Vtable hijacking: Object Type Integrity for run-time type information

Marco Schröder, Stefan Machmeier, Vincent Heuveline

Preprint No. 2023-01

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
Preprint No. 2023-01

Affiliation of the Authors

Marco Schöder[a], Stefan Machmeier[a,1], Vincent Heuveline[a]

[a] *Engineering Mathematics and Computing Lab (EMCL), Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Germany*
[1] *Corresponding Author: Stefan Machmeier, stefan.machmeier@uni-heidelberg.de*

Impressum

Heidelberg University
Interdisciplinary Center for Scientific Computing (IWR)
Engineering Mathematics and Computing Lab (EMCL)

Im Neuenheimer Feld 205,
69120 Heidelberg
Germany

Vtable hijacking: Object Type Integrity for run-time type information

Marco Schröder, Stefan Machmeier, Vincent Heuveline

March 2, 2023

Abstract

Attackers try to hijack the control-flow of a victim's process by exploiting a run-time vulnerability. Vtable hijacking is a state-of-the-art technique adversaries use to conduct control-flow hijacking attacks. It abuses the reliance of language constructs related to polymorphism on dynamic type information. The Control Flow Integrity (CFI) security policy is a well-established solution designed to prevent attacks that corrupt the control-flow. Deployed defense mechanisms based on CFI are often generic, which means that they do not consider high-level programming language semantics. This makes them vulnerable to vtable hijacking attacks. Object Type Integrity (OTI) is an orthogonal security policy that specifically addresses vtable hijacking. CFIXX is a Clang compiler extension that enforces OTI in the context of dynamic dispatch, which prevents vtable hijacking in this setting. However, this extension does not enforce OTI in context of polymorphism. The contribution of this work is a practical implementation to enable OTI in the context of C++'s run-time type information for the `dynamic_cast` expressions and the `typeid` operator.

1 Introduction

Nowadays, applications such as browsers like Firefox, or Chrome are incorporated into our daily life. Adversaries try to exploit the absence of certain security checks of such applications on source code level. The Control Flow Integrity(CFI) has been established to mitigate these attacks, however, due to their generic approach it remains unresolved for high-level programming language semantics [Association, 2003, Davi et al., 2014]. The key concept is to alter the execution path intended by the programmer, called control-flow [Davi, 2015]. For instance, an attacker that manipulated, or redirected the control-flow tries to determine the next value of the program counter which allows them to arbitrarily modify the instructions that are executed next. Consequently, an attack could install a backdoor or access sensitive data. An important precondition for such an attack is an initial memory corruption such as a buffer overflow, or use after free (UaF) error [Davi, 2015].

Control-flow hijacking can be broadly categorized into code-injection and code-reuse attacks [Davi, 2015]. Code-injection pivots the execution path toward instructions injected into the program's address space and requires an initial memory corruption. Whereas a code-reuse attack creates a malicious piece of code by composing benign sequences of instructions already present in the address space. Due to the limitation by the target system's capabilities, as well as, defense mechanisms like data execution prevention (DEP), code-injection attacks remain infeasible in practice. In case of a successful attack, adversaries rely on code sequences, called *gadgets*, and can use them to form basic operations that result in malicious computation. This approach is also known as return-oriented programming (ROP) [Checkoway et al., 2010, Schuster et al., 2015]. However, the discovery of *gadgets* inside the victim application's address space can be obstructed through Address Space Layout Randomization (ASLR), making it opaque for adversaries to find such *gadgets* Snow et al. [2013]. It maps the memory segments of a process (the call stack, the heap, the executable code) in a secret, pseudo-random manner [Schuster et al., 2015]. Memory corruption errors in C/C++ programs are the most common source of vulnerabilities in today's systems other approaches, displaying the need for other approaches such as control-flow integrity Burow et al. [2017]. This draws motivation for the implementation of CFIXX.

In this work, we present an extension of CFIXX that enables OTI for polymorphic objects during dynamic dispatch and in the context of run-time type information (RTTI). First, in Section 2, we outline background knowledge about control-flow attacks, and present corresponding prevention mechanisms. Section 3 presents the original CFIXX implementation and gives motivation for our new adaption. Next, we draw the design of our implementation and give insides into the implementation in the upcoming section. Finally, Section 6 completes this report with a conclusion that summarizes the results and describes future work in this regard.

2 Background

As a primer for the discussion of the OTI implementation, we provide a short summary of important aspects of OTI. C++, being an object-oriented programming language, naturally offers support for polymorphism [Stroustrup, 2013]. At its core, polymorphism provides a single interface to entities of different types [Stroustrup, 2022]. C++ supports static (compile-time) polymorphism in the form of overloaded functions and templates. Additionally, it supports dynamic (run-time) polymorphism, which is error-prone to vtable hijacking [Stroustrup, 2022]. For instance, a derived class D that is accessed by the pointer $B*$ of their base class B is unknown at compile time.

Name binding is a part of the compilation process. Here, each identifier in the source code links to a programming object - variable or function. With respect to function calls, it identifies a call site with the corresponding functions' implementation. For virtual functions, the binding is postponed to run-time due to the function that calls the dependency on the dynamic type of the object.

In order to solve the problem of the selection of a function implementation at the virtual call site, we need dynamic dispatch [Stroustrup, 2013]. The function calls the correct implementation that depends on the dynamic type of the object. C++ compilers can implement dynamic dispatch through a mechanism consisting of two components: a virtual function table (vtable) and a virtual function table pointer (vtable ptr) [Bauer and Rossow, 2021]. The compiler creates for each polymorphic class, i.e. each class that defines a virtual function, a vtable [Bauer and Rossow, 2021]. Vtables consist of function pointers to the most derived implementation of each virtual function of the corresponding class [Bauer and Rossow, 2021]. Note that the vtable generates on a per-class basis (dynamic type) rather than on a per-object basis [Bauer and Rossow, 2021]. In addition, the compiler places a vtable ptr in all polymorphic class instances [Bauer and Rossow, 2021]. Each constructor assigns the address of the vtable that is specifically associated with that class to the vtable ptr [Bauer and Rossow, 2021]. Dynamic dispatch is accomplished by retrieving the function pointer to the implementation from the vtable and dereferencing it [Stroustrup, 2022].

Run-time type information is a mechanism that provides the dynamic type of a polymorphic object by a given base class pointer [Stroustrup, 2013]. Compilers can solve it with vtable and vtable ptrs like dynamic dispatch. Vtables being emitted on a per-class basis can be used to determine an object's dynamic type [Bauer and Rossow, 2021]. In addition, they are carrying information next to function pointers that unambiguously identify the associated class type. RTTI provides the `dynamic_cast` and the `typeid` operators for polymorphic objects [Stroustrup, 2022]. `typeid` takes a type, or expression and returns a reference to the `type_info` object representing the type, or the type of the expression. `type_info` objects are part of C++'s standard library. The compiler creates such an object for every type used in the program, this includes built-in and user-defined types. For non-polymorphic types, the compiler statically deduce the type `typeid`. For polymorphic types, however, `typeid` can only be resolved at run-time, as it must return the dynamic type.

Before using `dynamic_cast` expression, we need to understand the inheritance model of C++ that allows *upcast* and *downcast*. For Upcast, the cast progresses up the inheritance hierarchy and can be conducted via a `static_cast` expression whereas downcast resolves from a base class pointer type to a derived class pointer type. However, it is only valid if the specified derived class pointer type matches the dynamic type of the object pointed to by the base class pointer. The cast can be verified by comparing at run-time the specified destination type to the dynamic type of the casted object. Both expressions, `typeid` and `dynamic_cast`, are deduced by following the argument's vtable ptr and examining the `type_info` object referenced by the corresponding vtable.

2.1 Vtable Hijacking

The abuse of vtable and vtable ptr data structure is referred to vtable hijacking. Vtables are constant look-up tables emitted by the compiler and placed in read-only memory, thus, protected from unwilling alteration. Vtable ptrs inhabit polymorphic class instances that must fill writable memory pages due to their mutable semantics in C++ [Davi, 2015]. For instance, if attackers succeed in manipulating a vtable ptr, which is subsequently used to make a virtual function call, they divert the intended control flow of the program. Instead of the original vtable, the attacker-chosen data is interpreted as function pointers. By determining which data the hijacked virtual call invokes as a function pointer, the attacker also determines which instructions are ultimately executed. This type of dynamic dispatch corruption is a form of vtable hijacking [Davi, 2015]. Therefore, we assume that attackers can also exploit a corrupted vtable ptr used for RTTI. Due to DEP, vtable hijacking needs to rely on code-reuse attack patterns [Davi, 2015].

Counterfeit Object-oriented Programming (COOP) is a code-reuse vtable hijacking attack scheme introduced by Schuster et al. [2015]. COOP works by repeatedly calling existing virtual functions on fake objects, which are essentially data inserted by the attacker. COOPlus is a variant of COOP proposed by Chen et al. [2021]. By imitating the control-flow of benign C++ applications even more closely than COOP does, it can even bypass CFI-based defenses that take C++ semantics into account, such as SafeDispatch. SafeDispatch is a CFI-based defense mechanism suggested by Jang et al. [2014]. It is specially tailored towards vtable hijacking. Generic CFI approaches are generally not aware of C++ semantics, they usually only restrict virtual call sites to address-taken functions [Jang et al., 2014]. In contrast, SafeDispatch focuses exclusively on control-flow transfers related to dynamic dispatch [Jang et al., 2014].

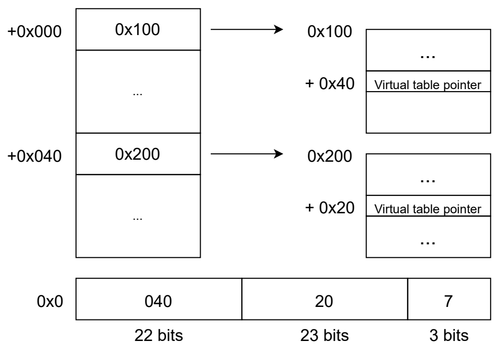
2.2 Control-Flow Integrity

First proposed by [Abadi et al., 2009, Banach and Lau, 2005], the key idea behind CFI is the protection of control-flow transfers with dynamically determined targets. For CFI in order to work on a system, it must meet two criteria [Abadi et al., 2009] First, the protected program's source code must be immutable. Second, a mechanism preventing code-injection, such as DEP, must be in place. CFI works in two phases [Abadi et al., 2009]: (1) during compilation the program's control-flow graph (CFG) is constructed, and (2) the source is instrumented to perform run-time checks. The CFG consists of nodes representing basic blocks (code sections without branches) and edges representing execution paths [Abadi et al., 2009]. Static analysis either on the program's source code or the compiled binary identify execution paths. Each edge is a valid run-time control-flow transfer. At run-time inserted instrumentation validates that the control-flow is aligned with the graph. After the CFG is constructed, run-time checks on the source-code level assert the compliance of dynamic control-flow transfers with the CFG. The core idea is to link the CFG to the source code to check whether a node that initiates a control-flow transfer is connected to the node that is the destination of the transfer. For instance, verifying the integrity of the control-path could be realized by labeling each node with a unique integer and performing a membership test to check if the integer is included in a set of valid targets. If it is not, control-flow information must have been corrupted.

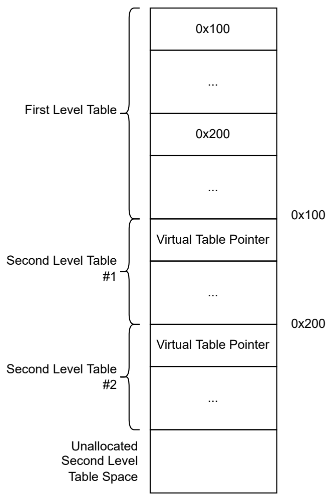
The quality of a given CFI implementation is influenced by the quality of the static analysis used to construct the CFG [Abadi et al., 2009, Banach and Lau, 2005, Jang et al., 2014]. In general, static analysis based on source code is more accurate than static analysis based on compiled binaries. Numerous CFI implementations have been proposed, as there has been a decade of research [Burow et al., 2017].

2.3 Object Type Integrity

Object Type Integrity (OTI) operates at run-time [Burow et al., 2018]. Unlike CFI, it can incorporate the run-time information to ensure that each polymorphic object instance is associated with its legitimate (dynamic) class type [Burow et al., 2018]. In order to mitigate the corruption of the vtable ptr which is equivalent to altering the dynamic type of the object, OTI tracks and applies integrity protection to the dynamic type of all polymorphic objects allocated [Burow et al., 2018]. In practice, this is enforced by using the correct, i.e. the compiler-assigned, vtable ptr. For instance, counterfeit objects which are not assigned a vtable ptr by compiled emitted code, thus, are not being associated with a type by OTI resulting in them being recognized as non-legitimate objects. In comparison to CFI, OTI is orthogonal due to the protection of the integrity of the control-flow information whereas CFI enforces the transfer of the control-flow [Burow et al., 2018]. A practical implementation of OTI is CFIXX by Burow et al. [2018].



(a) Logical Structure of Metadata



(b) CFIXX Metadata Memory Layout

Figure 1: CFIXX metadata table structure and memory layout (derived from Burow et al. [2018]).

3 Related Work

CFIXX, proposed by Burow et al. [2018], is a defense mechanism against *vtable* hijacking. It is implemented as a Clang compiler extension and guarantees integrity protection of the dynamic type of all polymorphic objects during dynamic dispatch. This is accomplished by preserving the vtable ptrs of these objects in a separate write-protected metadata table. In essence, this key-value store is a hash table that holds the vtable ptrs on a per-object basis. For instance, each polymorphic object has a copy of its vtable ptr in a table entry. If an attacker tries to exploit the vtable and a corrupted dynamic dispatch occurs, CFIXX will not consider their vtable ptrs to be legitimate as they are not compiler-assigned. This concept offers the opportunity to dispatch function calls with the vtable ptr from the metadata table. CFIXX protects the metadata table from unintended alteration by using the Intel Memory Protection Extensions (MPX).

Due to the invariance of an object's type, the vtable has a fixed location in the processes' address space and the vtable ptrs are constant Burow et al. [2018]. During the object construction, CFIXX only needs to record an object's vtable ptr once Burow et al. [2018]. Any write access to a vtable ptr compromises the object's type integrity and results in a corruption of dynamic dispatch Burow et al. [2018]. CFIXX verifies the validity of dynamic dispatch by looking up the function pointer via a copy of the vtable ptr that the metadata table stores, and thus, guaranteed to be uncompromised [Burow et al., 2018]. Apart from that, the procedure remains unchanged. Next to the dynamic dispatch, two further mechanisms depend on the vtable and vtable ptr; the `typeid` operator and `dynamic_cast` expression (also called RTTI). However, the original implementation by Burow et al. [2018] does not focus on RTTI. The contribution of this work is the extension of CFIXX to protect the entire RTTI system (see Section 5).

Technically, CFIXX extends the Clang 3.9.1 compiler and is targeting *x86_64* systems using the C++ *Itanium ABI* exclusively [Burow et al., 2018]. The extension mainly modifies the code generation of constructors and virtual function calls. The compiler run-time library is also adapted to set up the metadata table during process start-up. The metadata table must be allocated before any polymorphic object is constructed. The allocation is specified in the `.preinit`-array section of the ELF file format, which is used on x86 Unix systems. The metadata table is memory mapped and is a two-level hash table. High-order bits of the pointer used as the key is the index into the table's first level, which contains pointers to the second-level tables. These are indexed by low-order bits of the pointer used as the key and contain the actual value – the corresponding vtable ptr.

Figure 1 illustrates the logical structure and physical memory layout of the table. Figure 1a shows the logical structure, which consists of two levels. Pointers in *x86_64* systems are 64 bits wide, however, the virtual memory addresses consist of only 48 bits. The keys in the metadata table are either the address of the object instance or the address of a vtable ptr inside the object instance in cases of multiple vtable ptrs per object and are 48 bits wide. The first 22 bits serve as the first-level index of the table. The next 23

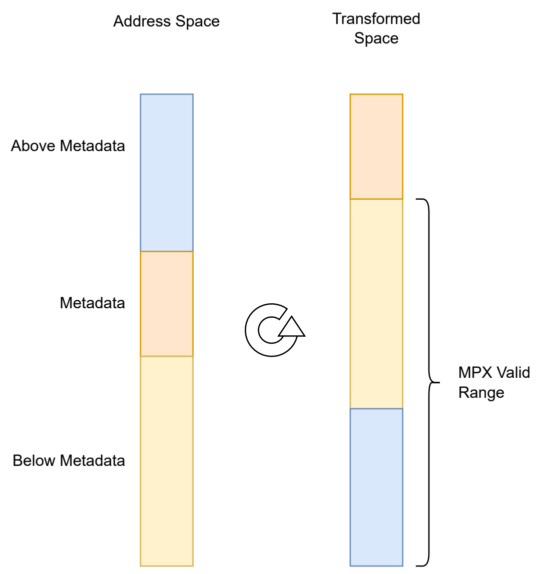


Figure 2: CFIXX address space rotation (derived from Burow et al. [2018]).

bits are the second-level index. Since every polymorphic object instance's memory layout must include at least one vtable ptr, the size of these instances is at least 64 bits. For this reason, the last 3 bits in the key do not contain relevant information and are discarded. The memory allocated during start-up is contiguous in the address space and sufficiently large to contain the first-level table and a certain number of second-level tables. The operating system's virtual memory management only allocates pages when they are accessed for the first time. CFIXX only touches the tables when needed, therefore, optimizing its memory usage [Burow et al., 2018]. The data structure is two levels deep resulting from the analysis of allocation patterns of polymorphic objects, which was conducted by the authors during development. High- and mid-level bits of polymorphic objects' memory addresses did not contain enough entropy to justify three levels.

CFIXX's enforcement of OTI is tied to the integrity of the metadata table [Burow et al., 2018]. Any kind of corruption must be prevented. A possible protection approach is information hiding. The CFIXX's authors decided to make use of the MPX instruction set extension to warrant the table's intactness [Oleksenko et al., 2018]. MPX provides an efficient bounds check of a pointer. For instance, there is a special instruction verifying that a pointer is used for writing access points inside some lower and upper bound. CFIXX modifies the code generation and ensures that only its own instrumentation is allowed to modify the metadata table [Burow et al., 2018]. This prevents attackers from bypassing the extension.

Figure 2 outlines an address space rotation. The left side shows the normal memory layout. The metadata table (colored brown) is located in between other data (colored blue and yellow). In order to ensure that the metadata is not written, the MPX instructions are used to verify that the write target is either above or below the table. As these two valid ranges are not contiguous, a two-bounds check is necessary. CFIXX rotates the address space for that reason [Burow et al., 2018]. The result is shown on the right-hand side of Figure 2. Since the valid area to be written is contiguous in the transformed address space, a single check is sufficient.

4 Vtable Hijacking in context of run-time type information

In this section we present how vtable hijacking is achieved by abusing the RTTI language constructs.

The implementation of `dynamic_cast` expressions and the `typeid` operator relies on the vtable data structure. Both constructs determine their argument's dynamic type by means of the `type_info` object referenced by the corresponding vtable. Consequently, passing a polymorphic class instance with a corrupted vtable ptr to either of them results in an invalid type cast, or the return of a wrong `type_info` object, respectively. Which, then again, is highly likely to cause an unintended control-flow transfer, that can be further abused. The fundamental principle behind vtable hijacking attacks initiated through RTTI is the generation of out-of-bound memory access. Further exploitation involves a grand scheme of the attack, similar to COOPlus described in Section 2. The out-of-bound memory access is introduced by the invalid inference and subsequent usage of a wrong dynamic type for a polymorphic object.

For instance, the `dynamic_cast` expressions are commonly used to handle special cases in functions operating on otherwise generic class interfaces. If a polymorphic object instance that contains a corrupted vtable ptr is passed to such functions, differing memory layouts between distinct child classes of the generic class interface are highly likely to result in out-of-bound memory access. On the other hand, the `typeid` operator is commonly used to implement multiple (dynamic) dispatch. The returned `type_info` object provides a hash value uniquely identifying the referenced type. It can serve as an index into a custom vtable data structure. In such cases, returning the wrong `type_info` object has the same implications as resolving C++'s built-in dynamic dispatch through a corrupted vtable ptr, which allows for attacks like COOP, that was discussed in Section 2. The accompanying prototype implementation contains example code showcasing how this could look like in practice.

Bottom line, attackers can not only achieve vtable hijacking by exploiting virtual function calls, but also via the C++ RTTI system. While the first named is the most frequent vtable use case, and hence, facilitates the majority of vtable hijacking, exploitation of `dynamic_cast` or `typeid` poses a serious security risk as well.

4.1 Exploiting dynamic_cast expressions

This section illustrates how vtable hijacking could be carried out via a `dynamic_cast` expression. In the example scenario the application is in need of text based identifiers, like URLs, or DOIs. Listing 1 depicts a class hierarchy modelling these identifiers. The abstract class *Identifier* defines an interface common to all identifiers. This includes a pure virtual function *assign* that takes a C string literal as argument (l. 3). However, the implementation is unique for each concrete identifier, as it depends on the specific data structure in which the content of the identifier is stored.

DefaultIdentifier is a class representing the standard identifier which uses a simple string data structure internally. It consists of the member size, representing the string's size, and the member *data*, that points to a `char` buffer storing the actual string (l. 8-9).

DefaultIdentifier::assign copies size characters from the argument string *s* into the buffer pointed to by *data* (l. 12-15). The application also includes the class *SpecialIdentifier*. Such identifiers are fixed size, hence, the member *data* being a stack array of length 10 (l. 22). Additionally, *SpecialIdentifier* instances hold a member *flags* that stores metadata (l. 21). *SpecialIdentifier::setSpecificFlags* sets a certain flag combination (l. 25). Part of the application is a complicated algorithm that takes an arbitrary identifier as input. It is implemented in the function *g* that is showcased in Listing 2. In case the input *Identifier* instance is of dynamic type *SpecialIdentifier*, the specific flags value needs to be set by *setSpecificFlags* before the actual algorithm is executed. The function *g* checks whether this is the case by trying to perform a downcast to *SpecialIdentifier* at first, and, if successful, calls *setSpecificFlags* on the object (l. 2-5).

Listing 3 below depicts how the vtable hijacking could take place. A *DefaultIdentifier* instance *id* is created on the stack (l. 3). Later it is passed to the function *g* (l. 5). In between, however, the attacker manages to corrupt the vtable ptr of *id* (l. 4). He replaces it with a pointer referencing the vtable corresponding to the *SpecialIdentifier* class. As a consequence, when *g* is invoked on *id*, the downcast to *SpecialIdentifier* will succeed and *SpecialIdentifier::setSpecificFlags* is invoked on the *DefaultIdentifier* instance *id*. As member access is implemented as hardcoded offset from the object's base address, this causes the *size* member of *id*, that is wrongly interpreted as the flags member of a *SpecialIdentifier* instance, to be set to the constant `SOME_CONSTANT`. As a result, *id* stores the wrong size for its buffer after *g* was invoked on it. Should this invalid size be greater than the original size, a buffer-overread/write is provoked as soon as *size* is used to control buffer access the next time.

In the example code, after *g* concluded, the virtual function assign is called on *id* (l. 6). Note that, even though the vtable ptr of *id* is manipulated, *DefaultIdentifier::assign* is invoked, as this is not a virtual call site bound at run-time, but at compile-time, since the dynamic type of *id* is statically known at the call site.

The memory address that *DefaultIdentifier::assign* interprets as *size* now holds the value `SOME_CONSTANT` assigned to the *flags* member in *SpecialIdentifier::setSpecificFlags*. Given this new value is large enough, the execution of the for-loop in *DefaultIdentifier::assign* causes a buffer-overwrite of the array pointed to by *data*. This overflow is the result of the vtable hijacking via `dynamic_cast`. It can aid the overall attack by subsequently serving as basis for a further exploit.

4.2 Exploiting typeid operator

Vtable hijacking based on the `typeid` operator pursues the same goal as the `dynamic_cast` variant. It provokes an out-of-bounds memory access error when using a type in the wrong context. This results in a function operating on data whose memory layout is incompatible with the function's instructions.

The `typeid` operator can realize a multiple dispatch system. Recall how virtual function calls are bound to an implementation depending on the dynamic type of the object that serves as hidden argument - the *this* pointer. This process is called single (dynamic) dispatch. For instance, the selection of the correct function implementation should not only depend on the dynamic type of a single argument, but the dynamic type of two or more arguments. This is called multiple (dynamic) dispatch Stroustrup [2013]. While single dynamic dispatch comes built into C++ where the compiler takes care of selecting the correct function implementation at run-time, there is no built-in support for multiple dynamic dispatch. One way of realizing a multiple dynamic dispatch is successive single dispatch, like the Visitor pattern [Gamma, 1995]. Another way to implement multiple dispatch is the `typeid` operator. Given a function that operates on multiple polymorphic base class types, the idea is to create a mapping from tuples of concrete derived class types to specific function implementations. The multiple dynamic dispatch can then be realized by selecting the correct implementation via the mapping. The `type_info` object returned by `typeid` provides a unique hash code for each class typ, that can be used for this purpose. Thus, the mapping associating a given tuple of polymorphic class types with a function implementation can be implemented as a hash table.

Listing 4 shows a code example that achieves multiple dynamic dispatch for two polymorphic arguments. The function *f* (l. 23-27) takes two base class pointers of type *A* and *B*. Both *A* and *B* are inherited by two child classes *A1* and *A2*, or *B1* and *B2*, respectively. There is a unique implementation of *f* for each combination of these derived class types: *f_ a1_ b1*, *f_ a1_ b2*, *f_ a2_ b1*, *f_ a2_ b2* (l. 12-15). The hash table *h* stores the mapping from a pair of derived class types to their dedicated implementation (l. 17-21). The `typeid` operator returns `type_info` objects which provide the `hash_code` function to retrieve a hash unique to the class type. The `hash_combine` function (l. 2) combines the two hashes returned for each derived class pair. the `hash_combine` function (l. 2) returns a single unique hash that serves as key into *h*. The function *f* acts as wrapper for the implementations and is called by the user. *f* looks up the correct implementation in the hash table and delegates the call to the corresponding function (l. 25-26).

Listing 5 portrays how the vtable hijacking could take place. First, two class instances *A1* and *B2* are allocated. They are referenced by the pointers *pa* and *pb* respectively (l. 1-2). The attacker then manages to overwrite the vtable ptr of the object pointed to by *pa* with a pointer referencing *A2*'s vtable (l. 3). Consider the consequences of this overwrite, when the function *f* is later invoked on *pa* and *pb* (l. 4). This causes the `typeid` operator, which *f* applies to *pa*, to return the `type_info` object representing *A2*. Because of that, the hash table look-up yields a function pointer to the implementation `f_a2_b2` that takes arguments of type *A2* and *B2* exclusively. However, the actual data to which this implementation is subsequently applied represents an argument of type *A1*. This type mismatch is highly likely to cause out-of-bound memory access, such as the buffer overflow illustrated in the `dynamic_cast` vtable hijacking example, or other memory corruptions. This acts as a door opener for further exploits.

5 Preventing Vtable Hijacking in context of run-time type information

In this section we present our implementation that renders vtable hijacking infeasible. To accomplish our approach, we build up upon the CFIXX compiler extension that Burow et al. [2018] presented (see Section 3). The prototype implementation is a fork of the original CFIXX project[1]. The prototype is

[1]`https://github.com/HexHive/CFIXX`

available at GitHub[2] It is based on three major modifications:

1. OTI is extended to cover `dynamic_cast` expressions
2. OTI is extended to cover the `typeid` operator
3. The protection of the metadata table is altered

By means of 1. and 2., all use cases of vtables are protected and vtable hijacking is rendered infeasible in its entirety. 3. consists of the replacement of the now deprecated MPX instructions that were previously used to protect the metadata table's integrity with Intel's newer Memory Protection Keys (PKEYS) [Accardi et al., 2023].

5.1 Protecting dynamic_cast expressions

Our extension guarantees that the compiler-assigned vtable ptr is used to determine the dynamic type of the polymorphic object. It achieves this the same way CFIXX does in the context of dynamic dispatch. Instead of relying on the vtable ptr located in the object instance itself to determine the dynamic type of the object, our extension retrieves the associated metadata table entry, i.e., the integrity-protected vtable ptr.

The code generation for `dynamic_cast` expression is handled by *CGExprCXX.cpp* in Clang's source tree [Lattner and Topper, 2023]. The function *CodeGenFunction::EmitDynamicCast* checks whether the `dynamic_cast` can be resolved statically, and does so, if possible. For instance, this is the case if the argument is known to be a null pointer. Otherwise it invokes the ABI specific function *ItaniumCXXABI::EmitDynamicCastCall* which is located in *ItaniumCXXABI.cpp.* The compiler does not insert the implementation of `dynamic_cast` as instruction sequence but delegates it to the ABI-specific run-time library *libcxxabi.* The library includes a function called `__dynamic_cast` which is the actual implementation of the `dynamic_cast` expression. *ItaniumCXXABI::EmitDynamicCastCall* generates the call instructions that invoke `__dynamic_cast` at run-time. The function takes the "this" pointer of the object as first argument which is technically the address of the vtable ptr. To resolve the cast, it infers the dynamic type through the `type_info` object referenced by the vtable.

5.2 Protecting typeid expressions

The extension provides OTI for the `typeid` operator in the same way it does with `dynamic_cast` expressions. It guarantees that `typeid` when being passed as a polymorphic argument uses its compiler-assigned vtable ptr to determine its dynamic type. This is accomplished by retrieving the vtable ptr of objects in question via the metadata table. Other than that, the behaviour of `typeid` remains unchanged. The code generation for `typeid` is implemented in *CGExprCXX.cpp* which is in Clang's source tree. The function *CodeGenFunction::EmitCXXTypeidExpr* first checks if the operator's arguement is polymorphic. If not, it statically deduces the `type_info` object to be returned. Otherwise, it calls the funciton *EmitTypeidFromVTable.* It examines whether the argument is null and calls the ABI specific *ItaniumCXXABI::EmitTypeid* which is defined in *ItaniumCXXABI.cpp.* This function retrieves the `type_info` object to be returned by following the vtable ptr which is done by invoking *CodeGenFunction::GetVTablePtr* from *CGClass.cpp.* The original CFIXX extension provides *CodeGenFunction::GetVTablePtrCFIXX* in the same file which retrieves the vtable ptr by querying the metadata table. Our extension modifies *ItaniumCXXABI::EmitTypeid* to invoke *CodeGenFunction::GetVTablePtrCFIXX* instead of *CodeGenFunction::GetVTablePtr.* This is the only change required to enforce OTI in context of the `typeid` operator.

5.3 Modifications to metadata table protection

As described in Section 3, the original CFIXX extension protects the metadata table by means of MPX. At run-time the metadata table verifies that no memory write access targets the pages populated, unless it is a write access that was generated as part of polymorphic object construction to insert a new table entry. MPX has been deprecated since then and is not available on all future architectures [Intel Corporation,

[2] `https://github.com/MarcoSchroeder/vtable_hijacking`

2023]. For this reason, the metadata table protection through MPX has been replaced with a different mechanism. Protection of the metadata is essential for the extension to be practical. Otherwise, the attacker could realize vtable hijacking by manipulating the metadata entry associated with an object. This would lead to the same result as if the vtable ptr placed in the object instance is corrupted in contexts that are not protected by CFIXX. As stated by Burow et al. [2018], the Intel's Memory Protection Keys (PKEYS) would improve the protection. Our extension realizes this approach. PKEYS share most of their interface with *mprotect* but utilizes previously unused bits in page table entries to enforce access rights on individual page level [Accardi et al., 2023]

Our extension removes the x86 assembly code generation that inserts run-time checks to verify if no emitted instructions target the pages holding the metadata table. The changes CFIXX introduced to *X86AsmPrinter::EmitInstruction* located in *X86MCInstLower.cpp* are reverted Burow et al. [2018]. In addition, the assertion if MPX is available on the host system defined as *process_ specific_ init* and *process_ specific_ finish* in *mpxrt.c* has been removed. In consequence, the metadata table protection through MPX is removed.

CFIXX extends the compiler-rt run-time library with a *cfixx.c* which contains, among other functions, the function *cfixxInitialization*, which allocates the metadata table during process startup [Burow et al., 2018]. Our extension implements two new function pairs in *cfixx.c*:

1. *cfixxEnableMetadataWritesAll* and *cfixxDisableMetadataWritesAll*
2. *cfixxEnableMetadataWrites* and *cfixxDisableMetadataWrites*

The first pair enables or disables write access for the entire metadata table.

Listing 6 sets the length of the metadata table by the difference of the table's base address and the current table. This is essential due to the change in length when more second-level pages are allocated. Next, *mprotect* is invoked on the entire length of the table starting from the base address. Write access to the pages is enabled by passing the flag `PROT_WRITE`. If *mprotect* fails, the application will be terminated. The implementation of *cfixxDisableMetadataWritesAll* only differs in the access right `PROT_READ` passed to *mprotect.* This implementation uses the regular *mprotect* system call which is only used as a fallback if PKEYS are unavailable on the target system [Burow, 2018, Accardi et al., 2023]. *pkey_ mprotect* has the same effect as the regular *mprotect.* The interface differs in its fourth parameter *pkey* which is a protection key allocated at process start-up. The selection between *pkey_ mprotect* and regular *mprotect* is done through conditional compilation.

After allocating the metadata table via *mmap*, *cfixxInitialization* calls *cfixxEnableMetadataWritesAll* to protect the entire metadata table from corruption. Write access is only enabled briefly whenever the extension needs to legally insert a new entry. There are only two cases in which this occurs. First, during construction of polymorphic objects. Second, as outlined in Section 3, the compiler emits `type_info` objects, that are returned by the typeid operator. As these objects are polymorphic, they require an entry in the metadata table to be recognized as legal under CFIXX. As no regular constructor is called for them, CFIXX manually inserts entries for these objects [Burow et al., 2018]. Accordingly, write access is enabled in this context as well. The run-time overhead of *mprotect* is linear in the number of pages for which access rights are changed. Creating a new table entry touches two pages, specifically the page containing the first-level entry and the page holding the second-level entry. Altering access rights of all pages the metadata table inhabits by calling *cfixxEnableMetadataWritesAll* and its counterpart, would therefore be inefficient. For that reason, the function pair *cfixxEnableMetadataWrites* and *cfixxDisableMetadataWrites* are additionally provided. Both take the key as an argument and adjust access rights solely for the two pages that will be modified.

Listing 7 shows *cfixxEnableMetadataWrites* implementation. The function receives as argument the pointer representing the key associated with the table entry for which write access is to be enabled. It passes this pointer to the function *getMetadataAddresses*, which returns a structure containing the base addresses of the pages containing the first and second-level entries that have to be modified. The first-level entry might require modification if the key is used the first time and a new second-level page needs to be allocated. The second-level entry stores the vtable ptr and is written in any case. The rest of the function is similar to *cfixxEnableMetadataWritesAll.* Since two access rights are modified for two pages individually, two calls to *mprotect* are performed. They are passed the base address of the page, adr.level1 or adr.level2, the system's page size, which is returned by the *sysconf* system call, and the flag for write access `PROT_WRITE`. The counterpart *cfixxDisableMetadataWrites* differs only by the access right PROT_READ passed. As with the function pair operating on the entire metadata table, if PKEYS are available the system call *pkey_ mprotect* is selected instead of regular *mprotect* via conditional compilation.

The function *getMetadataAddresses* takes the pointer serving as key into the metadata table and returns the base addresses of the two pages, which contain the associated first and second-level entry. Listing 8 shows the implementation. The function returns a structure called *MetadataAddresses*, that consists of two pointers, one for the first, and one for the second-level entry. The function first calculates the first and second-level index based on the key it is passed. The computation follows the rules described in Section 3. Next, the address of the first-level entry is set, which is the start of the metadata table with the first-level index applied as offset. After that, the base address of the corresponding second-level table is retrieved, which is the first-level table entry. Lastly, the address of the second-level table entry is set. It's the base address of the second-level table, defined in the previous step with the second-level index applied as offset. The addresses of the first- and second-level table entries are stored as *level1* and *level2* respectively. For both, write access is supposed to be enabled. However, the *mprotect* and *pkey_ mprotect* system calls only operate on the base addresses of pages. Therefore, the base addresses of the corresponding pages need to be defined. To do that, the page size is determined via the *sysconf* system call. The page base address of a memory address can be set by dividing the address through the page size, which is an integral division, hence, the result is automatically rounded to the nearest smaller integer. By multiplying that result with the *pagesize* again, the base address is obtained. This formula is applied to the address of both the first- and second-level entry, and the result is returned.

The file *ItaniumCXXABI.cpp* in Clang's source tree contains the function *ItaniumRTTIBuilder::Build-TypeInfo*, in which the entries for the `type_info` objects are created [Lattner and Topper, 2023]. This is done by calling the compiler-rt run-time function *cfixxSetVTablePtr*, that is defined in *cfixx.c*. Our extension inserts calls to *cfixxEnableMetadataWrites* and *cfixxDisableMetadataWrites* after function entry and before function exit respectively. The second case is polymorphic object construction. The original CFIXX already modified the responsible code generation function *CodeGenFunction::InitializeVTablePointer*, which is located in *CGClass.cpp*, to insert a corresponding metadata table entry. Our extension inserts calls to *cfixxEnableMetadataWrites* and *cfixxDisableMetadataWrites* accordingly.

With all described modifications applied, the metadata table's integrity is protected from unintended modification by an adversary and vtable hijacking is prevented effectively.

6 Conclusion

In summary, vtable hijacking is a state-of-the-art technique leveraged by attackers to conduct code-reuse control-flow attacks. Defense mechanisms that enforce the well-established CFI policy can prevent control-flow hijacking effectively in theory. However, they regularly fail to do so when it comes to the subset of vtable hijacking attacks. CFI-based solutions are limited by the inaccuracy of their static analysis, which is caused by performance requirements and a lack of run-time information. CFIXX is an innovative approach that tackle the problem. While the corruption of dynamic dispatch of vtable hijacking has been prevented effectively, the corruption of RTTI is an open attack strategy that has not been covered yet. We have shown the precondition for the `dynamic_cast` expression and `typeid` operator in context of polymorphism of an successful exploit and how an adversary could realize such an attack. Moreover, we protected RTTI language construct by extending the compiler extension CFIXX. We also conducted the replacement of the deprecated MPX instructions with the newer PKEYS.

A Listings

Listing 1: dynamic_cast vtable hijacking attack code example I.

```
class Identifier {
    public:
        virtual void assign(char const *s) = 0;
    // ...
};

class DefaultIdentifier : Identifier {
    size_t size;
    char *data;

    public:
        void assign(char const *s) override {
            for (size_t i = 0; i < size; ++i) {
                data[i] = s[i];
            }
        }
    // ...
};

class SpecialIdentifier : public Identifier {
    size_t flags;
    char data[10];

    public:
        void setSpecificFlags() { flags = SOME_CONSTANT; }
    // ...
};
```

Listing 2: dynamic_cast vtable hijacking attack code example II.

```
void g(Identifier* s){
    SpecialIdentifier *p = dynamic_cast<SpecialIdentifier *>(s);
    if (p) {
        p -> setSpecialFlags();
    }

    // complicated algorithm ...
}
```

Listing 3: dynamic_cast vtable hijacking attack code example III.

```
const char* text = "some␣string";

DefaultIdentifier id;
// id's vtable ptr is replaced with a SpecialIdentifier vtable ptr
g(&id)
id.assign(text); // causes buffer overflow
```

Listing 4: typeid vtable hijacking attack code example I.

```
// combine two hashes to create a new hash
size_t hash_combines(size_t a, size_t b);

class A { /*...*/ };
class A1 : public A { /*...*/ };
class A2 : public A { /*...*/ };

class B { /*...*/ };
class B1 : public B { /*...*/ };
class B2 : public B { /*...*/ };

void f_a1_b1() { /*...*/ };
void f_a1_b2() { /*...*/ };
void f_a2_b1() { /*...*/ };
void f_a2_b2() { /*...*/ };

HashTable h;
h[hash_combine(typeid(A1).hash_code(), typeid(B1).hash_code())] = &f_a1_b1;
h[hash_combine(typeid(A1).hash_code(), typeid(B2).hash_code())] = &f_a1_b2;
h[hash_combine(typeid(A2).hash_code(), typeid(B1).hash_code())] = &f_a2_b1;
h[hash_combine(typeid(A2).hash_code(), typeid(B2).hash_code())] = &f_a2_b2;

void f(A *a, B *b)
{
    FctPtr fptr = h[hash_combine(typeid(a).hash_code(), typeid(b).hash_code())];
}
```

Listing 5: typeid vtable hijacking attack code example II.

```
A* pa = new A1;
B* pb = new B2;
// attacker corrupts pa's vtable ptr to reference A2's vtable
f(pa, pb);
```

Listing 6: enableMetadataWritesAll implementation

```c
void cfixxEnableMetadataWritesAll()
{
  size_t tableLength = (size_t)cfixxTableEnd - (size_t)cfixxLookupStart;
  if (mprotect((void *)cfixxLookupStart, tableLength, PROT_WRITE))
  {
    exit(cfixxExitError);
  }
}
```

Listing 7: enableMetadataWrites implementation

```c
cfixxEnableMetadataWrites(void *thisPtr)
{
  MetadataAddresses adr = getMetadataAddresses(thisPtr);
  if (mprotect(adr.level1, sysconf(_SC_PAGE_SIZE), PROT_WRITE))
  {
    exit(cfixxExitError);
  }

  if (mprotect(adr.level2, sysconf(_SC_PAGE_SIZE), PROT_WRITE))
  {
    exit(cfixxExitError);
  }
}
```

Listing 8: getMetadataAddresses implementation

```c
typedef struct MetadataAddresses {
  void *level1;
  void *level2;
} MetadataAddresses;

MetadataAddresses getMetadataAddresses(void* this_ptr) {
  size_t index1 = (size_t)this_ptr >> 26 & mask22;
  size_t index2 = (size_t)this_ptr >> 3 & mask23;
  size_t *level1 = (size_t*)(cfixxLookupStart + index1);
  size_t *level2_base = (size_t*)(((size_t*)cfixxLookupStart)[index1]);
  size_t *level2 = level2_base + index2;

  long pagesize = sysconf(_SC_PAGE_SIZE);
  MetadataAddresses ret;
  ret.level1 = (void*)(pagesize * ((size_t)level1/pagesize));
  ret.level2 = (void*)(pagesize * ((size_t)level2/pagesize));
  return ret;
}
```

References

M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):1–40, Oct. 2009. ISSN 1094-9224, 1557-7406. doi: 10.1145/1609956.1609960. URL `https://dl.acm.org/doi/10.1145/1609956.1609960`.

K. Accardi, S. Khan, G. Kroah-Hartman, and J. Lee. Memory Protection Keys — The Linux Kernel documentation, Jan. 2023. URL `https://www.kernel.org/doc/html/latest/core-api/protection-keys.html`.

U. Association, editor. *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA XVII): October 26 - 31, 2003, San Diego, CA, USA*. USENIX Association, Berkeley, Calif, 2003. ISBN 978-1-931971-15-7. Meeting Name: Large Installation Systems Administration Conference.

R. Banach and K.-K. Lau. *Formal Methods and Software Engineering (vol. [3785): 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*. Number 3785 in Springer e-books. Springer-Verlag, Berlin Heidelberg, 2005. ISBN 978-3-540-32250-4.

M. Bauer and C. Rossow. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 650–666, Sept. 2021. doi: 10.1109/EuroSP51992.2021.00049.

N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 50(1):16:1–16:33, Apr. 2017. ISSN 0360-0300. doi: 10.1145/3054924. URL `https://doi.org/10.1145/3054924`.

N. Burow, D. McKee, S. A. Carr, and M. Payer. CFIXX: Object Type Integrity for C++. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society. ISBN 978-1-891562-49-5. doi: 10.14722/ndss.2018.23279. URL `https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_05A-2_Burow_paper.pdf`.

N. H. Burow. Taking back control: Closing the gap between C/C++ and machine semantics. 2018. URL `https://hexhive.epfl.ch/theses/18-burow-thesis.pdf`.

S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, Oct. 2010. Association for Computing Machinery. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866370. URL `https://doi.org/10.1145/1866307.1866370`.

K. Chen, C. Zhang, T. Yin, X. Chen, and L. Zhao. VScape: Assessing and Escaping Virtual Call Protections. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1719–1736. USENIX Association, Aug. 2021. ISBN 978-1-939133-24-3. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/chen-kaixiang`.

L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. Aug. 2014. URL `https://www.semanticscholar.org/paper/Stitching-the-Gadgets%3A-On-the-Ineffectiveness-of-Davi-Sadeghi/82b3346080783031ddaf73e6df73b5302ca19248`.

L. V. Davi. *Code-Reuse Attacks and Defenses*. PhD Thesis, Technische Universität, Darmstadt, Apr. 2015. URL `http://tuprints.ulb.tu-darmstadt.de/4622/`.

E. Gamma, editor. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass, 1995. ISBN 978-0-201-63361-0.

Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals, 2023. URL `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. URL `https://www.ndss-symposium.org/ndss2014/safedispatch-securing-c-virtual-calls-memory-corruption-attacks`.

C. Lattner and C. Topper. The LLVM Compiler, Jan. 2023. URL `https://github.com/llvm/llvm-project`. original-date: 2016-12-07T09:39:33Z.

O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28:1–28:30, June 2018. doi: 10.1145/3224423. URL `https://doi.org/10.1145/3224423`.

F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015. doi: 10.1109/SP.2015.51. ISSN: 2375-1207.

K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013. doi: 10.1109/SP.2013.45. ISSN: 1081-6011.

B. Stroustrup. *The C++ programming language*. Addison-Wesley, Upper Saddle River, NJ, fourth edition edition, 2013. ISBN 978-0-321-56384-2.

B. Stroustrup. *A tour of C++*. C++ in-depth series. Addison-Wesley, Boston, third edition, 2022. ISBN 978-0-13-681648-5.

Preprint Series of the Engineering Mathematics and Computing Lab

recent issues

No. 2021-02	Alejandra Jayme, Philipp D. Lösel, Joachim Fischer, Vincent Heuveline: Comparison of Machine Learning Methods for Predicting Employee Absences
No. 2021-01	Chen Song, Jonas Roller, Ana Victoria Ponce-Bobadilla, Nicolas Palacio-Escat,Julio Saez-Rodriguez, Vincent Heuveline: Spatial Effect on Separatrix of Two-Cell Systemand Parameter Sensitivity Analysis
No. 2020-01	Saskia Haupt, Nassim Fard-Rutherford, Philipp D. Lösel, Lars Grenacher, Arianeb Mehrabi, Vincent Heuveline: Mathematical Clustering Based on CrossSections in Medicine: Application to the Pancreatic Neck
No. 2019-02	Nils Schween, Nico Meyer-H übner, Philipp Gerstner, Vincent Heuveline: A time stepreduction method for Multi-Period Optimal Power Flow problems
No. 2019-01	Philipp Gerstner, Martin Baumann, Vincent Heuveline: Analysis of the StationaryThermal-Electro Hydrodynamic Boussinesq Equations
No. 2018-02	Simon Gawlok, Vincent Heuveline: Nested Schur-Complement Solver for a Low-MachNumber Model: Application to a Cyclone-Cyclone Interaction
No. 2018-01	David John, Michael Schick, Vincent Heuveline: Learning model discrepancy of anelectric motor with Bayesian inference
No. 2017-06	Simon Gawlok, Philipp Gerstner, Saskia Haupt, Vincent Heuveline, Jonas Kratzke, Philipp Lösel, Katrin Mang, Maraike Schmidtobreick, Nicolai Schoch, Nils Schween, Jonathan Schwegler, Chen Song, Marin Wlotzka: HiFlow3 Technical Report on Release 2.0
No. 2017-05	Nicolai Schoch, Vincent Heuveline: Towards an Intelligent Framework for Personalized Simulation-enhanced Surgery Assistance: Linking a Simulation Ontology to a Reinforcement Learning Algorithm for Calibration of Numerical Simulations
No. 2017-04	Martin Wlotzka, Thierry Morel, Andrea Piacentini, Vincent Heuveline: New features for advanced dynamic parallel communication routines in OpenPALM: Algorithms and documentation
No. 2017-03	Martin Wlotzka, Vincent Heuveline: An energy-efficient parallel multigrid method for multi-core CPU platforms and HPC clusters
No. 2017-02	Thomas Loderer, Vincent Heuveline: New sparsing approach for real-time simulations of stiff models on electronic control units
No. 2017-01	Chen Song, Markus Stoll, Kristina Giske, Rolf Bendl, Vincent Heuveline: Sparse Gridsfor quantifying motion uncertainties in biomechanical models of radiotherapy patients
No. 2016-02	Jonas Kratzke, Vincent Heuveline: An analytically solvable benchmark problem for fluid-structure interaction with uncertain parameters
No. 2016-01	Philipp Gerstner, Michael Schick, Vincent Heuveline, Nico Meyer-Hübner, Michael Suriyah, Thomas Leibfried, Viktor Slednev, Wolf Fichtner, Valentin Bertsch: A Domain Decomposition Approach for Solving Dynamic Optimal Power Flow Problems in Parallel with Application to the German Transmission Grid