# ShrinkWrap: VTable protection without loose ends

Istvan Haller
Vrije Universiteit Amsterdam
i.haller@student.vu.nl

Enes Göktaş
Vrije Universiteit Amsterdam
enes.goktas@vu.nl

Elias Athanasopoulos
FORTH-ICS
elathan@ics.forth.gr

Georgios Portokalidis
Stevens Institute of Technology
gportoka@stevens.edu

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@few.vu.nl

## ABSTRACT

As VTable hijacking becomes the primary mode of exploitation against modern browsers, protecting said VTables has recently become a prime research interest. While multiple source- and binary-based solutions for protecting VTables have been proposed already, we found that in practice they are too conservative, which allows determined attackers to circumvent them. In this paper we delve into the design of C++ VTables and match that knowledge against the now industry standard protection scheme of VTV. We propose an end-to-end design that significantly refines VTV, to offer a provably optimal protection scheme. As we build on top of VTV, we preserve all of its advantages in terms of software compatibility and overhead. Thus, our proposed design comes "for free" for any user today. Besides the design we propose a testing methodology, which can be used by future developers to validate their implementations. We evaluated our protection scheme on Google Chrome and show that no compatibility issues were introduced, while overhead is also unchanged compared to the baseline of VTV.

## 1. INTRODUCTION

C++ is a popular, fast, object-oriented (OO) language used to develop some of the most popular Web browsers, including Chrome and Mozilla. Due to their popularity, size and complexity, applications developed in C++ are frequently targeted by attackers. Despite advances in software security, like the introduction of data-execution prevention [4], stack-smashing protection [9], and address-space layout randomization [16], their exploitation is still possible. New techniques involving information leaks [19] and return-oriented programming [18] are employed to bypass protection mechanisms and perform arbitrary code execution attacks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA*

© 2015 ACM. ISBN 978-1-4503-3682-6/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818025

One of the features of C++ applications targeted by attackers are *virtual function tables*, or *VTables*. OO languages support run-time method binding, i.e., determining the method to be called based on the run-time type of an object, instead of the static type of the pointer pointing to that object. Modern compilers typically provide this functionality through *VTables*, which provide an efficient way to call the correct method at run time. Unfortunately, VTables are based on indirect calls, which is what makes them a prominent targets for hijacking the control flow of a program.

To prevent such control-hijacking attacks, the research community has turned to *control-flow integrity* (CFI). First conceived in 2005 [2], CFI has seen a long line of followers and variants since [14, 22, 23]. CFI strives to constrain the control flow of a program to its statically-determined control-flow graph (CFG) as strictly as possible. In principle, CFI can be very effective in preventing a wide-range of attacks. In recent times however, we bear witness to a cat-and-mouse game, where each new CFI technique is immediately attacked and bypassed. Earlier works have shown that attackers can bypass loose CFI mechanisms [11], so follow-up works tried to exploit source code information [14] and VTables semantics [5, 7, 13] to make CFI more fine-grained. A very recent work has shown that the above approaches still leave programs vulnerable and argue that unless you correctly extract C++ semantics from source code they will remain vulnerable in the future [17].

This paper aims to provide the final say on VTable protection, by tightly constraining virtual function pointers (vfp) to the VTables corresponding to the classes intended by the programmer—as defined by the semantics of the C++ language. We begin by examining a recent compiler-based CFI approach, namely VTV [20], and evaluate it to determine whether its own vfp restrictions are accurate.[1] We proceed by extending VTV to apply even tighter restrictions to the available VTable targets and argue that our approach is optimal. More precisely, we aim at offering the best protection possible to vfps in a context insensitive fashion. We build our solution into VTV and evaluate it by means of a new

[1]VTV is now a standard compiler option also used in production systems.

framework for testing it. Using the framework, we experimentally show that it is the strictest access policy possible for VTables without breaking legitimate code. Last but not least, our solution is *faster* than the original VTV implementation.

Our investigation exposes *three* weaknesses of existing schemes. First, we find that existing solutions fail to precisely identify the object types associated with a virtual call-site, even in the presence of source code. Second, we find that even state-of-the-art solutions, like VTV, handle multiple inheritance over-permissively. Normally, every class has its own VTable and base classes contain all the VTables of their subclasses. When a class $C$ inherits from multiple classes, VTV extends the VTables of its base classes to include, and thus share, all entries in their individual VTables. I.e., "sibling" classes share VTable entries. This is another example where control-flow integrity is loosely enforced. Finally, we identify a fundamental error in the assumptions made by other solutions. Previous approaches operate on the premise that allowable control-flow transfers at call-sites (i.e., where a method of an object is invoked) can be determined solely based on the type of the object pointer involved. We show that this assumption is false and more information must be extracted from the call-site to reach optimal protection.

We use our observations to design a new VTable-protection scheme that uses information available during compilation to extract the most restrictive set of VTables that should accessible at a virtual call-site within the code. We implement this enhanced, fine-grained design on top of VTV and evaluate it by creating a testing framework that exhaustively explores all possible combinations of class inheritance and method invocation to demonstrate that our technique provides the best possible defense for VTables. We also experimentally test our approach by compiling and running the Chrome Browser to demonstrate that our modifications do not break complex, real-life applications. The evaluation of our prototype also shows that our scheme is faster than the original VTV scheme.

We summarize our contributions below:

- We identify limitations in the design and implementation of current VTable protection schemes, including the primary industrial implementation, VTV. (Section 2.2)

- We identify key design decisions that should be accounted for, when dealing with VTable protection. This also includes a definition of optimal (minimal) VTables sets that should be accesible at each point in the program. (Section 3)

- We develop a practical testing methodology to evaluate VTable protection schemes and to highlight potential limitations.(Section 5.1)

- We implement a prototype of the proposed protection scheme and evaluate it on a large, complex real-world application, the Chrome, browser, in terms of security *and* speed. (Section 5.2)

# 2. VTABLE PROTECTION TODAY

In this section, we discuss VTables and their protection in current solution, as well as the reasons why such defenses are not as tight as they should be.

## 2.1 C++ dynamic dispatching

Function polymorphism in object oriented languages like C++ needs a way to dynamically resolve the appropriate method implementation based on the dynamic type of the object. For instance, if $B$ and $C$ are subclasses of $A$, and both implement a method `f()`, we can initialize any reference to $A$ with an object of either type $B$ or $C$. However, when we now call the method `f()`, we execute either `B.f()` or `C.f()`, depending on the dynamic type. The typical solution to this problem is to group all methods of a particular class into a table of function pointers, called VTable in C++. Subclasses extend the VTable of their base class with new entries for newly defined methods, while previously defined methods feature updated function pointer entries. During object construction, a pointer towards the appropriate VTable is stored within the object. This object, together with the VTable it points to, allows the compiler to select the appropriate polymorphic method variant, irrespective of the compile-time type.

C++ also features complex inheritance strategies, such as multiple and virtual inheritance, that affect VTable usage. For example, assume that class $B$ simultaneously inherits from multiple base classes $A1, A2, \ldots, An$. The VTable of $B$ can only extend the VTable of the *primary* base class for $B$. Otherwise entries in the VTable would have to overlap on the same offset. To allow using the class $B$ in place of all of its parents, *secondary* VTables are associated with it, corresponding to each non-primary base. These secondary VTables are also inherited in all of the subclasses of $B$. Virtual inheritance is a necessary side-effect of supporting multiple inheritance. The latter allows the same base class to be inherited multiple times via different inheritance chains. This is potentially undesirable behavior. Virtual inheritance solves the problem, by ensuring that a given base class is inherited a single time in any further subclasses. Virtually inherited base classes also trigger the generation of their own secondary VTables to be used when accessing methods from this particular base. For more details about VTable interaction with inheritance, the reader can refer to the C++ ABI documentation [1].

While each compiler generates its own code for supporting VTables, a common approach is to store at the beginning of an allocated object a pointer to the object's VTable. Normally, VTables themselves are stored in read-only memory to prevent tampering by attackers. However, C++ objects can be allocated on the stack and the heap, which are both writable. Therefore, the pointer that points to the VTable can be overwritten by leveraging a software bug (like a buffer overflows or user-after-free bug [3]), for instance to make it point to VTable-like data the attacker controls.

## 2.2 VTable integrity and limitations

It is evident that protecting VTable pointers will make software exploitation dramatically harder. As a result, recent security conferences abound with publications on how to protect VTables. Although all these proposals apply some notion of CFI, some are VTable-agnostic [22, 23], while others target VTables specifically [5, 7, 13, 20]. In the following we focus on the second category, as their understanding of VTable semantics allows the protection to be more refined than generic approaches.

Since all recent VTable protection schemes [5,7,13,20] share a similar architecture, we map them onto the following model to analyze their strengths and weaknesses.

1. Statically search for VTable based call-sites.
2. Statically generate VTable sets that could be associated with each class/call-site.
3. Statically identify the class type used at each call-site.
4. Enforce that run-time VTables are part of the statically inferred set at each call-site.

In short, each protection scheme aims to associate a set of valid VTables to each particular call-site. By enforcing that the run-time VTable belongs to the statically generated set, it aims to limit the influence of the attacker on the control-flow. The sets should contain all possible VTables that could be used at the given call-site. In order to avoid having a new set for every call-site in the program, the sets are typically grouped together, based on type information. The intuition is that call-sites with the same static object type have access to the same VTables.

Recent binary- and source-based solutions have been successful at solving the first and fourth points in this model, and we do not cover them much in this paper. Instead, we focus on the second and third points, where we identified limitations in all existing solutions.

### 2.2.1 Generating VTable sets

VTable sets contain all VTables that a particular call-site can legitimately target. The best way to generate them is through analyzing the class hierarchy, since it defines how valid C++ code interacts with VTables.

**Binary level** Current binary based approaches are limited by the information that they can extract for a particular call-site. Since type information is unavailable in closed-source programs, binary protection schemes cannot differentiate between the legitimate targets for different call-sites. As a result, they typically use a single VTable set that contains all VTables accumulated from the binary. While this stops many existing exploits, attackers are still able to corrupt the program flow. For example, a call-site calling a virtual method of 0 arguments can be used to call into a method with 3 arguments. Attackers can exploit this pattern to perform stack pivoting on Windows based systems as shown by Göktaş et al. [11]. Prakash et al. [5] try to extract limited semantics from the call-site to support more specialized VTable sets. However, the authors admit that even their advanced policies lead to VTable sets being larger by a factor of 2X, compared to the existing source-based VTable protection in GCC.

**Source level** VTV [20] takes a class-based approach for generating the sets. It associates a VTable set with each class which includes all its VTables and the VTables of its subclasses. This is intuitive, as all subclasses can be used at the call-sites of their base classes. SafeDispatch [13] provides two alternatives for generated VTables: (a) the VTV scheme, and (b) method-based VTable sets. In case of the latter each virtual method is associated with a VTable set of its own, based on method overloading in the subclasses. Although this alternative reduces the run-time overhead, according to the authors it is weaker or equivalent to the VTV approach and we will not focus on it in the remainder of this paper.

While VTV's solution of adding all VTables of subclasses to the set is intuitive at first glance, it fails to arrive at the right set in case of multiple inheritance. In this case,classes inherit completely unrelated functionalities using multiple unrelated VTables. Code at the call-site assumes that the appropriate casting mechanisms have been applied to select the desired VTable of the object before making the call. However, VTV allows attackers to inject a different (and mismatching) VTable of the same class at the call-site, undermining its semantics. We present the security impact of this limitation on the Chrome browser in Section 5.2.2 (with callsites erroneously giving access to thousands of VTables).

In this paper we propose an in-depth analysis of class hierarchies and VTable generation policies. Based on the analysis, we then propose a VTable set generation policy, which does not suffer from false positives and still supports all valid C++ semantics (Section 3.2).

### 2.2.2 Call-site type inference

While this step is typically left as an implementation detail in previous papers [5, 7, 13, 20], we believe that it should be an integral part of the design and evaluation of VTable protection. Since type inference is the key for associating call-sites with particular VTable sets, this component has a direct influence on the number of allowable VTables. Even if the VTable sets are generated to be optimal, it is enough to associate the wrong (overly conservative) set at a particular call-site to increase the attack surface. For example, imagine a class hierarchy, similar to Java with a common root class `Object`. Since all other classes are based on `Object`, conservatively associating a call-site with the root class allows an attacker to use of all VTables within the system, which can lead to a significant and undesirable attack surface increase.

Since type inference is still very difficult to perform at the level of binaries, most binary solutions forego call-site type inference entirely and limit themselves to a single VTable set that they associate with each call-site. As mentioned before, doing so allows attackers to leverage all VTables within the system at every virtual call-site, which might be enough for future exploits.

We expected source based solutions to have solved

the problem of type inference completely, as they have access to the underlying code, but this turned out to be wrong. For instance, VTV [20] turns out to be overly conservative in this stage, and as a result not nearly as effective in validating VTable pointers as it could be. Specifically, we observed that, in the case of multiple and virtual inheritance, the type inference scheme in VTV is prone to associate call-sites with base classes of the type specified in the source. We discuss the problem of precise call-site type inference in detail and provide a compiler-agnostic solution in Section 3.1.

# 3. SHRINKWRAPPING THE VTABLES

## 3.1 Precise call-site type inference

As precise VTable protection relies on associating the appropriate VTable set with each call-site, call-site type inference is crucial. If type inference is too conservative, the call-site might be associated with a super-class instead, allowing the use of VTables with no relationship to the given call-site. We found that VTV [20], currently the state-of-the-art in VTable protection, suffers from overly conservative type inference and later in Section 5.2.2, we highlight the impact of VTV's conservative nature when protecting call-sites in complex programs like Google Chrome. In the following we analyze the root cause of type inference issues within GCC. We also map our observations to other compiler frameworks to suggest a generic design for future VTable protection implementations.

The source of the conservativeness stems from the fact that the core of a typical C/C++ compiler is built to handle a wide range of language front-ends, and thus oblivious to language specific features, such as C++ VTables. It is the responsibility of the C++ front-end to transform VTable-based method calls into traditional calls. This process involves implicitly casting the object to its base class which explicitly implements the desired method. When performing instrumentation within the core of the compiler, this pattern is impossible to separate from explicit casts and field accesses. For example, in Figure 1 a call-site using a pointer of type *C* accessing a virtual method inherited from *B* will be associated with type *B* or even *A2* due to a bug in the VTV implementation. This inherently enables access to a larger number of VTables than desired by the programmer. We discovered these issues, while analyzing the code in GCC, but the problem applies to most compiler frameworks. For example, the core of Clang uses the language agnostic LLVM intermediate representation, which also lacks type information for virtual method call-sites as mentioned by Jang et. al. [13]. Likewise, the Microsoft compiler also uses separate language-specific front-ends C1 (C) and C1XX (C++) to parse the code and transform it into an intermediate representation processed by the C2 back-end, and while compiler internals are not known, it seems likely that it loses virtual method semantics at the level of the back-end as a result of normalization with raw C code. With this analysis we hope to draw attention to the issue of type inference in C++, so that future VTable defenses (regardless of the targeted compiler) will use *precise* call-site type inference.

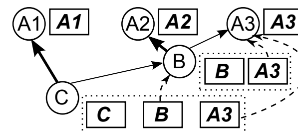


**Figure 1: Example class hierarchy. The classes are represented by circles. The solid arrows show parent relationships between classes, the thicker ones signaling the primary parent. The VTables of each class are represented by the rectangles next to it. The dashed arrows signal the class from which a particular VTable was inherited. The text in each rectangle is the type associated with the corresponding VTable, based on the inheritance.**

As a baseline solution, we propose using the earliest possible stage within the C++ front-end and parser to perform type inference, and propagate the information to the instrumentation code via internal compiler annotations. This allows the instrumentation to reside either in the front-end or the core of the compiler, without affecting the precision of type inference. In our setup we annotate the access to the VTable pointer itself (as it is generated in the front-end) and transform the annotation into a VTable check at instrumentation time. It is even possible to include further analysis in the compiler to infer a more restricitive type to associate with the call-site based the static pointer-tracking, but we leave this up for future work.

## 3.2 Legitimate VTable targets

As described in Section 2.2, VTV [20] uses a coarse definition of allowable VTable sets for each call-site. If class `B` is a sub-class of `A`, then all virtual call-sites using the latter type are allowed to use any of the VTables found within `B`. However, Section 2.1 showed that multiple and virtual inheritance can result in classes having a large range of VTables. In this case some of the VTables in class `B` are not inherited from `A`, and should thus be inaccessible at a call-site using type `A`. In Section 5, we show that VTV inadvertently allows some call sites in the Chrome browser access to thousands of VTables. We introduce a pair of concepts to model the relationships between VTables: the type of a VTable and the parent relationship between a pair of VTables. They form the basis for generating provably optimal VTable sets for each call-site.

*Concepts.* First, we define the *type* of a VTable to be the base class of the object responsible for triggering the generation of this particular VTable. For example, the *primary* VTable of a class has the same type as the class. In the case of multiple inheritance, every *secondary* base (including inherited ones) generates its own VTable, with the given class as its type. An example of the type association is presented in Figure 1. Second, we define a *parent relationship* between VTables of different classes.

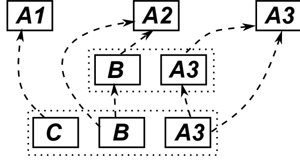


**Figure 2: Visualization of the parent relationship of VTables for the class hierarchy from Figure 1. This figure leaves out the classes, preserving only the VTables. The dashed arrows represent the parent relationship between two VTables. The VTable in `A2` is a parent for both VTables of type `B`, since `A2` is the primary parent of `B`, and the primary VTables of the latter is an extension of the VTable inherited from `A2`. The same between the VTable of type `C` and `A1`.**

A VTable `X` in class `A` is a parent of VTable `Y` from class `B` if and only if:

- A==B or class `A` is a base class of class `B`.
- VTable `Y` is inherited from class `A`.
- VTable `Y` matches or extends VTable `X`.

Analogously VTable `Y` is a descendant of VTable `X` if VTable `X` is a parent of VTable `Y`.

VTable extension is defined in C++ as generating a new VTable that starts out with the same layout as the original one, but with additional entries appended. VTable extension allows efficient type-casting without the need of generating additional VTables. An extended VTable can always be used in place of the original, as the relevant part matches in layout.

To identify when a VTable extends another, we take a look at inheritance rules for both multiple and virtual inheritance. The primary VTable of a class always extends the primary VTable of its first non-virtual base class. Extension continues transitively as long as non-virtual inheritance is involved. The example in Figure 2 shows the parent relationships between the VTables introduced in Figure 1. Because class `A2` is the primary base class of `B`, its VTable is extended as the primary VTable of `B`, leading to a parent relationship between the two VTables.

In case of virtual inheritance things get more complicated. Virtual base classes are only inherited once in sub-classes. As a result the VTable of this base is only extended a single time, even in the face of diamond inheritance as presented in Figure 3. While the VTables of both `B1` and `B2` extend the VTable of `A`, this property is not transitively propagated into the sub-class `C`, only the primary VTable of `C` extends the one inherited from `A`. In the second VTable of class `C` the entries corresponding to `A` are eliminated from the VTable. As a result, in the face of virtual inheritance, explicit analysis of the VTable content is necessary to identify extension between two VTables.

*Usage.* Based on this definition of the parent relationship, it is intuitive that only descendant VTables can be used in place of their parents at any particular virtual

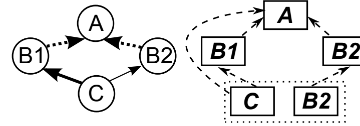


**Figure 3: Example of diamond virtual-inheritance, where by courtesy of the virtual inheritance only a single copy of the top class (`A`) is inherited within `C`. While the primary VTable of class `B2` does include all entries corresponding to parent A, when this VTable is inherited into `C`, the entries are cleared out. Irrelevant of the type-casting chain used to convert an object of type `C` into type `A`, the primary VTable of `C` will always be used to access the corresponding methods. The parent relationship follows this semantic, since only one of the VTables within class `C` has the primary VTable of `A` as its parent.**

method call-site. The layouts of the descendants always match the layout expected at the call-site, resulting in a successful method invocation. Any VTable which is not a descendant should *not* be allowed at the call-site, as it was either inherited from a different, unrelated base class, or its layout does not match the expectations at the call-site. These properties make the parent relationship the perfect basis for generating allowable VTable sets, based on C++ class semantics.

# 4. STRONGER VTABLE PROTECTION

We introduce stronger VTable protection through (i) a simple extension to VTV, and (ii) an optimal solution.

## 4.1 An extension to VTV

As a first step toward better security, we use the concepts presented above to redefine the restrictions enforced by VTV [20] and increase the strictness of its protection. The existing implementation traverses the class hierarchy at compile-time to identify targeted VTable sets for each class within the system. We extend this mechanism by a set of additional restrictions to limit the contents of this set further, while maintaining full compatibility with all C++ semantics.

We propose the following simple policy for VTable sets at a call-site corresponding to a particular class:

- All VTables of the class are part of the set.
- All descendant VTables of the above are also part of the set.

This policy ensures that VTables in the sub-classes, with no relation to the given class, are never added to the set.

Implementing the policy is straightforward, only requiring information about VTable type and parent relationships. These are extracted from the class hierarchy, based on their definition from above. As with the call-site type inference, we believe that the proper testing methodology would have highlighted the limitations of the existing design earlier in the development process.

## 4.2 Optimal VTable protection

While the previous solution is intuitively strong, it is limited by a core design decision, common to both VTV [20] and SafeDispatch [13]. Both these papers assume that VTable-level protection is based on a *single* piece of type information, particular to the call-site. In contrast, when a call is performed using a virtual method, the compiler knows *two* things: the type of the object on which the method is called, and the particular VTable of this object, where the method can be found (represented by the type of the VTable for example). When limiting the protection scheme to a single type being associated with the call-site, information is inadvertently lost, degrading the precision of the protection. We propose leveraging both pieces of information, which enables a fine-grained and provably optimal VTable protection scheme. The new design is defined as follows:

- Each call-site is represented using a class-type and VTable pair.
- The set is initialized with only the call-site VTable.
- All descendant VTables of the above are also added to the set.

This scheme is optimal, since the VTable sets only include the entries mandated by the C++ semantics. This is guaranteed by the definition of the parent relationship from Section 3.2. Every entry from in the set can potentially be used at the call-site, via a valid type-casting chain. Figure 4 shows a comparison between the original VTV, the extension we presented in Section 4.1 and our new fine-grained solution (see Section 5 for a quantitative analysis).

Implementation-wise this new scheme can also be added on top of VTV, while preserving much of the core code intact. Call-site type inference needs to be extended to also provide information about the VTable in use, while the VTable sets are also kept track of with finer granularity. The end result is a slight increase in the code size to support the newly defined additional VTable sets. Notice that this policy results in smaller individual set sizes, which leads to a reduction in run-time overhead.

Besides being an optimal VTable protection policy, fine-grained verification also has the advantage of offering guaranteed protection for VTable-based call-sites. In the case of VTV, it is still possible to call functions unrelated to the current call-sites, or to use VTable offsets that overflow the VTable in use (as highlighted in Section 5.2). Fine-grained protection guarantees by design, that the selected offset is always valid within all accessible VTables (since the layouts match). Furthermore, the function pointer at the offset always refers to the method specified in the source code or one of its overloaded variants. This means that attackers are unable to take advantage of mismatches in function prototypes or argument usage to further corrupt the program flow. All potential targets are also theoretically valid at the particular call-site, thus it is the responsibility of the programmer to design the methods with compatible semantics. While this does not stop all exploitation attempts against the program, it does eliminate the use

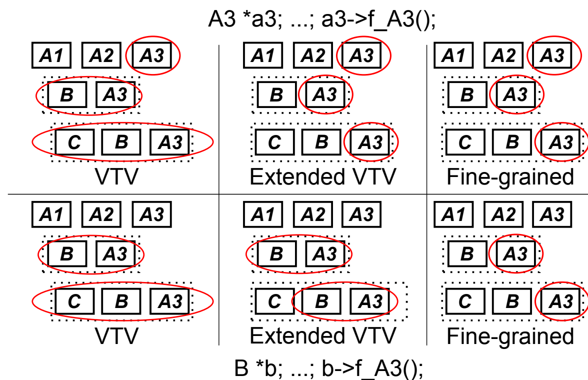


**Figure 4: Example of VTable sets for two particular call-sites. The class hierarchy is reused from Figure 1. Each class defines a function *f_className*, while it does not overload any of its parents. The red circles are used to highlight the VTables added to the set accessible at the particular call-site for each protection scheme. The top row shows the VTables sets associated with a call-site of type *A3* using method *f_A3*. In case of the original VTV all VTables of all descendants of *A3* are added to the set, including VTables inherited from *A1* and *A2*. The extended VTV ensures that only the VTables generated due to *A3* are part of the set. For this call-site there is no difference between the extended and the fine-grained versions. The second call-site involves the type *B*, but the same method *f_A3*. The original VTV shows the same problem as before, but in this instance even the extended version is not optimal. Since the call-site cannot differentiate between the two VTables of *B*, both of them and their descendants need to be added to the set. By using the fine-grained approach, we identify that the call-site leverages the second VTable of *B* specifically. Thus we only add this particular VTable its descendant to the set.**

of VTable-based call-sites as control-flow hijacking targets. Since these represent 90% of all indirect call-sites in modern C++ programs [20] the vulnerability surface is reduced significantly, while maintaining overhead low enough to be acceptable by software development companies. We believe that other vulnerability vectors should also be protected using similar defenses focusing on the underlying semantics, instead of generic, coarse-grained protection mechanisms.

# 5. EVALUATION

## 5.1 Microbenchmark evaluating correctness

The complexities of VTable inheritance policies within C++ make it difficult to guarantee a correct implementation using only a simple intuitive design. Thus we propose using a custom-designed microbenchmark to prove correctness in both the proposed and future VTable protection mechanisms. The point of the microbenchmark is to cover all inheritance scenarios as well as call-sites to

ensure that the VTable protection implementation does not break valid C++ semantics, while also not including unneeded VTables in the corresponding sets.

We design the microbenchmark to cover all valid class hierarchies, including combinations of multiple and virtual inheritance. Since class hierarchies can have infinite size in theory and the possible number of class inheritance combinations increases exponentially, we define some practical limits to the class hierarchies we generate. The first limit is the maximum number of classes included in the hierarchy. The second option is the maximum number of base classes. Given these limits we generate every possible class hierarchy and include it in a source file of its own. These files make up the microbenchmark for evaluating the correctness and precision of VTable protection.

In order to trigger the use of all potential VTables, we create objects corresponding to each class and cast these objects to all possible valid dynamic casting targets. Finally we introduce VTable based call-sites for each of the cast results. The benchmark is evaluated along 3 axes: (i) precision of call-site type inference, (ii) correct execution with respect to C++ semantics, and (iii) optimality of VTable set content.

The call-site type inference is validated by statically analyzing the binary generated for each set. All VTable verification calls within a testing function corresponding to class `X` should only use VTable sets associated with `X`. This is defined by the program logic, which specifies that the argument points to a valid object of type `X`. Any outliers to this rule raises an error in the benchmark to signal that type inference is overly conservative. Correct execution is checked by running all binaries and monitoring that VTable verification does not raise errors for any combination of run-time object and call-site.

While previous papers considered enough the binaries to execute successfully, with ShrinkWrap we aim for optimal VTable sets. Thus we also evaluate if none of the VTable sets include unused entries. By construction, the microbenchmark covers all combinations of objects and call-sites that may occur within valid and semantically correct C++ code. This property of completeness allows us to identify unused entries in the VTable sets correctly and to report them. A VTable protection is only considered correct and optimal if it is capable of running the benchmark successfully, while reporting no unused entries in its VTable sets.

The existing implementation of VTV [20] in GCC 4.9.2 fails the microbenchmark both in terms of call-site type inference, as well as the optimality of the VTable sets. Our proposed redesign of call-site type inference fixes the first problem, but only fine-grained VTable set generation is capable of passing all three aspects of our benchmark. This means that our proposed solution not only supports full C++ semantics, but it is optimal in terms of the VTable set contents. We found the benchmark incredibly valuable during the implementation process to identify corner-cases within the C++ standard. We have observed several instances, where Google Chrome would execute correctly with a particular implementation variant, while the benchmark would fail. As a result of our experiences, we highly recommend that all future research on VTable protection leverages this benchmark or a similar one. This will ensure that theoretical solutions are backed up by a correct implementation making the solutions sound in face of a determined attacker. To facilitate this aspect, we will make our micro-benchmark public as open-source code.

## 5.2 Chrome

### 5.2.1 Building Chrome with VTable protection

While building Chrome with precise call-site type inference, we noticed that VTable verification fails in a single place within the Skia library for the Chrome version we wanted to test. The particular fragment of code was eliminated in a later version, using the patch with identifier *76f5cc6e9e87ff247c3ef1f4b3fb03668db06e2e*, as it was deemed unnecessary and restricting the class hierarchy. For the evaluation we changed two lines of code to eliminate the artifact without affecting run-time behaviour. The code itself is a classic case of bening type confusion, where a base class is statically casted into one of it's sub-classes, when it truly is just an object of the base class. The code does not crash with current compilers as the layout of the objects stays identical from a VTable perspective, since the sub-class does not implement any new methods. The cast operation is still invalid considering high-level C++ semantics and should be avoided in clean code.

We compiled the Chrome browser (version 42.0.2305.0, 64 bit) without VTable protection, with the original VTV and with the proposed enhancements deployed in multiple stages. This resulted in the following variants used throughout the evaluation:

- Original: VTV as it is implemented in GCC 4.9.2.
- Call-site: Applying precise call-site type inference to "Original" (described in 3.1).
- Extended: The proposed extension to VTV filtering on top of "Call-site" (described in 4.1).
- Fine-grained: The proposed fine-grained filtering scheme on top of "Call-site" (described in 4.2).

We plan to release all of these variants as patches to GCC 4.9.2, which will hopefully help developers to swiftly integrate the changes into the main development tree. Having the code as open-source will also allow other researchers to scrutinize our work in the future.

### 5.2.2 Security evaluation

The security evaluation of the original VTV and the different proposed enhancements is performed along two axes. The first is the size of the VTable sets allowed at each of the virtual call-sites within the program. The lower this number is, the smaller the control an attacker can exhibit when corrupting a VTable pointer. This simple metric for both source- and binary-based solutions, allows for easy comparisons against existing and future systems. For example, Prakash et. al. [5] also use this metric to evaluate their solution, vfGuard. The main issue with this metric is the lack of a proper baseline, since C++ semantics require that virtual call-sites

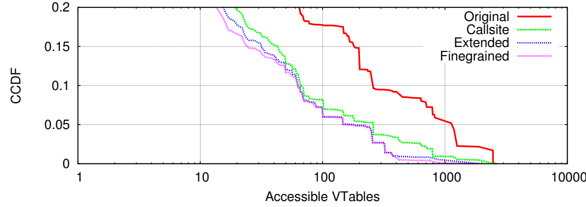


**Figure 5: Complementary cumulative distribution function between the number of call-sites and the number of VTables they allow. The X-axis represents the number of VTables allowed at a call-site. The Y-axis represents the number of call-sites that use more than the given number of VTables.**

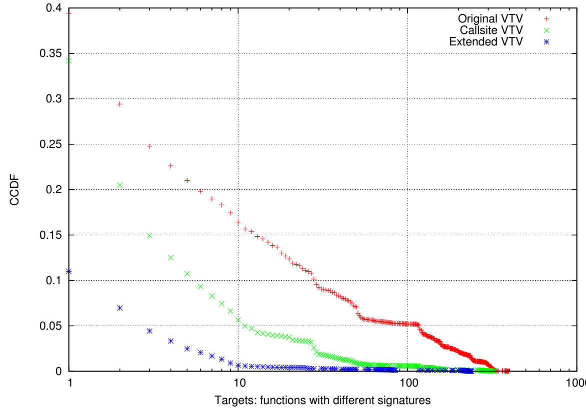


**Figure 6: Complementary cumulative distribution function between the number of call-sites and the number of methods (by name) they can target. The X-axis represents the number of methods (by name) that a call-site can target. The Y-axis represents the number of call-sites that use more than the given number of methods (by name).**

allow a set of VTables based on the class hierarchy. To the best of our knowledge, ShrinkWrap is the first work to extract these sets precisely from the source code.

In an attempt to provide an alternative baseline, we propose a second metric focused on functional semantics instead of raw VTables. At each virtual call-site we analyze the set of methods accessible via the set of VTables allowed within the protection scheme. Instead of counting the addresses, we count the number of unique method names that we encounter at each virtual call-site. Destructors are all counted as a single method name. A method name encompasses a group of polymorphic methods with compatible semantics. The intuition is that within the source every virtual call-site is specific to a given method name as specified by the developer. Precise VTable protection scheme is expected to enforce this property, otherwise an attacker can gain significant leverage, by diverting the control-flow to an unexpected method body, different from the one specified in the source code. This mechanism has been used [11,12] for establishing the initial control over

```
SetRemoteSSRCType(int, webrtc::StreamType,
      unsigned int)
SetFECStatus(int, bool, unsigned char, unsigned
      char)
RegisterObserver(int, webrtc::
      ViECaptureObserver&)
StartRender(int)
DeregisterEncoderObserver(int)
LastError()
ReceivedBWEPacket(int, long, unsigned long,
      webrtc::RTPHeaderconst&)
```

**Figure 7: List of methods that an attacker can target at a particular call-site within Chrome. Even with the Original VTV enabled, the attacker can redirect execution to any of these 7 families of methods.**

the return address, thus we consider it a serious threat to security.

The analysis corresponding to these two metrics are depicted in Figure 5 and Figure 6. One key observation is that around 2.5% of call-sites allow access to > 2500 VTables (> 7.5% of all VTables in Chrome). While the percentages are not impressive at first sight, one has to take into account the size of the Chrome binary (without VTV) used for these evaluations being 115MB in size after symbols are removed. This number suggests that an attack similar to the one presented by Schuster et. al. [17] might still be viable in the presence of VTV. The latter showed that main-loop gadgets were found in binaries as small as 1MB in size, which is smaller than 2.5% of the Chrome binary that we observe as being highly dangerous. The paper also claims that the other gadgets were successfully found within binaries smaller than 20MB. In the case of VTV, 7.5% of Chrome corresponds to somewhat less code, but still within the same magnitude. These results suggest that VTV can be susceptible to the attack or a future variant of it.

Another point of concern is the large number of polymorphic method families accessible at each call-site, when using the existing implementation of VTV. More than 17% of call-sites allows attackers to target at least 10 different method families. In Figure 7 we present a concrete example to show the wide range of semantics accessible to an attacker at one such call-site. Some call-sites go as far as to allow access to more than 300 different method families, which is a significant attack surface, with potential to be exploited by resourceful adversaries. Figure 8 highlights an example of a class hierarchy, inspired by complex C++ projects, where an attacker can change the call-site's semantics for performing undesirable functionalities while original VTV is in place.

Precise call-site type inference has a significant impact on both metric, reducing the average number of VTables and methods accessible at call-sites, however it still suffers from corner-cases that could potentially become vulnerable in the future, e.g. around 6% of call-sites still allow access to 10 different method families or more. The Extended variant comes close to achieving the desired strictness according the method count metric, but it comes short in a small set of corner-cases.

```
// Compile with GCC 4.9.2 VTV enable
// g++ -fvtable-verify=std -mpreferred-stack-
    boundary=2 -m32
#include <unistd.h>
#include <stdlib.h>
struct RefCounted {
        virtual void addRef() {}
        virtual void delRef() {} };
struct Logged {
        virtual void log() {} };
struct ProcessWrapper : virtual Logged,
    virtual RefCounted {
        virtual void run(char *path) {
            execlp(path, path, NULL);
        } };
int main(int argc, char ** argv) {
    // --Original Object Pointer--
    RefCounted *ptr = new RefCounted();
    // --Memory Corruption--
    ptr = (RefCounted*)(void*)(new
        ProcessWrapper());
    __asm__("push %0\n"::"r"("ls"));
    // --Hijacked Call-Site--
    ptr->delRef(); }
```

**Figure 8: Proof-of-concept attack against VTV. An attacker corrupts an object pointer on the stack, changing its target to a subclass. The call-site for *delRef* will use the wrong VTable from within *ProcessWrapper*, since the appropriate up-casting is missing during the corruption. The call-site will redirect to the *run* method, taking the last stack entry as its argument (the string *"ls"* set up by the attacker).**

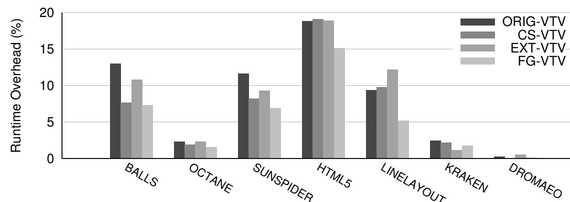


**Figure 9: Performance evaluation of the proposed scheme. The overhead imposed by the proposed fine-grained VTable protection compared to the original VTV.**

Fine-grained VTV is not shown in Figure 6 as all call-sites restrict access to a single method family, as desired when deploying strict VTable protection. The average VTable set size is also reduced to 27 compared to the average of 146 with the original VTV. This result is key to point out for binary-based VTable protection schemes that currently compare against VTV as a reference solution (such as [5]).

### 5.2.3 *Performance overhead*

Besides the security evaluation, it is also important to ensure that the proposed enhancements do not affect the low overhead offered by the original VTV implementation [20]. We evaluate the overhead imposed by the different variants using the Chrome browser. Our testbed is an HP Z230 i7-4770 3.4 GHz machine running Ubuntu Linux 12.04.5 with ASLR and turbo mode off to reduce the possible noise during the measurements. We compiled the Chrome browser (version 42.0.2305.0, 64 bit) with the default release configuration as well as with the different variants of VTable protection added to the configuration. We evaluated performance across a series of popular browser benchmarks. Every benchmark is executed 10 times and with the average being taken as the final value. The results are depicted in Figure 9 and, as expected, the fine-grained VTV, the scheme we propose in this paper, performs better than the other variant of VTV. This mainly stems from the fact that the available targets at virtual call-sites are reduced the most in fine-grained VTV. Therefore, we stress that our proposal does not sacrifice performance for better security, but, instead, it is better in both performance and security, compared to the original VTV.

## 6. RELATED WORK

Protecting return addresses stored on the stack [9] and support of non-executable data by many hardware processors and operating systems have raised the bar in software exploitation. Attackers have no way to inject code anymore, and they have to reuse existing code [18] by combining multiple bugs (one for taking control and one for leaking the process layout [19] for overcoming randomization [16]). Sophisticated exploits appeared and drove the community to seek a generic principle that could eventually offer sound protection for software, namely Control-Flow Integrity (CFI) [2].

CFI suggests that binaries should be able to exercise only the control-flows that are allowed by the program's source. All indirect branches that happen at run-time should not be arbitrarily influenced by user input. Applying CFI in real-world software is challenging, since legacy applications cannot be recompiled, and even in the case where source is available, discovery of the complete control-flow graph is challenging [14], while the performance of validating call targets can produce overheads. Therefore, relaxed implementations [22,23] were proposed that could be applied directly in binaries, but as it was quickly demonstrated, these implementations sacrificed security and therefore they are potentially exploitable [6,10,11].

Another approach for enforcing CFI is by leveraging certain hardware features, such as the Last Branch Record (LBR) debug registers, for performing anomaly detection in the last indirect branches a process has followed [8,15]. Unfortunately it seems that an exploit can evade detection by inserting legitimate-looking gadgets in its ROP chain [6,10,12].

A particular set of CFI solutions focus on protecting only VTable pointers. The details of different variants are discussed in Section 2.2.

Last but not least, researchers have proposed methods for defending against use-after-free vulnerabilities based on custom allocators [3], which do not allow a memory area to host different type of objects during the life cycle of the process, or patching the developer's code for keeping track of dangling pointers [21]. These techniques protect *only* against use-after-free bugs, and they experience serious memory and computational overheads.

# 7. CONCLUSION

In this paper we revisited VTable protections. Although it was recently demonstrated that binary-based solutions that aim at protecting VTables fail to reconstruct C++ semantics, and thus they are potentially vulnerable, we further argued that even when source code is available, the analysis is not straight-forward. We went through the state-of-the-art industry standard implementations, VTV, and highlighted weaknesses. Based on that, we formally modeled and designed an optimal solution for protecting VTables, and we implemented our proposal in GCC. In addition, we developed a testing methodology that can demonstrate that our analysis is correct, while it can also assist in evaluating similar VTable-protection frameworks. This paper suggests a formal guideline and evaluation framework for any methodology that aims at hardening binaries by protecting VTables.

**Acknowledgment** This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, by the Microsoft Research PhD Scholarship Programme through the project MRL 2011-049.9, by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI "Dowsing", and by the European Commission through the project SHARCS under Grant Agreement No. 644571.

# 8. REFERENCES


[1] Itanium C++ ABI. mentorembedded.github.io/cxx-abi/abi.html.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proc. of the 12th ACM CCS*, 2005.

[3] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proc. of Usenix Security'10*.

[4] S. Andersen and V. Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention, 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[5] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proc. of the 22nd NDSS*, 2015.

[6] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proc. of Usenix Security'14*.

[7] Chao Zhang, Chengyu Songz, Kevin Zhijie Chen, Zhaofeng Cheny, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proc. of the 22nd NDSS*, 2015.

[8] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *Proc. of the 21st NDSS*, 2014.

[9] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of Usenix Security'98*.

[10] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proc. of Usenix Security'14*, August.

[11] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proc. of the 35th IEEE S&P*. IEEE, 2014.

[12] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *Proc. of Usenix Security'14*.

[13] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Proc. of the 21st NDSS*, 2014.

[14] Ben Niu and Gang Tan. Modular Control-flow Integrity. In *Proc. of the 35th ACM PLDI*, 2014.

[15] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc. of Usenix Security'13*.

[16] PaX Team. Address Space Layout Randomization (ASLR), 2003. pax.grsecurity.net/docs/aslr.txt.

[17] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proc. of the 36th IEEE S&P*, May 2015.

[18] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. of the 14th ACM CCS*, 2007.

[19] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proc. of the 34th IEEE S&P*, 2013.

[20] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-edge Control-flow Integrity in GCC and LLVM. In *Proc. of Usenix Security'14*.

[21] Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proc. of the 22nd NDSS*, 2015.

[22] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proc. of the 34th IEEE S&P*, 2013.

[23] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *Proc. of Usenix Security'13*.