MARX

Uncovering Class Hierarchies in C++ Programs

Raghuveer Shivakumar (Raghu)

University of Colorado, Boulder

2022-09-28

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

Authors: Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Atahansopoulos, Cristiano Giuffrida

Description: Using language semantics to generate CFG (particularly Class Hierarchy) for more robust defense against attacks like COOP – Advanced CFI

Background

In the itanium C++ ABI, a vcall always has the following form:

```
mov rdi, thisptr
call [vtblptr + offset]
```

RTTI: Holds metadata such as typeinfo, base class name, etc

Offset-to-Top: Offset from the top of the subvtable to the object's start addr

*Think of this as a continuation of the previous presentation

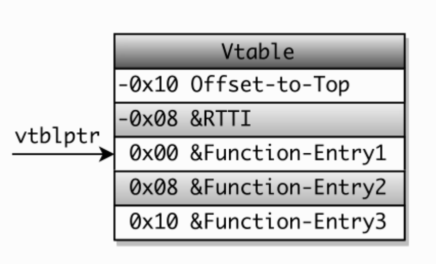

Figure 1: Inheritance in Itanium

Background - Contd..

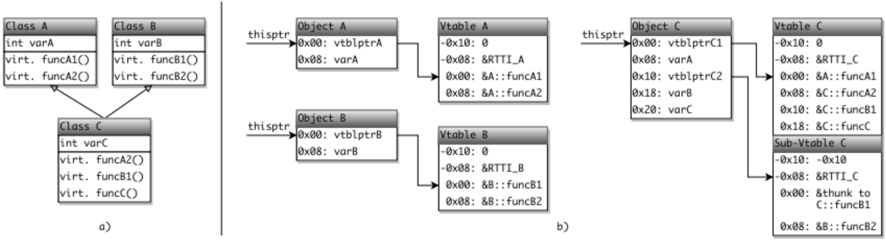

Figure 2: Inheritance in Itanium

Things to remember:

Order of Construction: `Base` $\rightarrow$ `Derived` | `Top-Down`

Order of Destruction: `Derived` $\rightarrow$ `Base` | `Bottom-Up`

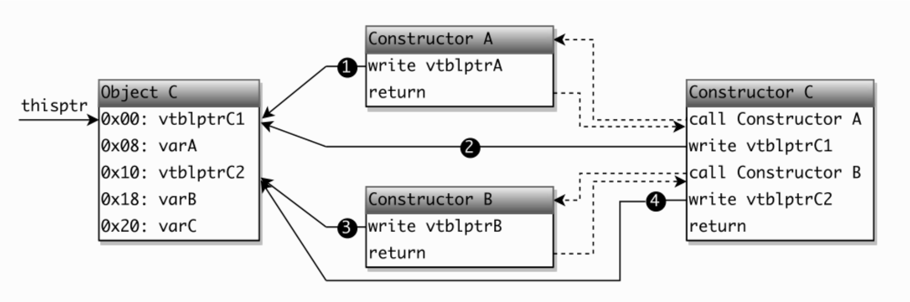

Figure 3: Vtable Construction

Things to remember:

```
mov rdi, thisptr
call [vtblptr + offset]
```

Order of Construction: `Base` $\rightarrow$ `Derived` | `Top-Down`
Order of Destruction: `Derived` $\rightarrow$ `Base` | `Bottom-Up`

Assumptions

- Compiler optimizations that change the order of construction and destruction are not used
- Compiler optimizations that elide vtables and vtblptrs are not used

Why?

- Hierarchical CFI is based on the assumption that the order of construction and destruction is fixed
- MARX depends on vtables and vtblptrs to identify the class hierarchy

Analysis and Implementation

Analysis is done in two phases:

- **Phase 1:** `vtable extraction`
  - Extract valid vtables from the binary
- **Phase 2:** `static analysis`
  - Analyze the extracted vtables to compensate for overestimated vtables

Vtable Extraction

- **Goal:** `Extract valid vtables from the binary`

Heuristic

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

*read-only sections: .ro-data, .data.rel.ro, etc
*relocation entries: .text, .plt, .extern, etc (in binary)
*valid offsets: -0xFFFFFF - 0xFFFFFF | 0x0 (No multiple inheritance allowed)

Static Analysis

- **Goal:** `Analyze the extracted vtables to compensate for overestimated vtables`
- **Overwrite Analysis**
  - Determine class hierarchy from vtables overwrites
- **Vtable Function Entries**
  - Multiple vtables may point to the same function entry at the same offset
- **Interprocedural Data Flow**
  - Retracing the return address of a vcall using the backward edges of the CFG
- **Intermodular Data Flow**
  - Run the analysis on the shared libraries first

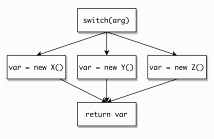

Figure 4: Interprocedural Data Flow

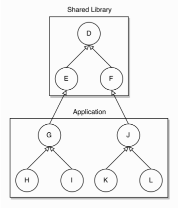

Figure 5: Intermodular Data Flow

Applications

VTable Protection for Binaries

- Traditionally techniques like ro_vtable_loc, argument_count, etc are used to protect vtables
- MARX uses triangulated types (*custom 2 byte function type) to verify vcalls.
- To increases coverage, MARX also uses Dynamic Analysis to verify the correctness of the identified types.

*Path-armor: CFI framework
*Dynlist: Dynamic analysis framework to inject 2 byte function types and move to a shadow region

Type-safe Object Reuse

- **Hash Table:** (location, size) -> (type, offset-to-top)
- Use the type to pool new object construction
- A modified version of `tcmalloc` was used to implement this

Discussion

- **Compiler Optimization**:
  - Removal of vtables of abstract base classes in the optimized version of Filezilla lead of incorrect class hierarchy
- **Improving Analysis Contexts:**
  - Artificial restrictions imposed on the call depth and other characteristics lead to missed info which could be used to improve the analysis
- **Reconstruction using RTTI:**
  - Current implementation of MARX relies on vtable updates to reconstruct the direction of hierarchy, but using RTTI can be used to improve the analysis.

Evaluation

Hierarchy Evaluation

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

Figure 6: Hierarchy Evaluation

Vcall sites

Program	Finding Virtual Callsites #GT	#analysis	#correct	identified	Resolving Virtual Callsites #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)

Figure 7: Vcall sites

*For vcall sites GT, gcc VTV pass was used.

Evaluation - Contd..

Vtable Protection:

- They were able to correctly identify the vtable and initiate a vcall to the correct function with 10% runtime overhead on an instrumented version of Nodejs

Type-safe Object Reuse:

- They found a performance overhead of 5% on an instrumented version of Nodejs when using the custom version of `tcmalloc` with triangulated types and typed pooling.

Questions?