MARX: Uncovering Class Hierarchies in C++ Programs

Andre Pawlowski*, Moritz Contag*, Victor van der Veen†, Chris Ouwehand†,
Thorsten Holz*, Herbert Bos†, Elias Athanasopoulos‡, and Cristiano Giuffrida†

* Horst Görtz Institut for IT-Security (HGI)
Ruhr-Universität Bochum, Germany
{andre.pawlowski, moritz.contag, thorsten.holz}@rub.de

† Computer Science Institute
Vrije Universiteit Amsterdam,
{vvdveen, herbertb, giuffrida}@cs.vu.nl,
{chris.ouwehand}@vu.nl

‡ Computer Science Department
University of Cyprus, Cyprus
eliasathan@cs.ucy.ac.cy

***Abstract*—Reverse engineering of binary executables is a difficult task which gets more involved by the way compilers translate high-level concepts used in paradigms such as object-oriented programming into native code, as it is the case for C++. Such code is harder to grasp than, e. g., traditional procedural code, since it is generally more verbose and adds complexity through features such as polymorphism or inheritance. Hence, a deep understanding of interactions between instantiated objects, their corresponding classes, and the connection between classes would vastly reduce the time it takes an analyst to understand the application. The growth in complexity in contemporary C++ applications only amplifies the effect.**

In this paper, we introduce *Marx*, an analysis framework to reconstruct class hierarchies of C++ programs and resolve virtual callsites. We have evaluated the results on a diverse set of large, real-world applications. Our experimental results show that our approach achieves a high precision (93.2% of the hierarchies reconstructed accurately for *Node.js*, 88.4% for *MySQL Server*) while keeping analysis times practical. Furthermore, we show that, despite any imprecision in the analysis, the derived information can be reliably used in classic software security hardening applications *without breaking programs*. We showcase this property for two applications built on top of the output of our framework: *vtable protection* and *type-safe object reuse*. This demonstrates that, in addition to traditional reverse engineering applications, *Marx* can aid in implementing concrete, valuable tools e. g., in the domain of exploit mitigations.

I. INTRODUCTION

Software exploitation has significantly increased in complexity and sophistication in recent years. Despite many attempts to harden applications, exploitation of vulnerabilities is still possible, especially for large and complex C/C++ programs, where attackers can leverage a rich environment of dynamically computed jumps. The targets of these branches are resolved only at runtime, and therefore they can be influenced for introducing new malicious control flows by taking advantage of software vulnerabilities. In contrast to C, C++, the choice for implementing a huge industrial software base [25], contains an additional source of indirect branches. While C programs need to resolve the target of a branch when, say, a function returns or a function pointer is used, C++ programs also need to support dynamic dispatching of virtual calls. Since virtual objects support several methods from different classes in their hierarchy, most compilers implement dynamic dispatching of virtual calls using indirect branches. In practice, C++ programs are thus full of indirect calls, and most of these can be influenced not just by overflow-type vulnerabilities, but also by temporal bugs (i. e., use-after-free vulnerabilities).

This plethora of indirect calls makes analyzing C++ binaries very important, since many exploits target exclusively C++ programs, but also significantly hard. For instance, according to a recent study [29], most libraries linked to Firefox contain almost 7% of indirect calls over direct calls and about 40% of them are virtual calls. Such indirect control-flow transfers rank among the greatest challenges for even the most basic analysis steps, such as the recovery of the control flow graph (CFG) [9], [24]. Resolving the targets of indirect calls and jumps in a binary is difficult. At the binary level, we have no way to directly learn class hierarchy information in the program. While we know that every virtual function call indexes a virtual function table (so called *vtable*), we neither know the vtables' exact locations, nor their relationships to each other. Reverse engineering such code from a given binary executable is therefore a very challenging task in practice.

Albeit challenging, vtable reconstruction directly from binaries can be useful in several domains. First, the class hierarchy helps the analysis of C++ legacy or closed code. Second, since vtables are commonly abused by exploits, security analysts can explore incidents affecting C++ applications when source code is not available. Finally, many defenses that harden C++ binaries can leverage the class hierarchy information for delivering sound protection of programs in the absence of source code. Current state-of-the-art binary-only protection approaches use weaker characteristics typical for C++ applications to protect virtual callsites, such as allowing all existing classes at a virtual callsite [21], or enforcing that the pointer to the vtable resides in read-only memory [13]. This stems from a lack of precision and scalability of current class hierarchy reconstruction approaches [12], [17], [18].

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23096

This means that, if we can successfully recover the class hierarchy from a binary, we can improve state-of-the-art binary-level defenses that can benefit from such information. For instance, we can ensure that virtual function calls conform to the class hierarchy, and therefore provide strong guarantees against attempts to hijack the control flow of the program (so called *vtable hijacking attacks*). Another example is to ensure that objects of different type classes are allocated from different memory pools to prevent the reuse of memory in a type-unsafe manner—a common source of use-after-free exploits. For both these example applications, extracting the class hierarchy of the binary program is important. Notice that this is *only* a set of mitigations that rely on C++ semantics, although an important one given that prior work argued that C++ binary-level defenses have trouble stopping control-flow hijacking attacks due to the lack of class hierarchy information [23].

In this paper, we consider the problem of reconstructing class relations directly from binaries. Our approach does not rely on embedded RTTI information (metadata emitted by the compiler for resolving class information at runtime, often stripped), does not rely on particular compiler flags, and works on industrial software. Since reconstructing class relations is a hard problem by itself and information concerning the direction of the relation is not available in binaries, we only focus on reconstructing class hierarchies as a set and ignore the direction of inheritance. Our system, *Marx*, can accurately reconstruct 93.2% of the hierarchies for Node.js and 88.4% of the hierarchies for MySQL Server. Overall, we have successfully applied *Marx* to more than 80 MiB of binary code to demonstrate the practicality of our implementation.

Marx is a valuable framework for the reverse engineering community, however, as we have already mentioned, security applications can leverage class relations for protecting binaries. The information provided by the analysis allows us to implement stronger binary-level defenses using type-based invariants. To showcase the practicality of *Marx*, we develop two binary-level defenses on top of it. The first application is a *vtable protection* system to prevent virtual calls to methods that do not belong to the class hierarchy and mitigate vtable hijacking attacks. The second application is a custom heap allocator to support *type-safe object reuse*, by placing newly allocated objects in memory pools based on their type.

Both security applications use the class hierarchy recovered from a binary. We demonstrate that, even when the extracted class hierarchy is imperfect, our defenses can improve security at reasonable performance and without breaking programs. To compensate for the imprecision of the analysis, our vtable protection solution treats invariant violations as anomalies and triggers more heavyweight checks on a slow path (trading off on performance). Our type-safe object reuse solution, in turn, can gracefully tolerate type-to-pool mapping mismatches (trading off on security). In short, we show that it is possible to build *fully conservative* binary-level defense solutions on top of imprecise information, exposing new interesting and previously unexplored tradeoffs.

Contributions. In summary, the contributions of this paper are as follows:

1) We design and implement *Marx*, a framework for reconstructing class hierarchies directly from binary executables that do not embed RTTI information, and are produced with arbitrary compiler flags. *Marx* is freely available at https://github.com/RUB-SysSec/Marx.
2) We evaluate *Marx* with more than 80 MiB of binary code and we show that vtables can be reconstructed from binaries with high precision. As an example, *Marx* can accurately reconstruct 93.2% of the hierarchies for Node.js and 88.4% of the hierarchies for MySQL Server.
3) We develop two security applications for binaries based on class hierarchies exported by *Marx*: *vtable protection* and *type-safe object reuse*. Our applications show it is possible to tolerate imprecise information when building sound binary-level defense solutions by trading off on performance and security.

II. Technical Background

Given that *Marx* is applied to C++ binaries for extracting the class hierarchy, some basic knowledge of C++ internals is required for understanding the mechanics of our analysis. Therefore, we discuss in this section some fundamental C++ concepts and how they are implemented in modern compilers.

A. Object-oriented Programming

C++ is an object-oriented programming (OOP) language which compiles to native code. In OOP, *classes* are data types used to instantiate concrete *objects*. On the latter, one can call *functions* (also called *member functions* or *methods*) as defined by the class an object was instantiated from. Apart from functions, classes can also define *attributes* and hence couple code (via functions) and data (via attributes) together.

OOP allows classes to *inherit* functions and attributes from other classes. This defines a relation. The class providing functions and attributes to another class is commonly called the *base* class in that relation, whereas the class inheriting these is called the *derived* class. This concept leads to what is called a *class hierarchy*: Every class is related to either zero or multiple bases as well as zero or multiple derived classes. Class hierarchies can be depicted as a directed graph in which the inheritance relation is given by the direction (base or derived class). If a class inherits from multiple base classes, this is referred to as *multiple inheritance*; otherwise, it is called *single inheritance*.

Classes may add several modifiers to their functions. One of the most important throughout this paper is the `virtual` modifier. If this modifier is used on a function, a derived class is free to *override* said function and provide its own implementation. This concept is called *polymorphism*, where a single function invocation may behave differently, depending on the context in which it is called. More concretely, the programmer can call a virtual function on *either* an object of the base or any derived classes. Depending on the type of the object, the appropriate implementation of the function is called. This, in turn, allows programmers to work on the most general class and vastly simplify their code. In cases where the compiler cannot determine statically on which object the function is to be invoked, the selection of the appropriate implementation is made at runtime. Furthermore, *abstract*

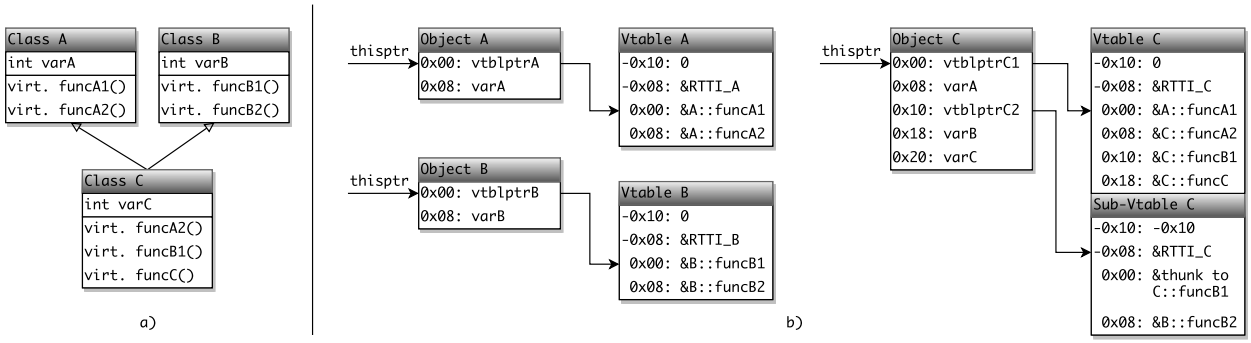

Fig. 1. Example hierarchy layout shown at a high-level in a) and b) shows its layout in native code. Class `C` inherits attributes and functions of both class `A` and `B`. Further, class `C` overrides `funcA2` and `funcB1` and provides its own implementation, `C::funcA2` and `C::funcB1`.

base classes represent an edge case: They provide several virtual functions, but *no* implementation on their own (*pure virtual* functions). This forces the deriving class to implement functions conforming to the declaration the base class provides.

Figure 1 a) depicts an exemplary relation of three classes `A`, `B`, and `C`. Class `C` inherits the functions from classes `A` and `B`, i. e., one may call the functions `funcA1`, `funcA2`, `funcB1`, `funcB2` on an object of class `C`, in addition to the functions class `C` provides itself. The same is true for the attributes; hence, class `C` allocates space for attributes `varA`, `varB`, and `varC`. Further, `C` overrides `funcA2` and `funcB1`, i. e., it specifies a more fitting implementation for its class.

In order to create an object of a specific class, the operator `new` can be used, amongst others. It allocates space for the object (whose size is mostly determined by its attributes) and calls a designated initialization function that initializes the object's attributes with meaningful values. This function is called a *constructor*. Similarly, a *destructor* releases further resources the constructor requested previously, and is usually invoked through the operator `delete`.

In the following, we explain how the aforementioned concepts are implemented on the binary level.

B. Virtual Function Tables

On the binary level, polymorphism is implemented with the help of what is called a *virtual function table* (*vtable* for short). It contains the addresses of all virtual functions a class provides. Each object of such a class contains a pointer to the corresponding vtable. In the following, we refer to this pointer as *vtblptr*.

In the Itanium C++ ABI [3], two metadata fields in the vtable are specified: *Runtime Type Identification* (RTTI) and *Offset-to-Top*. The *RTTI* field holds a pointer to a data structure in which metadata about the class resides, i. e., the name of the class and its base classes. Even though this information is useful for type reconstruction, it may not be available in compiled binaries. It only has to be included if, e. g., `dynamic_cast` or `type_info` is used, which requires precise type information at runtime. The *Offset-to-Top* field holds an offset that is required when implementing multiple inheritance. To this end, it is used to adjust an object pointer, as discussed in a later section.

C. Virtual Function Dispatch

As opposed to regular functions (which are implemented using direct calls), virtual function calls require a specific type of callsite (*virtual callsite*, or *vcall*). They handle the selection of the proper virtual function depending on the object on which the function is invoked using the object's vtable.

Consider a virtual callsite invoking `funcA2` on an object of either class `A` or `C`. Independent of the class the object at the callsite is instantiated from, in this case, one merely has to call whatever function is referenced at offset `0x08` in the object's vtable. As seen in Figure 1 b), this offset either points to `A::funcA2` or `C::funcA2` and always calls the correct implementation for the given object. Note that this offset has to be the same across all related vtables. In this case, this constraint applies for vtable `A` and `C`, as classes `A` and `C` are the only candidates when invoking function `funcA2`. This mechanism effectively implements polymorphism at the binary level. In the following, we will refer to the pointer to the current object as *thisptr*.

The compiler emits code that directly implements this mechanism. At each *vcall*, the *thisptr* to the object is also set as an implicit argument (meaning the argument is not specifically set in the source code). Depending on the calling convention, the *thisptr* is either stored in a specific register or on the stack. In the Itanium C++ ABI on x86-64, a *vcall* always has the following structure:

```
mov RDI, thisptr
call [vtblptr + x]
```

The *thisptr* is stored in the `RDI` register as the first argument and the *vtblptr* is used to select the correct virtual table. The value `x` denotes the offset into the selected vtable in order to branch to the correct function. Note that it may be zero or omitted if the first function of the vtable is targeted.

D. Multiple Inheritance

In addition to single inheritance, C++ supports multiple inheritance. This allows a class to have multiple base classes from which it inherits functions and attributes. In the example given in Figure 1, class `C` uses multiple inheritance and derives from class `A` and `B`.

Considering the way virtual calls are dispatched, it becomes apparent that vtables are inherited as well. Given that the

dispatching mechanism targets certain offsets in the vtable, the order of the functions must be preserved throughout the hierarchy. Conceptually, for multiple base classes, multiple vtables have to be inherited.

In the given example, class `C` has a copy of the vtable of class `A` with a modified entry (pointer to function `C::funcA2` instead of `A::funcA2`) and appends pointers to its own implementations of virtual functions to it. Further, a modified copy of the vtable of class `B` is added as a sub-vtable of class `C`. In this sub-vtable, only the function entries of *overridden* virtual functions have changed. These entries have been replaced by a pointer to special functions called *thunks*. If a class derives from only one base class, both vtables can be merged without conflicts – the current class simply uses higher offsets when accessing its part of the vtable. In case of multiple inheritance, however, *sub-vtables* have to be used. When accessing an object of class `C` as an instance of class `B`, the *thisptr* has to be increased by `0x10`. Hence, the layout matches the one expected by vcalls of class `B`. Thunks are used to call back into virtual functions belonging to class `C`.

In the given example, `funcB1` was overwritten by class `C`. At the position of the entry of *funcB1* in the sub-vtable, the pointer to the thunk, `thunk to C::funcB1`, has been written. When executing this thunk, the *thisptr* is modified to point to the beginning of the object `C` and invokes `funcB1` of class `C`. This ensures that the function uses the correct offsets into the object, i. e., the *thisptr* points to the start of object `C`.

The *Offset-to-Top* field of the sub-vtables holds the value that has to be added to the *thisptr* to reach the beginning of the object. In our example, the *Offset-to-Top* holds the value `-0x10`. When the sub-vtable of object `C` is used, the *thisptr* points to offset `0x10` of the object (the *vtblptr* of the sub-vtable). When adding the value of the *Offset-to-Top* field, the *thisptr* points to the beginning of object `C`. More details on multiple inheritance can be found in the C++ ABI documentation [3].

E. Object Creation and Destruction

As mentioned in Section II-A, constructors are used for initializing the memory area previously allocated to hold a specific object. Additionally to any initialization performed by the programmer (such as setting default values for object attributes), the compiler adds statements to set an object's vtable pointer(s). In case of inheritance, constructors are called *top-down*, i. e., the base class constructor is executed prior to the constructor of the derived class.

Analogous principles apply to the destructor of a class. However, destructors are invoked in reverse order (*bottom-up*, invoking the most specific destructor first).

III. Analysis Approach

Given a binary executable, we aim at extracting the C++ class hierarchies as accurately as possible. To this end, we extract distinct properties that result from the way a C++ compiler implements the aforementioned high-level concepts on the binary level. In the following, we describe the design of our approach that is implemented in a tool called *Marx*.

Generally speaking, our analysis is divided into two steps:

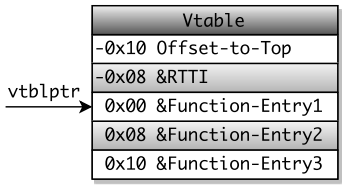

Fig. 2. Structure of an Itanium C++ ABI vtable. The vtable pointer referenced in the code points to offset 0, where the table of function pointers starts. The two metadata fields (RTTI and Offset-to-Top) precede said table.

1) *Vtable Extraction.* Distinct patterns that are typical for vtables in the code are searched and information about the vtables is extracted.
2) *Static Analysis.* Given that these heuristics might lead to an overestimation, a static analysis of the code is conducted which searches for usages of the vtables found in the previous step.

In the following, we focus on the Itanium C++ ABI [3]. However, we stress that the presented methodology is applicable to other ABIs as well, such as the ARM [6] or the Microsoft C++ ABI [14].

A. Vtable Extraction

Virtual function tables in a binary are the key element to our analysis. By extracting usages of vtables, one easily finds points in the program where objects are either created (constructors) or destroyed (destructors). This, in turn, yields valuable information about the relation of different classes. Hence, one can view vtables as roughly analogous to a certain class and relation of vtables as corresponding to certain class hierarchies.

Our analysis applies multiple heuristics in order to locate vtables in a binary (**H-1** to **H-6**), as we discuss in the following. A rough, albeit simple, estimate can be used to restrict the search space to specific sections: As vtables are fully specified at compile time, they can be placed in *read-only* sections (heuristic **H-1**). Therefore, only those sections that typically hold vtables, such as `.rodata`, `.data.rel.ro`, and `.data.rel.ro.local`, are analyzed.

Figure 2 shows the typical structure of an Itanium C++ ABI vtable. A vtable consists of three different parts: the *Offset-to-Top* field, the *RTTI* pointer, and multiple *function entries*. Each type has different properties which can be used to distinguish between them. Also, their order is fixed, which makes it possible to search for a series of consecutive patterns in a memory range, e. g., a specific section.

As seen in the figure, the *vtblptr* references the beginning of the function entries. Usually, this reference will be used in a constructor or destructor. However, we note that the other fields are usually not referenced at all (heuristic **H-2**).

Offset-to-Top is used to implement multiple inheritance for objects and encodes the offset from the sub-object to the base object. It is a mandatory field and always contains `0` if multiple inheritance is unused. Our approach checks the sanity of this entry by only allowing values in the range from `-0xFFFFFF` to `0xFFFFFF` as proposed by Prakash et al. [21]. In addition,

the value cannot be a relocation entry and these checks constitute heuristic **H-3**.

RTTI holds a pointer to further type information for the class. Since this field is optional, the entry is either a pointer or `0`. If the entry is, in fact, a pointer to *data*, it has to point to non-executable memory (heuristic **H-4**). Since the vtable can be part of a shared library, this entry can also be a relocation entry.

Function entries hold a pointer to the virtual functions the class provides. Hence, an entry either points to the `.text`, `.plt`, or the `.extern` section of the binary, or it is a relocation entry. One of these properties has to be satisfied such that the analysis deems the function pointer to be valid (heuristic **H-5**).

In rare cases, the compiler sets the first few function entries of the vtable to `0`. This can happen for multiple inheritance constructs inheriting from abstract classes. To cope with these edge cases, our approach allows the first two function entries of the vtable to be `0`. This number was empirically found to be sufficient (relaxing heuristic **H-6**).

Finally, we can determine the beginning of a vtable by searching three consecutive words in memory that fulfill the properties outlined above. Further, the length of the vtable can be estimated by checking the subsequent function entries for validity.

To sum up, the heuristics we employ are:

- **H-1** Vtables have to lie in read-only sections.
- **H-2** In a candidate vtable, only the beginning of the function entries is referenced from the code.
- **H-3** *Offset-to-Top* lies within a well-defined range and it is no relocation entry.
- **H-4** *RTTI* either points into a *data* section or is `0`.
- **H-5** A function entry points into a *code* section or is a relocation entry.
- **H-6** (relaxing) The first two function entries may be `0`.

Note that the heuristics to find these patterns can lead to an overestimation of extracted vtables. Nevertheless, this does not impact the subsequent analysis notably since only *existing* vtables are referenced in the code (cf. heuristic **H-2**). We note that only in rare cases an overestimated vtable can result in an overestimated hierarchy. On the other hand, only an underestimation of vtables would lower the precision of the analysis, which is unlikely for the presented approach.

B. Static Analysis

Now that we obtained a possibly overestimated set of vtable candidates, the second phase of our approach statically analyzes their relation based on indicators found in the binary code. Eventually, it yields distinct sets of vtables that are part of a class hierarchy. In the following, we discuss the various indicators the approach uses.

1) Overwrite Analysis: In Section II-E, we discussed how during object creation, the constructor writes the *vtblptr* into the object. Further, when class `C` inherits from class `A`, as depicted in Figure 1, the constructor of class `A` is executed before the constructor of class `C` (top-down approach). This ensures that the inherited attributes of class `A` are initialized before the constructor of class `C` accesses them. Consequently, the *vtblptr* of the base class `A` is written into the new objects *before* the derived constructor writes the *vtblptr* of class `C` as shown in Figure 3. This also holds for multiple inheritance. In the given example, class `C` also inherits from class `B`. Hence, the constructor of class `B` is also executed before the constructor of class `C`. In this case, however, the *vtblptr* of class `B` is overwritten by the *vtblptr* to the sub-vtable of class `C`.

In contrast, during object deletion, the destructor of class `C` is executed before the destructor of class `A`, i. e., invocation follows a bottom-up approach. Therefore, the *vtblptr* of class `C` is overwritten by the *vtblptr* of class `A`. This also holds in the case of multiple inheritance for the sub-vtable of class `C` and *vtblptr* of class `B`, analogously. We can leverage this and detect the dependency of two classes by tracking if one *vtblptr* in an object is overwritten by another *vtblptr*. Remember that classes are roughly analogous to their vtable for our approach.

Naturally, this also means that we have to track the creation of a potential object by monitoring the `new` operators of the memory allocator. By correctly identifying constructors and destructors, our approach would also be able to make a statement about the *direction* of the inheritance, i. e., detect which class is the base and which the deriving class. In its current implementation, however, our approach reconstructs the class hierarchy as a plain set.

Due to compiler optimizations, constructors are often *inlined* next to the memory allocation of the object in the same function. The same concept is applied to destructors, analogously. While our approach does not identify constructors and destructors directly, it detects the characteristic pattern of *vtblptr* overwrites. As a result, we are able to detect overwrites for which a concrete classification as constructor or destructor is more involved. More precisely, our overwrite analysis is performed on statically calculated paths through multiple functions, as discussed in Section IV. Thus, we avoid having problems with function inlining as opposed to other approaches such as the one presented by Jin et al. [17].

2) Vtable Function Entries: Classes in the same hierarchy share attributes and a subset of virtual functions. Also, a class that inherits virtual functions from another class does not have to overwrite it. As a result, the vtables of both classes, base and derived, may contain multiple entries that point to the same function as in the other classes' vtable. In order to work in polymorphic constructs, this function entry has to be at the same position in the vtables.

Hence, we can employ a heuristic that checks if multiple vtables share the same function entry at the same position. If they do, we consider them as related. Obviously, specific entries like a `0` entry or the *pure virtual* function have to be excluded from this heuristic. Note that, similar to the overwrite analysis in its current form, no direction information is included. Naturally, if the compiler places the same function entry at the same position in unrelated vtables due to optimization passes, our analysis would find them to be related. This would lead to an overestimation of the found class hierarchy. However, the evaluation results in Section VI-A show that this case can be neglected in practice.

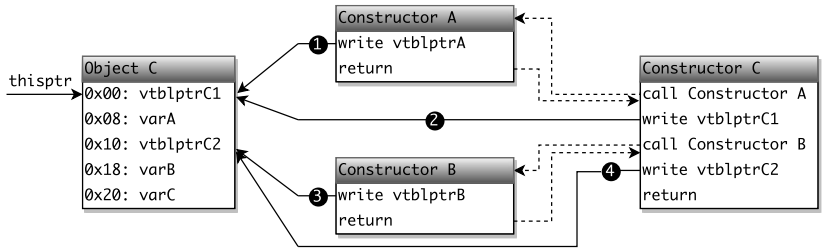

Fig. 3. Order of *vtblptr* overwrites during the creation of an object of class `C`. The control flow of the execution of `Constructor C` is depicted as dotted lines.

3) Inter-Procedural Data Flow: We perform our analysis on paths through multiple functions. Even if analysis within a function is well defined, special attention has to be paid to the point where function boundaries are traversed.

a) Forward Edge: Virtual functions are usually only called via an indirect call instruction. As these are dispatched dynamically based on a concrete object, the list of potential call targets is not easily retrieved statically. As the overwrite analysis analyzes paths through multiple functions, indirect callsites pose a roadblock and may prevent the analysis from following the call target. Consequently, vtable relations that are established beyond this point will be missed by the analysis. We say it lacks *context*, as no potential object at the callsite is known with which the callsite could be resolved.

In order to tackle this problem, the static analysis tries to resolve the indirect call instruction with the help of the context (essential a memory state) it built up on the current path. If the argument of an indirect call instruction is known, i. e., it dereferences a known *vtblptr*, we resolve the target function and continue the analysis in the newly discovered function on the path while keeping the current context.

As an additional side effect of resolving the branch targets at a *vcall* in the current analysis run, we know which vtable is used for it. Since in polymorphic constructs only classes within the same hierarchy are allowed, a *vcall* can only be used by objects of dependent classes. Hence, if during the analysis the same *vcall* is used by objects containing different vtables, these vtables are related (i. e., the objects are from the same class hierarchy). However, no information about the direction of the inheritance of the classes is obtained.

b) Backward Edge: In addition to passing known context on to the beginning of a more deeply nested function, the analysis also has to take return values of a callee back to its caller into account. Since different paths through the callee can result in multiple different return values, we generalize them into a set of return values, which is effectively the union of the individual return values on each path. Then, if the return value is used in a point where more context is required, such as a *vcall*, the information provided all possible return values can be used to, e. g., resolve an indirect callsite.

Consider the example given in Figure 4, which was encountered in the FileZilla FTP client. This function returns a different object depending on the given argument. The classes that are used to create the object (namely, `X`, `Y`, and `Z`) are part of the same hierarchy. Without tracking the possible return values of this function into a *vcall* of the caller, it is not possible to find the relation of these classes based on information of the forward edge alone.

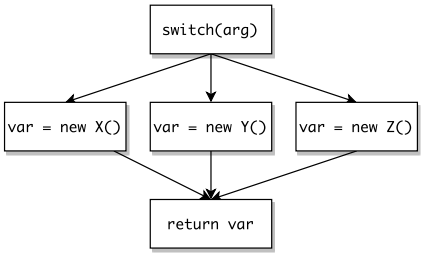

Fig. 4. A simplified version of the function `GetSortComparisonObject` from the FileZilla FTP client. Depending on the given argument, it creates a new object of different classes and returns it. Without considering the backward edge, no statement about the classes referenced in this function can be made.

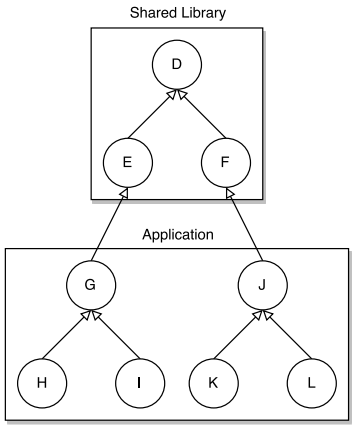

Fig. 5. A class hierarchy that is connected via a shared library. Classes `G` and `J` both inherit from classes belonging to the same hierarchy in a different module. This fact is only observable when taking module boundaries into account.

4) Inter-Modular Data Flow: Applications are commonly divided into multiple modules (also known as *libraries*), where each module performs a specific task. On Linux, these modules are implemented using *shared objects*, which already include the notion that common functionality can be reused by different applications. Obviously, modules can depend on each other. Specifically, in C++, it is possible to interact with classes exported by a shared object. Such relations would be hidden from our analysis when analyzing single modules only due to missing context. To counter this, we support inter-modular analyses.

Consider the example depicted in Figure 5. The application itself contains two class hierarchies that appear to be unrelated when looking at the main module. However, when taking the application's shared library into account, it becomes apparent that the two hierarchies are, indeed, related. Both hierarchies in the main module derive from a class of the very same hierarchy in the shared library. Isolated analysis of the individual modules would not have yielded the same result.

Our approach analyzes shared libraries first and creates data flow summaries of the return values and *vtblptr* overwrites in their respective modules. If a function in a shared library is called during the analysis phase, the aforementioned summaries are added to the current context accordingly. This way, we can also consider all class hierarchy connections that are outside of the currently analyzed module.

IV. Implementation

Based on the concepts presented in Section III, we implemented an analysis framework called *Marx* in C++ for Linux x86-64 binaries. Note that even though only Linux x86-64 binaries are supported as of now, the implementation can be easily extended to support more architectures. This is supported by the fact that we use VEX-IR from the Valgrind project [5] as our intermediate language.

In the following, we describe the implementation of *Marx*, discuss challenges we encountered, and explain how we solved them. To foster research on this topic, the source code of *Marx* is freely available at https://github.com/RUB-SysSec/Marx.

The first step of our analysis, vtable extraction, is performed on an IDA database using scripting facilities provided by IDAPython [2]. In addition to the candidate vtables identified via the heuristics H1 – H6 described in Section III-A, the control-flow graph (CFG) of all known functions is extracted as well, which is used in the subsequent static analysis step.

The static analysis is mostly driven by a data tracking engine which updates the context collected upon a path through the binary, as discussed in Section III-B. Said context is used to track assignments of vtables to new objects and their overwrites in constructors and destructors. In its current state, the engine implements basic 64-bit VEX instructions, as this is already sufficient for our needs. The constructs we want to identify hardly involve any complex calculations and with our focus on real-world applicability, we have to weigh up *Marx*' precision against its performance.

A. Starting Points and Context Sensitivity

Marx starts its analysis for each function in the target program separately, i. e., each function serves as starting point for at least one path. In order to obtain reasonable results for a specific path, however, enough *context* has to be known in which the function is executed. Otherwise, relations may be missed and the current traversal would not add to the overall results of the analysis.

There are multiple ways to ensure that the analysis visits a function g with a reasonable amount of context. For one, if g lies *deeper* within a path, it is reasonable that its caller f already adds vital information to the context. By starting an analysis path at f, the context added by it is available when arriving at g. If f constructs an object and g overwrites its vtable, this information would be missed by analyzing g out of context. This aspect is discussed further below.

On the other hand, it helps to consider the context a function *inherently* lies in. For example, g may be a virtual function. This, in turn, means that it belongs to (at least) one vtable. Hence, our analysis can be provided with some initial context: a *thisptr* exists and the object's *vtblptr* can be initialized to point to the vtable g lies in. For example, for the x86-64 Itanium C++ ABI, any occurrence of `[rdi + 0]` is then known to resolve to the current *vtblptr*.

This enables the analysis to handle operations on the object itself now that the target object is known (*vtblptr* overwrites or *vcall*s). Further, if g belongs to multiple vtables, g is analyzed in just as many contexts.

B. Path Creation and Convergence

The previous section already hinted on where possible paths through the binary *start*—at any known function. However, care has to be taken that a path *ends* at a point where all relations have been picked up by our analysis and no superfluous calculations are performed by further following the path. A naïve approach would be to simply analyze all possible paths at the given starting point. Yet, this leads to what is called the path explosion problem, as the number of paths easily exceeds a feasible amount for non-trivial CFGs.

Marx decides which paths through a function are worth analyzing by following a heuristic: on each basic block in the CFG, a predicate is run which decides whether the block is considered *interesting*. We consider those blocks as interesting that contain one or more of the following cases: (i) an indirect call (i.e., a possible *vcall*), (ii) a (direct) call to a `new` operator, or (iii) an instruction operating on a *vtblptr*. With this heuristic, we attempt to visit only those blocks that add to the overall context we are interested in.

We then compute paths that try to visit as many interesting basic blocks as possible before reaching an exit block. In order to avoid a high computational complexity, a threshold t is introduced. If a function contains more than t interesting basic blocks, paths are generated that guarantee to visit *at least* one interesting block, but no attempts are made to maximize this number. Empirically, we found a threshold of 20 interesting basic blocks to be sufficient for our purposes. By trying to visit multiple interesting basic blocks on a path within a function, overwrites in the very same function are more likely to be detected. This is, for example, the case for inlined constructors: one block may allocate memory, whereas another writes the object's vtable.

Loops are only traversed once, i. e., paths are guaranteed to visit every basic block at most once. Empirically, we did not encounter cases where loop unrolling would have yielded better results in terms of object creation coverage.

Up to now, we only considered paths within a certain function. However, such a path may contain calls to other functions. As already stated in the previous section, the call depth of a path impacts the amount of context available to our analysis, which, in turn, strongly impacts its results. Still, it is also an important factor to ensure that the analysis terminates

in a reasonable amount of time and, once again, illustrates how one has to weigh up performance and accuracy. Empirically, we determined that a maximum call depth of 2 is sufficient for analyzing large real-world applications. Note that, depending on time and resources available to an analyst, higher precision can be achieved by increasing this number.

C. Virtual Callsite Identification

Even though the information collected in these steps is already helpful for an analyst, we further refine the result and try to distinguish *vcall*s from other indirect call constructs and only resolve target sets for the former. To detect a *vcall*, *Marx* searches for the virtual function dispatch structure described in Section II-C. As this structure applies to both *vcall*s and other types of indirect calls (e. g., function pointers), we implemented two modes to resolve the targets: *conservative* and *non-conservative* mode.

In conservative mode, an indirect call is only identified as *vcall* if the *thisptr* holds a known object and a vtable is involved when computing the target address. This ensures that the analysis has an exact state for the *thisptr*.

In non-conservative mode, an indirect call is considered to be a *vcall* simply if the vtable is involved in the computation of the target address, i. e., we drop the requirement that the *thisptr* has to be valid. Due to missing context during the analysis (i. e., call depth is depleted), memory locations might not be identified as objects and therefore the *thisptr* check can fail. The non-conservative mode allows the analysis to resolve more *vcalls* since requirements are relaxed. However, overestimation can lead to a higher false positive rate. The difference between both modes is further evaluated in Section VI-B.

V. Applications

Beyond applications in the area of reverse engineering, the results of the reconstructed class hierarchies can also be used to significantly improve defenses that mitigate attacks against C++ applications. In this section, we present two protection approaches build on top of the analysis results provided by *Marx*: *vtable protection* and *type-safe object reuse*. In practice, a C++ application can be analyzed by *Marx* before deployment and then set up with the wanted protection.

A. VTable Protection for Binaries

VTable protection and, more generally, Control-Flow Integrity (CFI) [7] is a promising way to stop advanced code-reuse attacks. In its ideal form, it limits an attacker by enforcing that each indirect branch can only target valid—as intended by the programmer—code paths. Unfortunately, practical CFI implementations suffer from precision loss when determining the set of valid targets for each branch [11], [19], [23]. Naturally, this also goes for CFI implementations that only protect virtual callsites [15]. Since it is even harder to recover class hierarchies of an application without access to its source code, current state-of-the-art binary-level defenses rely on weak characteristics to narrow down the set of call targets [23]. Examples include looking only at argument count information [28], enforcing that the *vtblptr* has to point to read-only memory [13], or allowing all existing vtables at a *vcall* [21]. Despite drastically reducing the set of valid targets, these approaches may still leave enough wiggle room for attackers to launch devastating attacks [11], [19], [23].

With the reconstructed class hierarchies, we can extend existing binary-level CFI solutions with a vtable protection. Our goal is to increase CFI guarantees for C++ applications by expanding state-of-the-art defenses with a mechanism to enforce correct class hierarchies for indirect branches. On that account, we extract the index into the vtable for each identified *vcall* that is used to determine the function entry (as explained in Section II-C). With the help of the class hierarchy, we are then able to generate a *function type* for all virtual functions at this position in the class hierarchy. In our example shown in Figure 1, functions `A::funcA1`, `B::funcB1`, and `thunk to C::funcB1` would get the same *function type*. Obviously, targeting `A::funcA1` is not allowed at *vcalls* that are used to branch to `B::funcB1` and `thunk to C::funcB1`, indicating an overestimation of our approach. However, the achieved precision of the call target set is a vast improvement in comparison to existing binary-only vtable protection approaches and it remains to be shown that this small overestimation can be exploited by an adversary [29].

Ideally, the vtable protection would merely insert a label check before each *vcall* that verifies whether the target is of the same *function type* as the virtual callsite. Since our static analysis is in certain cases not able to precisely assign a hierarchy to each *vcall*, however, we apply two additional techniques:

- *Dynamic Analysis*. To increase coverage, we run the binary in a controlled environment with trusted input (e. g., by running unit tests). During dynamic analysis, we inspect whether (i) executed indirect calls exhibit characteristics of a *vcall*, (ii) hierarchies used at the same *vcall* are merged together, and (iii) detected *vcalls* are in fact *vcalls*.
- *Slow Path*. Since our analysis may still miss key information about callsites (e.g., class hierarchy relations, leading to false positives), our extension can enter a slow path when a *function type* check failed (treating the failure as an anomaly rather than breaking the program). This slow path can be used to further investigate the branch and to decide if it is allowed or not.

Moreover, as we are only interested in protecting C++ semantics, our static analysis filters callsites that are definitely not *vcalls*.

We implemented a binary-only vtable protection on Linux for x86-64, using a similar binary run-time instrumentation model as proposed by Van der Veen et al. [28]: we use Dyninst [10] to move all functions to a protected shadow code region and prepend them with a two-byte *function type* value, as obtained from our static analysis. Next, we instrument each *vcall* with a short sequence of instructions. These instructions check whether the target's *function type* matches that of the *vcall*. If not, it enters our slow path, which we implemented by using the *PathArmor* open source CFI framework [27]. Note that we did not implement our own user-space JIT verifier, but rather let the kernel module sleep for 10 ms

whenever a new path is found. This is to carefully mimic the behavior of the original *PathArmor*'s implementation with an over-approximation of the average values (Table 3 from the *PathArmor* paper [27]). We also remark that we adopted *PathArmor*'s context-sensitive CFI approach to demonstrate the feasibility of our protection strategy similar to other efforts [20], but our system can incorporate any other solution to operate heavyweight security checks on the slow path.

B. Type-safe Object Reuse

Typically a process reuses freed memory for new allocations blindly. Attackers can abuse this mechanism by exploiting use-after-free vulnerabilities, where a malicious object is carefully placed in memory that was previously occupied and (dangling) pointers still point to it. Type-safe memory allocators such as Cling [8] aim at reducing this risk by preventing memory chunks of different types from being allocated in the same location. Essentially, type-safe memory (and in our case object) reuse maintains pools (i. e., memory regions) that are used for allocating only a particular type of object. Newly allocated objects are placed only in their own typed pool, and objects with different types cannot share a common memory location in the lifetime of the process. Assuming the class hierarchy of a C++ program is known, types can be defined based on class relations. As a result, we can reduce the attack surface by forcing pointers of similar typed objects to overlap. Unlike Cling [8], we focus only on type-safe C++ object reuse, with object types derived from the recovered class hierarchy. On the other hand, Cling is C++ agnostic in principle and the class hierarchy as reconstructed by *Marx* can significantly improve it in handling C++ allocations. The benefit is to reduce the number of typed pools (and memory usage) and also avoid expensive instrumentation to derive the run-time type (Cling relies on callstack hashes rather than offline type information).

To demonstrate this concept, we built a type-safe object reuse system based on the class hierarchy exported by *Marx*. Our system consists of two parts: an allocator with type-safe object reuse support and a library to instrument object allocations. The allocator enhances `tcmalloc` with functions that leverage type information to place new objects in type-based pools.

Specifically, in `tcmalloc`, pools are subdivided in alignment pools. For performance reasons, `tcmalloc` keeps track of thread-local pools and one central pool. When a thread-local pool reaches a predefined limit of free pages, it transfers some of the free pages to the central pool. Merging pools can be an issue for typed allocations since different types can end up in the same pool. Notice that new typed allocations must be aligned with past ones. When the allocation size does not divide the page size, it is possible that an object overlaps two pages. If this is the case and at least one of the two pages with the overlapping object are given back to the central pool, we can not guarantee that following allocations are correctly aligned. Therefore, we do not give any memory back from typed pools if the alignment size class does not divide the size of a page.

Finally, our type-safe object reuse application contains a shared library that instruments all allocations at runtime. The shared library is preloaded with the protected binary to trigger type-based allocations when virtual objects are instantiated and to resolve the actual allocation type based on the available class hierarchy. The type resolution works as follows. We start by preprocessing the analysis of *Marx* to construct triples of the form `(location, size, type)`. Here, `location` is the address of the call-site of `new`, `size` the size given to `new`, and `type` is a unique identifier. Moreover, the type identifier (or *type tag*) is generated by assigning a distinct value to each unique class hierarchy found by *Marx*. At runtime, we load the file containing these triples and store them in a hashtable. This hashtable uses the tuple `(location, size)` as key and the `type` as value. The shared library overrides the `new` and `new[]` operators, so we can infer the type information before dispatching to our typed allocation function in `tcmalloc`. Note that some allocations may be missed. We stress here that our intention is to showcase a prototype based on the exported class hierarchy and not a mature defense. In a real setting, the binary should be rewritten [10] by adding the resolution code to all callsites that construct virtual objects—eliminating the need for any run-time type inference instrumentation.

For each occurring allocation, the size of the allocation is used as the key and the location is computed using the return address. With this information, the type tag is retrieved from the hashtable and passed to the modified allocator function, which maintains a pool per allocated type. When no type exists for the particular location and size combination, a value of zero is returned. The allocator uses this value to choose a fast path, where no typed memory pools are used.

VI. Evaluation

In this section, we evaluate *Marx* and its applications in terms of performance and accuracy. Unless stated otherwise, all test cases are compiled using GCC 4.8.5. Our test cases include a variety of real-world applications and shared libraries. Consequently, no alterations to the compiler options specified by a test case have been made, i. e., each program is compiled with the compiler flags intended by the authors. We evaluated our class hierarchy reconstruction and virtual callsite target resolution on Ubuntu 14.04 LTS running on an Intel Core i7-2600 CPU with 16 GB of RAM.

The evaluation testbed for our binary vtable protection and type-safe object reuse implementations is a system with an Intel Core i7-6700K CPU @ 4.00GHz and 16 GB of RAM, running Ubuntu 14.04 LTS with Linux kernel 4.2.0 and transparent huge paging disabled.

A. Class Hierarchy Reconstruction

The main goal of our framework is to provide an analyst with accurate information about the class hierarchies. Hence, we evaluated the precision of *Marx* by comparing the analysis results with the class hierarchies of the application as reported by the compiler. More specifically, the ground truth is obtained by parsing the RTTI of the target application. Remember that our analysis reconstructs an individual class hierarchy as a set and does not contain information about the direction of inheritance. Hence, the ground truth is also extracted as a set. Table I shows the accuracy of our class hierarchy reconstruction for various real-world applications and shared libraries. Sizes in the table are given in MiB and are taken from the stripped binaries without debug information.

TABLE I. RESULTS OF THE CLASS HIERARCHY RECONSTRUCTION ANALYSIS. *size* GIVES THE SIZE OF THE STRIPPED BINARY IN MIB. #*GT* AND #*analysis* GIVE THE NUMBER OF HIERARCHIES IN THE GROUND TRUTH AND FOUND DURING THE ANALYSIS, RESPECTIVELY. #*matching* GIVES THE NUMBER OF HIERARCHIES THAT ARE CORRECTLY RECONSTRUCTED. #*overestimated* AND #*underestimated* GIVE THE NUMBER OF RECONSTRUCTED HIERARCHIES THAT ARE OVERESTIMATED AND UNDERESTIMATED, RESPECTIVELY. #*not found* GIVES THE NUMBER OF HIERARCHIES THAT WERE NOT FOUND DURING THE ANALYSIS. #*not existing* GIVES THE NUMBER OF HIERARCHIES THAT WERE FOUND DURING THE ANALYSIS BUT DO NOT EXIST IN THE GROUND TRUTH. *time needed* GIVES THE TIME THAT THE STATIC ANALYSIS NEEDS TO COMPLETE.

Program	*size (MiB)*	#*GT*	#*analysis*	#*matching*	#*overestimated*	#*underestimated*	#*not found*	#*not existing*	*time needed (hh:mm:ss)*
VboxManage 5.0.24	0.97	33	45	32	–	1	–	9	0:06:12
MySQL Server 5.7.11	23.91	78	117	69	1	7	1	–	11:36:17
MongoDB 3.2.4	27.72	158	253	137	–	8	13	63	1:08:41
Node.js 5.10.1	15.18	59	84	55	2	2	–	14	0:33:16
FileZilla 3.13.1 (GCC 4.9)	4.42	21	9	3	6	4	8	1	1:19:59
VboxRT.so 5.0.24	2.27	3	3	2	–	–	1	1	0:00:02
VboxXPCOM.so 5.0.24	1.06	8	14	3	–	2	3	1	0:00:05
libFLAC++.so 6.3.0	0.10	3	3	3	–	–	–	–	0:00:01
libebml.so 1.3.3	0.14	2	2	2	–	–	–	–	0:00:01
libmatroska.so 1.4.4	0.65	2	2	2	–	–	–	–	0:00:17
libmusicbrainz5cc.so 5.1.0	0.56	3	2	1	–	1	1	–	0:00:01
libstdc++.so 6.0.18	0.93	5	24	2	–	2	1	–	0:00:01
libwx_baseu-3.1.so 3.1.0	2.55	33	26	26	–	–	7	–	0:00:47
libwx_baseu_net-3.1.so 3.1.0	0.29	5	7	4	–	1	–	–	0:00:01
libwx_gtk2u_adv-3.1.so 3.1.0	1.94	20	23	17	1	1	1	–	0:00:21
libwx_gtk2u_aui-3.1.so 3.1.0	0.59	7	7	5	1	1	–	–	0:00:01
libwx_gtk2u_core-3.1.so 3.1.0	5.92	41	46	31	6	2	2	1	0:01:17
libwx_gtk2u_html-3.1.so 3.1.0	0.79	5	9	2	2	1	–	–	0:00:06
libwx_gtk2u_xrc-3.1.so 3.1.0	1.06	4	4	2	1	1	–	–	0:00:03

Overall, we observe that *Marx* is capable of precisely recovering the information about the class hierarchies for many types of applications. We find that the results are better for applications than for shared libraries. For applications, the analysis process was able to correctly reconstruct 84.8% of the hierarchies on average. Only 6.3% are underestimated and also 6.3% of the hierarchies were not found. For shared libraries, on average, 72.3% of the hierarchies were correctly reconstructed, while 8.5% of the hierarchies were underestimated and 11.3% were not found. Consequently, we conclude that *Marx* is able to recover most of the class hierarchies of the target binaries completely and therefore provides helpful information for an analyst. The difference between applications and shared libraries results stem from the fact that the analysis of a shared object misses a lot of context (cf. Section IV-A). Shared objects are not written to be executed as a standalone application. Hence, most functions are not called from within the shared object, but only from an application, using the interface exposed by the library.

This is also evident when looking at the time needed to analyze an application in comparison to a shared library. Almost all of the tested shared libraries are analyzed in under a minute. The functions of applications are more connected with each other through calls. Since *Marx* follows these connections and analyzes the called functions within the current context, it needs more time to analyze the whole application. In contrast, shared libraries tend to provide a rather "flat" functionality and do not have so many connected functions. Hence, analyzing them is faster.

The application with the best results is *VboxManage*. *Marx* underestimated only one hierarchy and correctly reconstructed the remaining 32. However, *Marx* also found 9 hierarchies that do not exist in the application. Note that non-existing hierarchies are most likely not used in code constructs such as *vcalls* or object creation at a `new` operator. Hence, in applications such as vtable protection or type-safe object reuse such overestimations have no effect and do not influence the results.

For the largest application, *MongoDB*, *Marx* was able to reconstruct 137 out of 158 hierarchies correctly. Only 8 hierarchies were underestimated and 13 were not found during the analysis. Most of these missing hierarchies are connected via an abstract class which was not referenced in the binary code (most likely due to compiler optimizations) and hence not found during the analysis. For the largest shared library, *libwx_gtk2u_core-3.1.so*, 31 hierarchies were correctly reconstructed. 2 hierarchies were underestimated and only 2 were not found during the analysis.

The application *FileZilla* had to be compiled with GCC 4.9 since it requires support for C++14, which is not available for GCC 4.8. It has the worst results of all test cases, as only 3 out of 21 hierarchies were reconstructed correctly. 6 hierarchies were overestimated during reconstruction, 4 underestimated, and 8 not found at all. A manual evaluation of the underestimated and missing hierarchies yields two reasons for these results: First, most of these hierarchies are connected via classes for which no vtable has been emitted by the compiler, which is why *Marx* cannot leverage them. This is due to optimization passes that remove these vtables from the binary. A detailed discussion is given in Section VII. Second, *FileZilla* makes heavy use of the *wxWidgets* library (i.e., the shared objects with *libwx_* prefix in Table I). Some underestimated hierarchies are connected via vtables from these shared objects. Despite *Marx*'s inter-modular data flow ability, it was not able to find a connection between all classes of the underestimated hierarchy with the external ones. A manual investigation revealed that not all classes (despite their connection to an external class according to RTTI) execute a library function that overwrites the *vtblptr*—presumably due to compiler optimizations.

B. Virtual Callsite Targets

With static analysis, it is hard to determine the target function of an indirect call. As noted earlier, for binaries compiled from C++ code, virtual functions are mostly implemented using indirect call instructions. To assist a reverse engineer, our static analysis hence attempts to resolve the target set of a *vcall* as accurately as possible. To evaluate the correctness of

TABLE II. RESULTS OF THE VIRTUAL CALLSITE RESOLUTION. #*GT* AND #*analysis* GIVE THE NUMBER OF VIRTUAL CALLSITES IN THE GROUND TRUTH AND THE FRAMEWORK'S RESULTS, RESPECTIVELY. #*correct* GIVES THE NUMBER OF VIRTUAL CALLSITES IDENTIFIED CORRECTLY. *identified* GIVES THE VALUE IN PERCENT OF HOW MANY VIRTUAL CALLSITES OF THE GROUND TRUTH ARE IDENTIFIED. #*resolved* GIVES THE NUMBER OF RESOLVED VIRTUAL CALLSITE TARGETS FOR THE NON-CONSERVATIVE AND CONSERVATIVE MODE (THE LATTER IN PARENTHESES). #*matching* GIVES THE NUMBER OF RESOLVED TARGETS WHICH MATCH COMPLETELY WITH THE GROUND TRUTH. #*overestimated* AND #*underestimated* GIVE THE NUMBER OF TARGET SETS THAT ARE OVERESTIMATED AND UNDERESTIMATED, RESPECTIVELY. #*not existing* GIVES THE NUMBER OF VIRTUAL CALLLSITES RESOLVED THAT DO NOT EXIST IN THE GROUND TRUTH.

	Finding Virtual Callsites				**Resolving Virtual Callsites**				
Program	#*GT*	#*analysis*	#*correct*	*identified*	#*resolved*	#*matching*	#*overestimated*	#*underestimated*	#*not existing*
VboxManage	✗	✗	✗	✗	✗	✗	✗	✗	✗
MySQL Server	✗	✗	✗	✗	✗	✗	✗	✗	✗
MongoDB	14357	13369	12607	87.8%	736 (589)	159 (91)	550 (471)	27 (27)	0 (0)
Node.js	4925	5591	4879	99.0%	798 (754)	166 (142)	629 (611)	1 (0)	2 (1)
FileZilla	2779	2544	2495	89.7%	226 (210)	3 (3)	56 (48)	167 (159)	0 (0)

the analysis, we utilize the VTV (Virtual Table Verification) GCC pass [26] to generate the ground truth. VTV collects class information at compile time and emits code that verifies each virtual call before (potentially) executing it. Verification is performed by checking the object's vtable against a set of allowed vtables. In essence, this performs a check against a specific class hierarchy. For our ground truth, we extract said information and try to match it to the *vcall* it guards. As test cases, we evaluated the applications used in Section VI-A. Unfortunately, we were unable to compile the applications *MySQL Server* and *VBoxManage* with VTV. More specifically, the compiler crashed during the compilation of *MySQL Server* and for *VBoxManage* we were not able to pass the configure script.

Table II shows the results of the *vcall* target resolution. Remember that non-conservative mode did not require validity of the *thisptr*, but only a dependency on the *vtblptr* when calculating the target address. As evident from the table, non-conservative mode is able to resolve more *vcalls* during the analysis. Furthermore, the false positive rate did not increase significantly.

For the application *Node.js*, only 2 *vcalls* were wrongly detected in non-conservative mode, whereas only 1 was not found in conservative mode. In turn, the non-conservative mode finds 43 *vcalls* more compared to conservative mode. All in all, for *Node.js*, the analysis was able to identify 4,879 *vcalls* correctly, which are 99.0% of all virtual callsites.

The worst results were achieved for *FileZilla*. The analysis was only able to resolve 3 *vcalls* correctly and most of the remaining resolved *vcalls* were underestimated. This results from the relatively poor results during class hierarchy reconstruction in comparison to the other applications. Due to missing and underestimated hierarchies, *Marx* underestimates the targets of most of the resolved *vcalls*. However, 2,495 *vcalls* were identified correctly, which are 89.7% of all virtual callsites.

Overall, *Marx* is able to support an analyst by providing him with potential target addresses for *vcalls*. Depending on the precision of the class hierarchy reconstruction, the set of target addresses might be underestimated. However, most of the target sets are overestimated such that the analyst does not miss branches during the analysis. On average, 90.5% of all virtual callsites were identified by *Marx* during the analysis. While the results of the call target resolution are helpful for a reverse engineer, more comprehensive target sets can be obtained by combining our static approach with a dynamic profiling phase (such as in Section V-A).

C. VTable Protection

We focus the performance evaluation of our vtable protection implementation on two popular Linux C++ servers and the seven C++ applications found in SPEC. Specifically, we evaluated our binary vtable protection with a cross-platform runtime environment for server-side web applications (Node.js 5.10.1, statically compiled with Google's v8 JavaScript engine) and a database server (MySQL 5.7.11). To benchmark Node.js, we configured the Apache benchmark [1] to issue 250,000 requests with 10 concurrent connections and 10 requests per connection for the default page. To benchmark MySQL, we configured the Sysbench OLTP benchmark [4] to issue 10,000 transactions using a read-write workload.

We evaluated our vtable protection instrumentation using the analysis results from *Marx*. To determine the impact on runtime performance, we measured the time to complete the execution of the benchmarks and compared against the baseline—i.e., the original version of the benchmark with no binary instrumentation applied. Table III details our results.

As shown in the table, it is evident, considering the massive number of executed virtual calls, that our vtable protection performs surprisingly well—10.8% runtime overhead across all the tested applications (geometric mean). Interestingly, there seems to be no direct correlation between the number of executed virtual calls and the resulting overhead. SPEC binaries *astar* and *povray*, for example, both execute over 4.5 billion virtual calls—all resolved using *Marx*'s analysis results—but yield fairly different runtime overheads: 3% for *astar*, vs 10% for *povray*, a delta that might be caused by CPU caching behavior. We believe that these results are encouraging: they demonstrate that enforcing vtable protection (or CFI) over *likely* (rather than precise) invariants by using a slow path for second-stage verification is feasible in practice.

D. Type-safe Object Reuse

To evaluate our type-safe object reuse application, implemented on top of *Marx*, we ran experiments on the same set of applications described in Section VI-C. Table IV presents our results. The first two columns contain the number of unique `new` (including `new[]`) callsites and types found by *Marx*'s analysis. Next, we present the number of unique types caught by our library, followed by the number of times `malloc`, `new` and `new[]` were called (either typed or untyped) during the benchmark. Finally, we show the overhead from our library.

We observe a slight speedup for *astar*, *povray* and *soplex*. As can be observed from Table IV, the latter two are not heavy

TABLE III. EVALUATION RESULTS FOR OUR BINARY VTABLE PROTECTION IMPLEMENTATION. FOR EACH BINARY, THE TABLE SHOWS (I) *Binary Instrumentation* DETAILS, DEPICTING THE NUMBER OF INSTRUMENTED *vcalls*, WRITTEN *labels* AND *moved* FUNCTIONS; (II) *Runtime Statistics*, LISTING THE NUMBER OF *vcalls executed* AT RUNTIME, THE NUMBER OF VCALLS FOR WHICH A MATCHING TYPE WAS FOUND AT THE TARGET FUNCTION (*fastpath*), THE NUMBER OF TIMES THE *slowpath* WAS ENTERED, AND THE NUMBER OF *unique* PATHS THAT REQUIRE JIT VERIFICATION; AND (III) *Normalized Runtime*, LISTING OUR VTABLE PROTECTION RUNTIME OVERHEAD WITHOUT VERIFICATION (*hashing only*) AND WITH A SYNTHETIC VERIFICATION TIMEOUT OF 10MS PER UNIQUE PATH (*+verification*).

	Binary Instrumentation			**Runtime Statistics**				**Normalized Runtime**	
Program	*#vcalls*	*#labels*	*#moved*	*#vcalls executed*	*#fastpath*	*#slowpath*	*#unique*	*hashing only*	*+verification*
MySQL	10,864	8,421	28,971	106,330,186	105,035,488	1,294,698	9	1.145	1.155
Node.js	5,905	5,917	26,751	31,491,929	31,491,918	11	6	1.263	1.265
astar	1	1	96	4,595,981,552	4,595,981,552	0	–	1.031	1.031
dealII	1,434	1,428	7,217	96,751,718	96,751,718	0	–	1.012	1.012
namd	2	3	102	2,016	2,016	0	–	0.999	0.999
omnetpp	706	725	1,949	2,061,547,468	2,061,206,142	341,326	361	1.067	1.083
povray	109	111	1,622	4,704,273,295	4,704,273,295	0	–	1.103	1.103
soplex	497	498	873	1,772,890	1,155,673	617,217	661	1.016	1.086
xalancbmk	9,303	9,340	12,808	8,306,798,756	8,306,260,183	538,573	111	1.264	1.272
geomean	342	350	2,318	91,018,910	86,672,172	0	67	1.096	1.108

TABLE IV. EVALUATION RESULTS FOR OUR TYPE-SAFE OBJECT REUSE IMPLEMENTATION. FOR EACH BINARY, THE TABLE SHOWS (I) Marx *Statistics*, DEPICTING THE NUMBER OF EXTRACTED `NEW` CALLS AND THE NUMBER OF DIFFERENT `TYPES`; (II) *Runtime Statistics*, LISTING THE NUMBER OF USED `TYPES`, CALLS TO `MALLOC`, `NEW`, AND THE `NEW`-ARRAY OPERATOR DURING EXECUTION, AND (III) *Normalized Runtime*.

	Marx Statistics		**Runtime Statistics**				**Normalized Runtime**
Program	*#`new`*	*#types*	*#types*	*#`malloc`*	*#`new`*	*#`new[]`*	*overhead*
MySQL	1,017	47	16	2,705,675	82,225	13	1.009
Node.js	4,675	38	14	7,685,562	12,228,927	9,093,605	1.022
astar	11	0	0	1,008,577	108,037	8	0.999
dealII	1,632	11	5	48	144,642,689	6,616,448	1.016
namd	584	2	1	2	2	1,320	0.999
omnetpp	717	9	1	45,950,697	0	221,218,929	1.028
povray	54	7	6	2,414,075	83	176	0.995
soplex	20	6	2	3,718	3	4	0.997
xalancbmk	2,051	167	46	6,854	135,148,541	158	1.046
geomean	350	6	2	56,309	6,032	3,888	1.012

users of the `new` and `new[]` operators, and although *astar* performs many calls to `new`, we detected no types during its execution. This is caused by the fact that *astar* does not rely on many C++ features [16]: *Marx* recovered *one* vtable which is never written into a heap object in the program. Thus, the `new` operator is never used with a type.

Our results for the real-world applications Node.js and MySQL are much more realistic compared to the SPEC benchmarks: our type-safe object reuse implementation captures a significant fraction of the C++ types as reconstructed by *Marx*. Although both applications heavily depend on C++ objects, the overhead imposed by the type-safe object reuse application is low. For example, in Node.js, we recorded more than 21 million new objects, while its normalized runtime is 2.2%. We think that these results are encouraging: type-safe object reuse provides significant security invariants, while our experiments report a performance overhead of less than 5% (geometric mean).

VII. DISCUSSION

In the following, we discuss the effects of compiler optimizations on our analysis and review several ways to optimize our prototype implementation.

A. Compiler Optimizations and Lost Information

Even though all of our evaluation results are encouraging, we note that the biggest limitations of our approach are due to compiler optimizations and a loss of information on the binary level. Inherently, *Marx* is dependent on vtables (and references to them) emitted by the compiler. Especially for *abstract* base classes, however, such relations may not be revealed by certain vtable usage patterns; the information is simply missing from the binary and we cannot recover this information. This increases the observable *gap* between the formal class hierarchy as set up by the programmer and the results obtained by *Marx*, based on artifacts found in the (optimized) binary itself.

Such a case was encountered during the evaluation of *FileZilla*. A compiler optimization removed vtables of abstract classes from the binary which were the base classes of complete hierarchies. As a result, the overwrite analysis failed to join the smaller hierarchies. Since the vtables did not have other characteristics that allows our approach to find a connection (e. g., via heuristics discussed previously), the reconstructed hierarchies were either not complete or not found at all. Hence, the quality of our results depends on the size of the gap between the formal and the actual class hierarchy as encoded in the binary.

Other than this, we did not encounter any application-specific idiom that affect the accuracy of our results.

B. Improving Analysis Contexts

Since our static analysis approach focuses on real-world applications, we had to weigh up precision against performance to be able to scale to complex binaries. Hence, we introduced

limiting factors such as the call depth restriction and characteristics that we deem as interesting in a basic block. One problem that may arise with these restrictions is that we may miss important information during our analysis. Consider, for example, a function f_{imp} which yields valuable information for our analysis, but is called from a fixed callsite. In the following, we call such a function an *important* function. When our static analysis processes the function's caller and the basic block that calls f_{imp} is not considered interesting, it is highly likely that no path is generated which ends up in f_{imp}. Hence, our analysis misses context that would be provided by the function and its results loose precision.

A naïve approach to tackle this problem is to consider all call instructions as interesting during the path generation (i. e., follow every call). This, however, does not scale to real-world applications due to the path explosion problem. A better solution is to mark those basic blocks with call instructions as interesting that *eventually* reach important functions. In other words, we recognize *importance* of functions as a transitive function which, in turn, impacts the importance of its callers.

However, *Marx* analyzes the functions on a on-demand basis and does not know if the target of a call instruction is important for the analysis process (i. e., it only takes information *local* to the current function into account). In order to add *global* information about the importance of a function into *Marx*' decision process, we propose to add a preliminary preprocessing step. In essence, we build a static call graph which allows to propagate information about important basic blocks up to its callers. During path generation, this can affect the decision whether or not to follow an (otherwise uninteresting) call. Additionally, this call graph can be enriched at analysis time to include target sets resolved at a *vcall*. Further, it allows to dynamically adjust the call depth.

C. Improving Shared Library Results

As shown in Section VI-A, the class hierarchy reconstruction of shared libraries is not as precise as for applications. This is due to the fact that shared libraries are written to be used from other applications or shared libraries. Hence, most functions in a shared library are not called from within the very same module. As a result, *Marx* has to analyze these functions without any context given by the caller (e. g., a *vcall* is using an object that is provided by the caller). This missing information leads to a lower precision in reconstructing the class hierarchy and fewer *vcalls* are found. One way to tackle this problem is to analyze the shared library in combination with an application that is using it. Once a function inside a shared library is called from the application, the analysis framework has a context that might help improving the results. However, this does not necessarily cover all exported functions of the shared library. Also, an analyst might not always have an application at hand that is using the shared library that he has to analyze.

D. Reconstructing RTTI

An interesting application of the class hierarchy reconstruction results is the subsequent reconstruction of RTTI associated with vtables. This information can, in turn, be leveraged by other applications, such as analysis programs or protection mechanisms which are able to perform better when provided with RTTI. Notably, this would be an easy way to incorporate our results in potentially closed-source applications which would not require modifications to the programs themselves. However, since *Marx* is not able to recover the class hierarchies with full precision in the general case, the applications have to be able to cope with a certain amount of imprecision.

Furthermore, RTTI holds information about the inheritance direction. More specifically, it only contains a pointer to the RTTI of parent classes. Currently, our analysis approach is not able to extract the direction of the inheritance. Therefore, the recovered RTTI would contain all classes that are in the same hierarchy and therefore overestimate it. Still, we note that extraction of the inheritance direction can be added in the future.

E. Improving VTable Protection

As shown in Section VI-C, the results of *Marx* can be used for a binary-only CFI implementation focusing on *vcalls*. However, even with a dynamic profiling phase to improve the results of our static analysis, the slow path of our implementation is still required by some applications, which leads to a relatively high performance overhead. In order to tackle this problem, the implementation can be extended to use the technique proposed by Prakash et al. [21]. If our analysis cannot assign a reconstructed class hierarchy to a given *vcall*, the CFI implementation can allow all functions at the same offset in any known vtable. This way, the implementation would have two different protection granularities: For *vcalls* with an assigned class hierarchy, the set of allowed functions lies within the class hierarchy. For *vcalls* without an assigned class hierarchy, the set of allowed functions lies within the known vtables. Hence, the verification at a *vcall* without an assigned class hierarchy can also be implemented using a simple label check.

VIII. Related Work

We now review related work on the reconstruction of C++ class hierarchies and discuss how *Marx* advances the field. Most similar to our static analysis approach is the work conducted by Jin et al. [17]. Their approach, called *objdigger*, uses symbolic execution and inter-procedural data flow analysis to discover objects of classes, their attributes, and methods. However, their approach does not reconstruct the class hierarchy and only the ideas to recover it are described in the paper (which are similar to our *vtblptr* overwrite analysis). Furthermore, the evaluation is only done on small test cases with up to 10 classes instead of complex binaries.

The approach presented by Fokin et al. [12] focuses on reconstructing the class hierarchies of C++ programs. Their approach recovers the vtables in memory and analyzes them and their corresponding constructors. However, they focus on analyzing the structure of the vtable size and the usage of pure virtual functions to recover the direction of the inheritance. The data-flow through the program is not considered in their work, leading to a certain imprecision.

Katz et al. [18] proposed an approach to support an analyst that reverse engineers C++ binaries based on machine learning. Their approach outputs a probability that indicates what class

is used at a given *vcall*. This is done by using sequences of instructions that can be assigned to a specific class as a training set. The trained model is then used to estimate other *vcalls* and the used class, with the goal of giving the analyst a hint where the control-flow might go next. Unfortunately, their approach ignores polymorphism and is only able to provide one possible branch target. Additionally, their evaluation was done on small applications (largest one has a size of around 1MB) on a machine with 64 CPUs that took several hours. Therefore, their approach is not able to support an analyst on reverse engineering real-world C++ applications.

The binary analysis framework *angr* is presented by Shoshitaishvili et al. [24]. Their work focuses on re-implementing existing techniques for vulnerability identification in order to compare them with each other. The introduced framework has a modular design and provides the possibility to be extended with new analysis techniques. The presented algorithms of our approach could also be implemented with *angr* instead of writing an own framework. However, *angr* is written in Python and due to its performance, it is likely not efficient for large real-world binaries such as *Node.js* or *MySQL Server*.

Prakash et al. [21] presented *vfGuard*, a binary-only indirect call protection mechanism for C++ binaries. Their approach tries to protect *vcalls* by creating a whitelist with valid calll targets. If the target address is not within the whitelist, an attack is assumed and the execution is terminated. The whitelist is determined by the offset into the vtable that is used by the *vcall*. However, they do not try to recover the class hierarchies because of its difficulty and just allow any vtable at a *vcall* (with some additional filtering). *T-VIP*, proposed by Gawlik et al. [13], is also a binary-only approach to protect virtual callsites from vtable hijacking attacks. However, they do not recover C++ specific structures such as vtables, but reduce the virtual callsite characteristics to two heuristic policies. The first policy restricts the *vtblptr* to point to read-only memory at a *vcall*. The second policy checks if a random function pointer in the vtable points to memory that is not writable. Both policies narrow down the ability of an attacker to inject a crafted vtable. However, more advanced code-reuse attacks such as proposed by Schuster et al. [23] are not affected by these policies. Gawlik et al. also proposed a third policy to check if the used vtable resides in an allowed set built with the help of the class hierarchy. However, they did not implement this idea because previous existing work did not show a practicable recovery of class hierarchies for real-world programs.

Most similar to our presented application of a type-safe object reuse is Cling, a work presented by Akritidis [8]. Cling is a type-safe memory allocator used to mitigate use-after-free attacks. It modifies the heap allocation process to provide types for each memory allocation that is made in the application. Cling uses the address of the allocation site and size as a type for its pools. Hence, a use-after-free bug only grants access to the remaining data of the same object type. In contrast, our presented application builds types on the base of the reconstructed class hierarchies. Since Cling is C++ agnostic in principle, the class hierarchy as reconstructed by *Marx* can significantly improve it in handling C++ allocations. The benefit is to reduce the number of typed pools (and memory usage) and also avoid expensive instrumentation for deriving the run-time type. In a similar fashion, VTPin [22], a vtable hijacking protection for binaries, which is currenlty class-agnostic, could potentially leverage the extracted hierarchies for increasing the accuracy in collecting pinned vtable pointers.

IX. Conclusion

In this paper, we presented a practical and efficient approach to reconstruct C++ class hierarchies from a given binary application. Our static analysis follows data flow and tracks objects through multiple paths through the target binary whilst taking C++ characteristics into account. Hence, we recognize artifacts resulting from the way compilers implement high-level features such as polymorphism and use them to recover information about the relation of classes in the binary.

We presented the design and implementation of a tool called *Marx* capable of performing the outlined approach and evaluated it on several large, real-world applications. The results are promising: On average, 84.6% of the class hierarchies of applications and 73.3% of the class hierarchies of shared libraries were precisely reconstructed. The information provided by our analysis can then be used to resolve the sets of potential target functions of virtual callsites and helps an analyst following control flow even across previously unresolvable indirect calls.

Furthermore, we present two applications built atop of the analysis results: an improved *vtable protection* mechanism for binary executables, verifying the integrity of control flow, and *type-safe object reuse*, which enhances type-safe memory allocators. We demonstrate that, even in cases where the extracted class hierarchy is reconstructed imperfectly, practical defenses that improve security while maintaining a reasonable performance level can be developed based on our results. To compensate for the imprecision of the analysis, our vtable protection treats violations as anomalies and triggers more heavyweight checks on a slow path (trading off on performance). On the other hand, our type-safe object reuse solution can gracefully tolerate type-to-pool mapping mismatches (trading off on security). In short, we show that it is possible to build *fully conservative* binary-level defense solutions on top of imprecise information, exposing new interesting and previously unexplored tradeoffs.

Since we believe that our analysis framework *Marx* provides promising results in the analysis of large, real-world applications and hence represents a building block for future research, we make it available for the research community.

Acknowledgements

We would like to thank the anonymous reviewers for their comments. This work was supported by the European Commission through project H2020 ICT-32-2014 (SHARCS) under Grant Agreement No. 644571, ERC Starting Grant No. 640110 (BASTION), and by the Netherlands Organization for Scientific Research through grants NWO 639.023.309 VICI (Dowsing) and NWO CSI-DHS 628.001.021.

References

[1] Apache benchmark. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] IDAPython. https://github.com/idapython.

[3] Itanium C++ ABI. https://mentorembedded.github.io/cxx-abi/abi.html.

[4] SysBench. http://sysbench.sourceforge.net.

[5] Valgrind. http://www.valgrind.org/.

[6] C++ ABI for the ARM Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041e/IHI0041E_cppabi.pdf, 2015.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[8] P. Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security Symposium*, 2010.

[9] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*, 2016.

[10] A. R. Bernat and B. P. Miller. Anywhere, Any-Time Binary Instrumentation. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2011.

[11] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[12] A. Fokin, K. Troshina, and A. Chernov. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[13] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)*.

[14] J. Gray. C++: Under the Hood. http://www.openrce.org/articles/files/jangrayhood.pdf, 1994.

[15] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: VTable protection without loose ends. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[16] C. Isen and L. John. On the Object Orientedness of C++ programs in SPEC CPU 2006. In *SPEC Benchmark Workshop*. Citeseer, 2008.

[17] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering C++ Objects From Binaries Using Inter-Procedural Data-Flow Analysis. In *ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[18] O. Katz, R. El-Yaniv, and E. Yahav. Estimating Types in Binaries using Predictive Modeling. *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[19] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In *USENIX Annual Technical Conference*, 2016.

[20] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[21] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[22] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2016.

[23] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[24] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[25] B. Stroustrup. C++ Applications. http://www.stroustrup.com/applications.html.

[26] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[27] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[28] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[29] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining Trust on Virtual Calls. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.