URL `https://software.intel.com/sites/landingpage/pintool/docs/53271/Pin/html/group__RTN__BASIC__API.html`.

[21] Intel, CodeSourcery, Compaq, EDG, HP, IBM, Red Hat, and SGI. Itanium c++ abi, 2017. URL `http://refspecs.linuxbase.org/cxxabi-1.83.html`.

[22] Pilsung Kang. Function call interception techniques. *Software: Practice and Experience*, pages n/a–n/a. ISSN 1097-024X. doi: 10.1002/spe.2501. URL http://dx.doi.org/10.1002/spe.2501. spe.2501.

[23] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, 26(5):313–, September 2001. ISSN 0163-5948. doi: 10.1145/503271.503260. URL http://doi.acm.org/10.1145/503271.503260.

[24] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[25] llvm.org. Clang will not accept a conversion from a bound pmf to a regular method pointer, 2017. URL https://bugs.llvm.org/show_bug.cgi?id=22121.

[26] llvm.org. Xray instrumentation, 2017. URL https://llvm.org/docs/XRay.html.

[27] llvm.org. clang: a c language family frontend for llvm, 2017. URL http://clang.llvm.org.

[28] llvm.org. Llvm language reference manual, 2017. URL http://llvm.org/docs/LangRef.html.

[29] llvm.org. Memory sanitizer, 2017. URL https://clang.llvm.org/docs/MemorySanitizer.html.

[30] llvm.org. Undefined behavior sanitizer, 2017. URL https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065034. URL http://doi.acm.org/10.1145/1065010.1065034.

[32] Linux Programmer's Manual. dlsym, dlvsym - obtain address of a symbol in a shared object or executable, 2017. URL http://man7.org/linux/man-pages/man3/dlsym.3.html.

[33] Linux Programmer's Manual. ld.so, ld-linux.so - dynamic linker/loader, 2017. URL http://man7.org/linux/man-pages/man8/ld.so.8.html.

[34] Linux Programmer's Manual. mmap, munmap - map or unmap files or devices into memory, 2017. URL http://man7.org/linux/man-pages/man2/mmap.2.html.

[35] Linux Programmer's Manual. ptrace - process trace, 2017. URL http://man7.org/linux/man-pages/man2/ptrace.2.html.

[36] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[37] Gábor Márton. Access private, 2017. URL https://goo.gl/ynaZv5.

[38] Gábor Márton. Instrumentation for testing, 2017. URL https://github.com/martong/finstrument_mock.

[39] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[40] mockator.com. An eclipse cdt plug-in for c++ seams and mock objects. URL http://mockator.com/. http://mockator.com/.

[41] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254820. URL http://doi.acm.org/10.1145/1254810.1254820.

[42] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6): 89–100, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL http://doi.acm.org/10.1145/1273442.1250746.

[43] D North. Introducing bdd, better software magazine, 2006.

[44] Pradeep Padala. Playing with ptrace, Part I. 103, November 2002. ISSN 1075-3583 (print), 1938-3827 (electronic).

[45] Pradeep Padala. Playing with ptrace, Part II. 104, December 2002. ISSN 1075-3583 (print), 1938-3827 (electronic).

[46] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: 10.1109/MICRO.2006.29. URL https://doi.org/10.1109/MICRO.2006.29.

[47] Michael Rüegg and Peter Sommerlad. Refactoring towards seams in c++. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 117–123, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1822-8. URL http://dl.acm.org/citation.cfm?id=2663608.2663632.

[48] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. doi: 10.1145/1791194.1791203. URL http://doi.acm.org/10.1145/1791194.1791203.

[49] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342849.

[50] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in aop with aspectc++. In *Proceedings of the 2005 Conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the Fourth SoMeT_W05*, pages 33–53, Amsterdam, The Netherlands, The Netherlands, 2005. IOS Press. ISBN 1-58603-556-8. URL http://dl.acm.org/citation.cfm?id=1563296.1563301.

[51] Sai Venkatakrishnan. Build operate check clear - test pattern. URL http://developer-in-test.blogspot.hu/2009/05/build-operate-check-clear-test-pattern.html.

[52] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 93–102, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0054-4. doi: 10.1145/1806651.1806667. URL http://doi.acm.org/10.1145/1806651.1806667.

[53] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 22–31, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772960. URL http://doi.acm.org/10.1145/1772954.1772960.