STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.02

COMPILE-TIME FUNCTION CALL INTERCEPTION FOR TESTING IN C/C++

GÁBOR MÁRTON AND ZOLTÁN PORKOLÁB

Abstract. In C/C++, during the test development process we often have to modify the public interface of a class to replace existing dependencies; e.g. supplementary setter or constructor functions or extra template parameters are added for dependency injection. 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.

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 [2, 28]. Function call interception (FCI) is often the only tool

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: PROGRAMMING LANGUAGES – *Languages Constructs and Features*.

Key words and phrases. C++ programming language, unit testing, function call interception, compiler instrumentation.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

which enables non-intrusive testing by making it possible to replace function bodies. By replacing functions we can eliminate the unwanted dependencies in tests. 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 [11]. The different FCI methods have different advantages and disadvantages. 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.

In this paper, we investigate a new compile-time instrumentation based FCI approach for C/C++ programs which enables the replacement of functions and methods. 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. We implemented a prototype based on the LLVM/Clang compiler infrastructure.

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. In Section 6, we describe the current limitations and possible future work. We have an overview of the related works in Section 7. Our paper concludes in Section 8.

2. Function Call Interception Techniques

We differentiate the FCI techniques based on the time FCI is applied [11]. *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.

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 [17]. With this technique, calling the original function is cumbersome. We have to use `dlsym` auxiliary function with the `RTLD_NEXT` argument [16]. 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 [10].

Run-time FCI. In Unix like systems, runtime dynamic interception is implemented with the help of the `ptrace` system call [19, 27]. 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) [7] and Intel Pin [15] 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 [9]. Another property of this technique is that we cannot instrument inline functions.

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 to collect statistics about the heap usage. 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.

Link-time FCI. One example for the link-time static interception is the `wrap` command line option of the GNU linker (`ld`) [8]. 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.

Post-compilation-time FCI. There exist tools to modify the compiled binary code for interception. As an example, in [1] 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.

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 provide 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 [6]. 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 limitation of this technique is that we cannot replace an intercepted function with another function; the original function will be called anyway.

3. Test Automation Conventions

The FCI techniques discussed above are frequently used in the process of creating automated tests. Thus, in this section we overview the general test automation patterns and we show the more specialized concepts about testing legacy code.

The *four-phase* test pattern is driven by the observation that each test requires some sort of setup and tear down routines. This pattern splits each test into four phases [24]. 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 known as the build-operate-check-clear pattern [31].

The *given-when-then* pattern of representing tests is originated from behavior-driven development [26, 3]. 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 behavior 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 [20]. 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.

Unwanted dependencies embody a critical problem in software development; we often have to break existing dependencies before we can change some piece of code [28]. Breaking existing dependencies is also an important prerequisite to introduce unit tests for legacy code [2].

A *seam* is an abstract concept introduced by Feathers to identify points where we can break dependencies [2]. The goal is to have a place where we can alter the behavior of a program without modifying it in that place; this is important because editing the source code is often not an option [28]. Feathers, Rüegg and Sommerlad define four different kinds of seams for C++ [2, 28]. *Link seam*: Change the definition of a function via some linker specific setup. *Preprocessor seam*: With the help of the preprocessor, redefine function names to use an alternative implementation. *Object seam*: Based on inheritance to inject a subclass with an alternative implementation. *Compile seam*: Inject dependencies at compile-time through template parameters. The *enabling point* of a seam is the place where we can make the decision to use one behavior or another. Different seams have different enabling points.

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 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

```
1  // Turtle.hpp
2  class Turtle {
3    int x = 0, y = 0;
4  public:
5    void PenUp() { /* ... */ }
6    void PenDown() { /* ... */ }
7    void Forward(int distance) { /* ... */ }
8    void Turn(int degrees) { /* ... */ }
9    void GoTo(int x, int y) { /* ... */ }
10   int GetX() const { return x; }
11   int GetY() const { return y; }
12 };
13
```

```
14 class Painter {
15   Turtle turtle;
16 public:
17   void DrawLine(int x0, int y0, int x1,
18                 int y1) {
19     turtle.GoTo(x0, y0);
20     turtle.PenDown();
21     turtle.GoTo(x1, y1);
22     turtle.PenUp();
23   }
24   // ...
25 };
```

FIGURE 1. A legacy graphics program

```
1  #include "Turtle.hpp"
2  #include <gmock/gmock.h>
3  #include <access_private.hpp>
4  #include <hook.hpp> // for SUBSTITUTE
5
6  class MockTurtle {
7  public:
8    MOCK_METHOD0(PenUp, void());
9    // PenDown, Forward, ...
10 };
11
12 MockTurtle &GetMockObject(Turtle *) {
13   static MockTurtle m;
14   return m;
15 }
16
17 namespace proxy {
18 void PenUp(Turtle *self) {
19   return GetMockObject(self).PenUp();
20 }
21 // Similarly to PenDown, Forward, ...
22 }
23
24 struct TurtleTest : ::testing::Test {
25   TurtleTest() {
26     SUBSTITUTE(Turtle::PenUp, proxy::PenUp);
```

```
27     // Similarly to PenDown, Forward, ...
28   }
29 };
30
31 ACCESS_PRIVATE_FIELD(Painter, Turtle,
32                      turtle)
33
34 TEST_F(TurtleTest, TestDrawLine) {
35   using ::testing::AtLeast;
36
37   Painter painter;
38   Turtle &turtle =
39       access_private::turtle(painter);
40   MockTurtle &mockTurtle =
41       GetMockObject(&turtle);
42
43   EXPECT_CALL(mockTurtle, PenDown())
44       .Times(AtLeast(1));
45   painter.DrawLine(0, 0, 10, 10);
46 }
47
48 int main(int argc, char **argv) {
49   ::testing::InitGoogleTest(&argc, argv);
50   return RUN_ALL_TESTS();
51 }
```

FIGURE 2. Testing the legacy program with compile-time FCI

like to write a test which checks the `DrawLine()` function. In this 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`) with the help of the gmock macros (lines 6-10). Our test-case is defined from line 34 to 46. In the test-case we create an instance of the `Painter` class, then we get a reference to its private `turtle` member (lines 38-39). Note that there are several different techniques

to access a private member, we use a method which relies on explicit template instantiations [21]. Then we get a reference to an instance of the `MockTurtle` class which acts as a test double for the `Turtle` instance (lines 40-41). We state our expectations on the mock object (lines 43-44). In line 45 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 26-28). These replacement functions behave as a proxy; they forward each function call on a given `Turtle` instance to a corresponding test double (lines 17-22). 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 12-15). 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 48-50 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: **(1)** The test setup is part of the test application and clearly visible together with the rest of the test code, thus it does not violate the given-when-then test automation pattern. **(2)** It does not introduce a new tool into the existing build chain. The functionality is embedded into the compiler. **(3)** On platforms where the compiler is supported, the new instrumentation could be supported as well. **(4)** There is no need to use mangled names. **(5)** 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 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.

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

(A)

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

(B)

FIGURE 3. Call expression substitution

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: `foo(args...);`. When our instrumentation is in action, the emitted code is equal to the pseudo code in Figure 3a. 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 as presented in Figure 3b. Our prototype is based on LLVM/Clang [12]. The implementation modifies the emitted LLVM Intermediate Representation (IR) [14] code. For instance, let us consider the definition of the `bar` C++ function in Figure 4a. The LLVM IR of `bar` after optimization is presented in Figure 4b. 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 4c. 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.

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

(A) C++

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

(B) Original LLVM IR

```
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  }
```

(C) Modified LLVM IR

FIGURE 4. LLVM IR modification for function replacement

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. 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`. Detailed measurement results are available online at [23].

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` [18] 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. 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. With this formula $shadowAddr = addr * N + shadowOffset(region(addr))$ we can calculate the shadow address. 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 achieves this by setting the `constructor` attribute [4] 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 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.

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

FIGURE 5. Get the address of a virtual function

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. *The naive approach.* Without compiler support, we can get the address of a virtual function in an architecture dependent way. On Figure 5 we present how we can get the address in case of the Itanium C++ ABI [10]. First, we receive the vtable from an object by dereferencing its vpointer (line 3). 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 4-9). Next, we setup the architecture dependent function pointer adjustment (line 10). Then, we get the offset and return with the appropriate element in the vtable (lines 11-12). We could replace virtual functions by exploiting this technique. Let us suppose we have a macro named `SUBSTITUTE_VIRTUAL` which use this technique and the following class hierarchy:

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

If we wanted to replace the `foo()` function when the dynamic type was `D` then we would have 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 would have to get 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 previous naive approach is ABI dependent and it also requires a reference to an existing object. Thus, we tried to find a better alternative without these 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 [5], 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 [13]. 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. With this approach, 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:

```
struct X {  int foo(int);  int foo(double);  };
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`. With 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 `__function _id` is hidden from the users of our instrumentation, 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 one have to write:

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

6. Limitations And Future Work

Our prototype is implemented in the code generation part of the Clang compiler, however it would be architecturally better if we realized that as a transforming optimizer pass. This pass should run before all other optimizer passes. By having an optimizer pass, all the logic related to this instrumentation would be well separated and self contained. Also, it would make it possible to use our tool with other language frontends, thus this is our most important future work. 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. The prototype works only on 64 bit x86 systems.

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

7. Related Work

The different function call interception techniques are explained in details by Kang [11]. The author also discusses aspect-oriented programming implementation techniques for intercepting method calls.

The four-phase test automation pattern is introduced by Meszaros [24] and the given-when-then pattern is described by North [26]. Feathers describes different techniques about testing legacy code in his book [2]. He introduces the concept of seams via we can alter behavior without changing the original unit. Rüegg and Sommerlad elaborate this concept in C++ [28].

There are plenty of 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) [25] or Dr. Memory. The most popular compiler instrumentation based error checker tools are the AddressSanitizer [30] and the ThreadSanitizer [29]. 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.

8. Conclusion

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. We have created a prototype implementation using the LLVM/Clang compiler infrastructure, which is publicly available at [22].

References

[1] Kumar Avijit et al. "Binary Rewriting and Call Interception for Efficient Runtime Protection Against Buffer Overflows: Research Articles". In: *Softw. Pract. Exper.* 36.9 (July 2006), pp. 971–998. ISSN: 0038-0644. URL: http://dx.doi.org/10.1002/spe.v36:9.

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

[3] Martin Fowler. *GivenWhenThen* . URL: https://martinfowler.com/bliki/GivenWhenThen.html.

[4] gcc.gnu.org. *Declaring Attributes of Functions*. 2017. URL: https://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Function-Attributes.html (visited on 06/24/2017).

[5] 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 (visited on 06/24/2017).

[6] gcc.gnu.org. *Program Instrumentation Options*. 2017. URL: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html (visited on 06/24/2017).

[7] gnu.org. *GDB: The GNU Project Debugger*. 2017. URL: https://www.gnu.org/software/gdb/ (visited on 06/24/2017).

[8] gnu.org. *Using GNU ld*. 2017. URL: ftp://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html (visited on 06/24/2017).

[9] 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 (visited on 06/24/2017).

[10] Intel et al. *Itanium C++ ABI*. 2017. URL: http://refspecs.linuxbase.org/cxxabi-1.83.html (visited on 06/24/2017).

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

[12] 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.

[13] 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 (visited on 06/24/2017).

[14] llvm.org. *LLVM Language Reference Manual*. 2017. URL: http://llvm.org/docs/LangRef.html (visited on 06/25/2017).

[15] Chi-Keung Luk et al. "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. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. URL: http://doi.acm.org/10.1145/1065010.1065034.

[16] 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 (visited on 06/24/2017).

[17] 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 (visited on 06/24/2017).

[18] 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 (visited on 06/24/2017).

[19] Linux Programmer's Manual. *ptrace - process trace*. 2017. URL: http://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 06/24/2017).

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

[21] Gábor Márton. *Access Private*. 2017. URL: https://goo.gl/ynaZv5 (visited on 06/25/2017).

[22] Gábor Márton. *finstrument-mock - Instrumentation for Testing*. 2017. URL: https://github.com/martong/finstrument_mock (visited on 06/25/2017).

[23] Gábor Márton. *Performance Measurements of finstrument-mock*. 2017. URL: https://github.com/martong/finstrument_mock/blob/master/measure/performance_evaluation.pdf (visited on 03/28/2018).

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

[25] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/1273442.1250746.

[26] D North. *Introducing BDD, Better Software Magazine*. 2006.

[27] Pradeep Padala. "Playing with ptrace, Part I". In: 103 (Nov. 2002). ISSN: 1075-3583 (print), 1938-3827 (electronic).

[28] Michael Rüegg and Peter Sommerlad. "Refactoring Towards Seams in C++". In: *Proceedings of the 7th International Workshop on Automation of Software Test*. AST '12. Zurich, Switzerland: IEEE Press, 2012, pp. 117–123. ISBN: 978-1-4673-1822-8. URL: http://dl.acm.org/citation.cfm?id=2663608.2663632.

[29] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: Data Race Detection in Practice". In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. WBIA '09. New York, New York, USA: ACM, 2009, pp. 62–71. ISBN: 978-1-60558-793-6. URL: http://doi.acm.org/10.1145/1791194.1791203.

[30] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, pp. 28–28. URL: http://dl.acm.org/citation.cfm?id=2342821.2342849.

[31] Sai Venkatakrishnan. *Build Operate Check Clear - Test Pattern*. URL: http://developer-in-test.blogspot.hu/2009/05/build-operate-check-clear-test-pattern.html.

Department of Programming Languages and Compilers, Eötvös Loránd University

Email address: martong@mailbox.elte.hu, gsd@elte.hu