VTrust: Regaining Trust on Virtual Calls

Chao Zhang
UC Berkeley
chaoz@berkeley.edu

Scott A. Carr
Purdue University
carr27@purdue.edu

Tongxin Li
Peking University
litongxin@pku.edu.cn

Yu Ding
Peking University
dingelish@pku.edu.cn

Chengyu Song
Georgia Institute of Technology
csong84@gatech.edu

Mathias Payer
Purdue University
mathias.payer@nebelwelt.net

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

***Abstract*—Virtual function calls are one of the most popular control-flow hijack attack targets. Compilers use a virtual function pointer table, called a *VTable*, to dynamically dispatch virtual function calls. These VTables are read-only, but pointers to them are not. VTable pointers reside in objects that are writable, allowing attackers to overwrite them. As a result, attackers can divert the control-flow of virtual function calls and launch *VTable hijacking* attacks. Researchers have proposed several solutions to protect virtual calls. However, they either incur high performance overhead or fail to defeat some VTable hijacking attacks.**

In this paper, we propose a lightweight defense solution, *VTrust*, to protect all virtual function calls from VTable hijacking attacks. It consists of two independent layers of defenses: *virtual function type enforcement* and *VTable pointer sanitization*. Combined with modern compilers' default configuration, i.e., placing VTables in read-only memory, VTrust can defeat all VTable hijacking attacks and supports modularity, allowing us to harden applications module by module. We have implemented a prototype on the LLVM compiler framework. Our experiments show that this solution only introduces a low performance overhead, and it defeats real world VTable hijacking attacks.

I. Introduction

Control-flow hijacking is the dominant attack vector to gain code execution on current systems. Attackers utilize memory safety vulnerabilities in a program and its libraries to tamper with existing data or prepare their own data structures in a target process' memory. When used by the program, the tampered data will redirect benign control-flow to attacker controlled locations. Attackers usually continue by reusing existing code sequences, e.g., Return Oriented Programming (ROP [1]–[3]) or Jump Oriented Programming (JOP [4]), to gain full code execution capabilities on the victim system, despite existing defenses like Data Execution Prevention (DEP [5]), Address Space Layout Randomization (ASLR [6]), or stack canaries [7].

Many defense mechanisms have been proposed to protect programs against control-flow hijacking attacks, including memory safety solutions [8]–[10] and Control-Flow Integrity (CFI) solutions [11]–[23]. Memory safety solutions stop memory corruption and provide a strong security guarantee, but with a high performance overhead (often larger than 30%). Recent work has shown CFI is a practical approach. Researchers have created implementations of CFI that were incorporated into GCC and LLVM [23]. The most recent operating system Windows 10 has deployed a coarse-grained CFI by default [24]. CFI solutions typically either provide a coarse-grained protection, or incur a high performance overhead.

A control-flow hijack attack usually targets return instructions, indirect jumps and indirect calls[1] to control the program counter. . Out of these three, indirect calls, which are frequently used for virtual calls in programs written in C++, are receiving increasing attention from attackers. For example, over 80% of attacks against Chrome utilize use-after-free vulnerabilities and virtual function calls [25], whereas about 91.8% of indirect calls are virtual calls [23]. More than 50% of known attacks targeting Windows 7 exploit use-after-free vulnerabilities and virtual calls [26].

Modern compilers use a table (called VTable), consisting of virtual function pointers, to dynamically dispatch virtual calls. Attackers may tamper with these VTables, or pointers to them, and launch VTable hijacking attacks [27], including *VTable corruption* attacks that corrupt writable VTables, *VTable injection* and *VTable reuse* attacks that overwrite VTable pointers with references to fake or existing VTables (or even plain data). Modern compilers place VTables in read-only sections, defeating VTable corruption attacks by default. But VTable injection attacks are still one of the most popular attacks, and VTable reuse attacks are also practical and hard to defeat [28].

Researchers have proposed several defenses against VTable hijacking attacks. SafeDispatch [29] resolves the set of legitimate virtual functions (or VTables) for each virtual function call site at compile time, and validates the runtime virtual function pointer (or VTable) against this legitimate set. It requires an exact class hierarchy analysis, and involves a heavy runtime lookup operation. Moreover, it requires recompilation of all modules when a new module is added to the application or the inheritance hierarchy changes. FCFI (aka VTV [23]) also validates the runtime VTable against a legitimate set. It supports incremental compilation by updating class hierarchy information at runtime, but also incurs high performance overhead, especially when the class inheritance graph is complex. VTint [27] uses binary rewriting to protect the integrity of VTables, and blocks corrupted or injected VTables from being used, but fails to protect against VTable reuse attacks. The research paper COOP [28] shows that VTable reuse attacks are practical and even Turing-complete in real applications.

[1] In rare cases, attackers may hijack the program via other instructions, e.g., iret. Such attacks are uncommon in the real world.

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23164

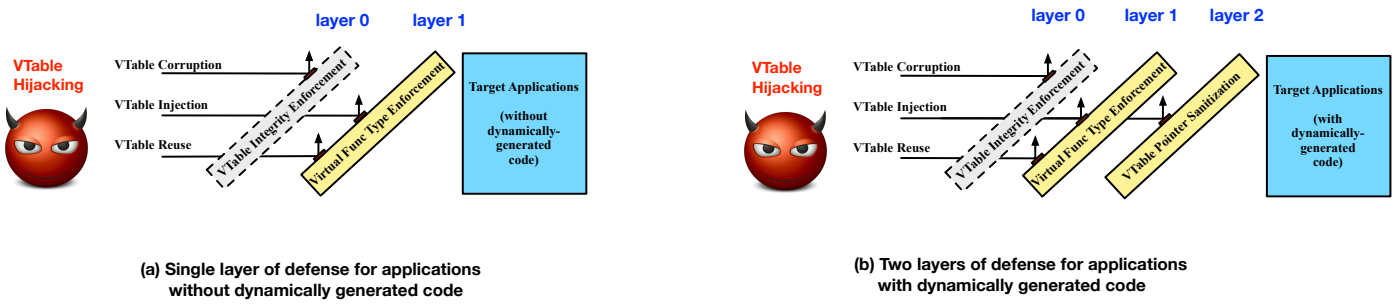

Fig. 1. Illustration of VTrust's overall defense. The layer 0 defense (i.e., placing VTables in read-only sections to protect their integrity) is deployed by modern compilers by default, and thus provides an extra layer of defense for free. The layer 1 defense enforces virtual functions' type at runtime. It defeats all VTable reuse attacks, and also defeats VTable injection attacks if there are no writable code sections. The layer 2 defense enforces the validity of VTable pointers. It defeats VTable injection attacks even if there are writable code sections.

In this paper, we propose a lightweight solution VTrust to protect virtual calls from all VTable hijacking attacks. It first validates the validity of virtual function pointers, and then optionally validates the validity of VTables. As shown in Figure 1, it consists of two layers of defense: (1) *virtual function type enforcement* and an optional layer (2) *VTable pointer sanitization*. In the first layer, we instrument virtual calls with an additional check to match the runtime target function's type with the one expected in the source code. Each virtual function call site is enforced to invoke virtual functions with the same name and argument type list, and a compatible class relationship. Ideally, this layer is able to defeat all VTable hijacking attacks, if we can check virtual functions' type at runtime, e.g., by utilizing RTTI (RunTime Type Information). However, this would cause a very high performance overhead [30].

Our solution encodes the virtual functions' type information into hash signatures, and matches the signatures at runtime. It provides a fine-grained protection against VTable hijacking attacks. Essentially, it is a C++-aware fine-grained CFI policy. As far as we know, all existing signature-based CFI solutions do not utilize the name of virtual functions and its associated class information to protect virtual calls, causing a loss of precision. On the other hand, taking function name and class information into consideration is not a trivial task. We are the first to present such a C++-aware precise signature-based CFI implementation for virtual calls.

Unlike other CFI solutions [11], VTrust supports separate compilation. The signatures can be computed within each module, without any dependency on external modules, allowing us to harden applications module by module. It introduces a very low performance overhead, e.g., 0.31% for Firefox and 0.72% for SPEC2006. It is able to defeat all VTable reuse attacks, including the COOP attack [28]. It is also able to defeat all VTable injection attacks, if target applications do not have writable code (e.g., dynamically generated code). Given that modern systems are protected by DEP, attackers cannot overwrite read-only code to forge virtual functions with correct signatures. Thus this layer of defense is practical and useful in the real world, because (1) most applications do not have writable code, and (2) forging signatures in writable code is hard due to defenses like ASLR and JIT spraying [31] mitigations. We strongly recommend deploying this defense in practice.

For applications with writable code, attackers may launch VTable injection attacks, as shown in Figure 1(b). Traditional signature-based CFI solutions fail to defeat this type of attacks. We provide an extra optional layer of defense, to defeat VTable injection attacks that are launched by forging virtual functions with signatures in writable code memory. In this layer, we ensure that each VTable pointer points to a *valid* VTable at runtime by sanitizing the writable and untrusted VTable pointers. More specifically, we encode legitimate VTable pointers when initializing objects and decode them before virtual function calls. In this way, it blocks illegal (forged) virtual functions from being used, by blocking illegal VTables from being used. Even if attackers can forge virtual functions with correct signatures to bypass the first layer of defense, they cannot call them because VTables are all sanitized.

Since this layer of defense changes the representation of VTable pointers, it ensures we can protect all uses of VTable pointers, e.g., RTTI lookup or virtual base objects indexing, or corner cases like custom virtual calls written in assembly. Traditional solutions, e.g., SafeDispatch [29] and FCFI [23], only cover regular virtual call instructions and are unable to identify these attack surfaces or protect them from being exploited.

We implemented a prototype on the LLVM compiler framework and tested the prototype on the SPEC CPU2006 benchmark [32], and the browser Firefox. The first layer and the second layer of defense introduce an overhead of about 0.31% and 1.80% respectively for Firefox, and an overhead of about 0.72% and 1.40% respectively for SPEC CPU2006. We also evaluated VTrust against several real world exploits targeting browsers, as well as exploits targeting some real world CTF (Capture The Flag) challenge programs. It showed VTrust is able to defeat them all.

In summary, our VTrust defense solution has the following key advantages:

- This solution is *effective*. It provides a fine-grained protection for virtual calls and defeats all different *VTable hijacking* attacks. Case-studies show that it defeats real world exploits.
- The defense is *efficient*. For applications without dynamic generated code, it introduces about 0.72% performance overhead. For other applications, it introduces an overhead of 2.2%.

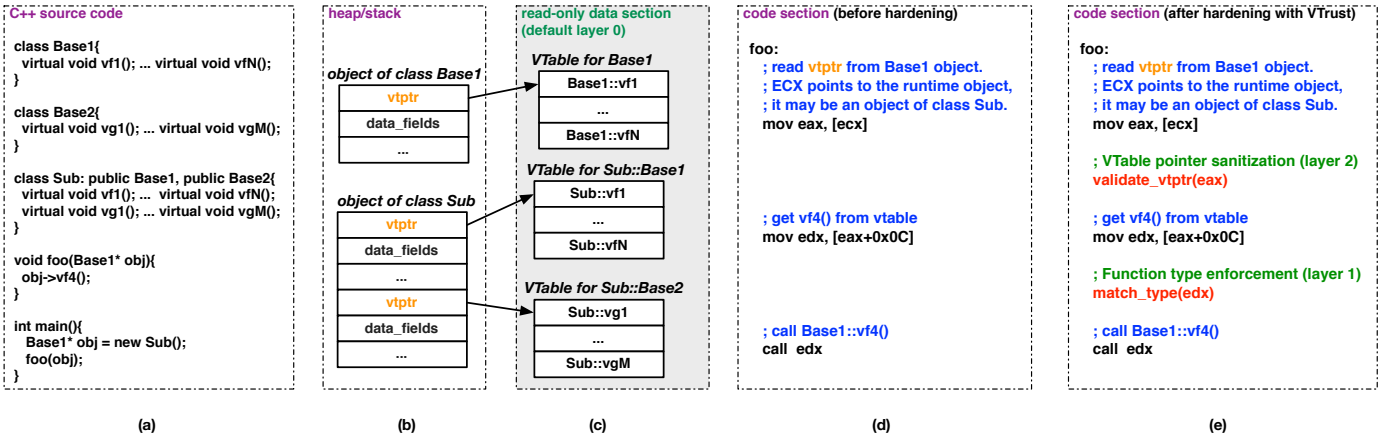

Fig. 2. Illustration of a virtual function call `Base1::vf4()`, including the source code (a), the runtime memory layout (b and c) and the executable code (d), as well as the executable code after deploying VTrust's defense (e). The layer 2 defense is only necessary for applications with writable code sections. The layer 1 defense is sufficient for most applications.

- It has *modularity support*, allowing us to harden applications module by module. Whenever a new module is added, or the class inheritance tree grows, we do not need to recompile other modules.
- Its program analysis process is lightweight and fast. Unlike other solutions, VTrust does not require the whole class inheritance graph of the applications.

The remainder of this paper is organized as follows: we describe the threat model in Section II. The design and implementation of our multi-layer defense solution is described in Section III and Section IV. We evaluate the performance and security of VTrust in Section V, compare our approach with related work in Section VI, and conclude in Section VII.

II. Threat Model

We assume a powerful yet realistic attacker model. Our model gives the attacker full control over all writable memory and allows arbitrary reads from any readable memory. While being conservative, this assumption is realistic as an attacker may use a vulnerability repeatedly, e.g., spawning threads to attack other threads.

A. Defense Mechanisms

We assume that protections against code injection and code corruption are in place (e.g., through DEP/NX bits for non-executable regions). All current operating systems make use of DEP. We also assume that attackers cannot remap memory regions (e.g., setting a VTable region writable, or setting a data region executable) to bypass DEP.

Our defense mechanism protects virtual calls only, we therefore assume auxiliary protections for return instructions, indirect jump instructions and any other indirect call instructions are deployed in the application. For example, we assume that the compiler uses fixed base addresses and bound checks when compiling jump tables (e.g., for switch statements). In other words, we assume indirect control transfers except virtual function calls are all well-protected. Attackers cannot hijack the control flow until they reach the virtual function calls. In addition, non-control-data attacks [33], [34] that may lead to control-flow hijacking are out of the scope of this paper.

B. Attack Surface

As specified in the C++ ABI [35], all virtual functions are dispatched through VTables. As shown in Figure 2(d), a virtual function is dispatched in three steps: (i) read the VTable pointer from the object, (ii) dereference the VTable pointer (plus the target function's index) to get the target virtual function pointer, and (iii) invoke the target virtual function by indirectly calling the function pointer. The latter two steps may be encoded in one instruction, e.g., `call [eax+0x0C]`.

The VTable mechanism enables virtual function dispatch, access to the base class object, and runtime type information (RTTI). However, it also introduces an attack surface. Objects are usually allocated at runtime and stored in writable memory (e.g., on the heap or stack), so the VTable pointers are untrusted and may be overwritten by attackers. As a result, the target virtual functions read from VTables are untrusted and may be hijacked.

Any successful attack against virtual function calls must either (i) corrupt a VTable or (ii) a VTable pointer, to change the target virtual function pointer. *VTable corruption* attacks modify VTables directly, overwriting function pointers in VTables with attacker-controlled values. For this attack vector the adversary directly controls the target of the virtual function call. Modern compilers place VTables in read-only sections, defeating this kind of attacks by default. The alternate form of attack corrupts the VTable pointer, forcing the program to load the VTable from an alternate location. For this attack vector, the attackers indirectly control the target virtual function calls.

There are two flavors of the latter attack: *VTable injection* attacks and *VTable reuse* attacks, depending on where the overwritten VTable pointers point to. In VTable injection attacks, the adversary injects a surrogate virtual table that is populated with attacker-controlled virtual function pointers. In VTable reuse attacks, the attacker reuses existing VTables out of context (e.g., using a different class' VTable or using an offset to a VTable), or even reuses existing data as VTables.

In practice, VTable injection attacks are the most frequently used VTable hijacking attack vector. Attackers can craft arbitrary VTables containing invalid virtual function pointers

pointing to code gadgets (e.g., ROP gadgets) or dynamically generated code sequences (e.g., JIT spraying). Combined with ROP, VTable injection is very easy to launch and reliable.

In VTable reuse attacks, attackers reuse existing VTables or data that looks like VTables (i.e., an array of function pointers). The attacker redirects a virtual call of a class A through attacker-chosen VTables to any function of any class B or any other existing code in memory as long as there is a pointer pointing to that code. This attack is a form of call-oriented programming (through virtual call gadgets).

VTable reuse attacks can bypass defenses like VTint [27]. As shown in one recent CTF (Capture The Flag) event [36], a defense similar to VTint is deployed on one challenge binary, and many teams have successfully bypassed this defense by launching VTable reuse attacks.

Researchers also proved VTable reuse attacks are realistic. The COOP (Counterfeit Object-oriented Programming) attack proposed by Schuster et.al. [28] introduces a specific type of VTable reuse attack. By stitching several virtual functions, attackers may execute arbitrary code. COOP shows that this attack is (i) practical in real applications (e.g., Firefox), and (ii) Turing complete for realistic conditions.

This attack surface is even larger in applications that have writable code (e.g., dynamic generated code). In these applications, even if there are no legitimate VTables or virtual functions in dynamic code memory, attackers may forge them in the dynamic code memory and launch VTable injection and VTable reuse attacks.

Moreover, VTable pointers are not only used in virtual calls. Attackers can overwrite VTable pointers to hijack the RTTI lookup or virtual base object indexing operations. Unlike existing defenses, VTrust protects these other uses from attacks too.

III. VTrust Design

In this section, we will describe the design of VTrust. We start with an overview of the defense solution, then explain the design of each defense layer.

A. Overview of VTrust

VTrust uses two layers of defenses to protect the integrity of virtual function calls in-depth, as shown in Figure 1 and 2.

a) virtual function type enforcement: VTrust ensures that the actual runtime type of a virtual function call matches the static type declared in the source code. In this way, attackers cannot divert virtual calls to invalid functions or code. For a particular virtual call site, only functions with the correct type can be invoked, and thus VTable reuse attacks (including the COOP attack) are infeasible. It also stops all VTable injection attacks, if attackers cannot forge functions.

b) VTable pointer sanitization: VTrust sanitizes VTable pointers at runtime to enforce VTables' validity. So, even if attackers can tamper with the VTable pointers, they can only make them point to the beginning of existing VTables. It thus defeats VTable injection attacks. It also stops most VTable reuse attacks, since attackers can only reuse existing VTables, not part of them or plain data.

The combination of these two layers of defense can protect applications from all VTable hijacking attacks. For applications without dynamically generated code, code pages will not be writable. This stops the attacker from creating her own VTables and functions with forged signatures. In this configuration, VTrust's first layer of defense, virtual function type enforcement, is sufficient to protect against all VTable hijacking attacks.

B. Virtual Function Type Enforcement

As a first layer of defense, we enforce that the runtime target virtual function matches the type expected in the source code. According to the C++ specification, the derived class will override the parent class' virtual function if and only if it defines a function with the same name, parameter type list (but not the return type), constant and volatile qualifiers, and same reference qualifiers. Moreover, for each particular virtual function call site, the object has a statically declared type (i.e., class), and only virtual functions defined in this class or its sub-classes can be invoked.

VTrust provides a precise protection for virtual calls. It enforces that each virtual call site's static type to matches the invoked virtual function's dynamic type. For the types to match, all of the criteria from the C++ specification's virtual function override requirement must match in addition to the function's class information. Otherwise, the control-flow will be blocked.

More concretely, for a particular virtual function call site, all legitimate runtime target virtual functions meet the following requirements:

- The *function name* must be the same, except for virtual destructor functions and virtual calls that use class member function pointers. The destructor functions always have the same name as the class name (except the leading character $\sim$). The class member function pointer may be bound to virtual functions with different names.
- The *argument type list* must be identical, except for the hidden argument *this* pointer that references the runtime object on which the virtual function works;
- The *qualifiers* must be identical, including the constant, volatile, and reference qualifiers.
- The *class relationship* must be compatible. The runtime virtual function must belong to a derived class of the static class declared on the virtual function call site. For example, for a virtual call site that expects a virtual function from a specific static class `Foo`, only virtual functions belonging to classes derived from `Foo` can be invoked at runtime.

Omitting any requirements here would expose more attack surface. For example, if we only consider the class information, then attackers may launch attacks (e.g., COOP) to make VTable pointers pointing to the middle of legitimate VTables, to invoke any virtual function of any compatible class.

On the other hand, any virtual function that meets these requirements can be legally invoked at a particular virtual call

sites, per the C++ specification. So we cannot further reduce the set of legitimate transfer targets, without breaking the program's functionality. In other words, this layer of defense provides the most fine-grained protection for virtual calls.

This layer also provides a strong protection against VTable hijacking attacks. It prevents the attacker from invoking any virtual function whose type does not match the call site type. Even if attackers can control VTable pointers and make them point to existing VTables or data (i.e., VTable reuse attacks) or even to attacker-controlled VTables (i.e., VTable injection attacks), they cannot invoke arbitrary virtual functions. Instead, they have to either (1) reuse existing virtual functions with correct type, or (2) find someway of forging the virtual function and the type. However, the first bypass is not exploitable, since it is legitimate control flow. The second bypass does not work, if target programs do not have writable code.

In the following, we will discuss some design choices, as well as the advantages and limitations of this solution.

1) Fast Runtime Matching: It would be slow to validate the type (including name, argument type list and so on) byte by byte at runtime, especially for validating whether a class is derived from another class. To facilitate the runtime type check, we encode the virtual function's type information into a word-sized signature. A simple comparison between the expected signature (a constant) and the runtime target function's signature (a memory value) is sufficient to validate the function's type.

More specifically, the signatures are statically computed (i) per virtual function based on the function's prototype and (ii) at the call sites based on the available call-site information. For each virtual function, we will place the signature together with its code. For each virtual function call site, we will instrument a security check, to match the target function's signature with the one expected on this call site. As the signature generation happens at compile time, VTrust can statically ensure that there are no collisions with other functions.

This signature-based type check provides a good runtime performance, better than existing solutions including SafeDispatch [29], FCFI [23] and RockJIT [37]. In general, they have to check whether the runtime VTable or function is in a pre-computed set. As a result, they need to do a slow lookup operation for each virtual call site.

2) Complex Inheritance Support: When generating the signature for a virtual function, we need to decide its owner class. A virtual function definition may be shared between several classes inherited from a same ancestor. But we can only place one signature with the function. So we choose the **top-most primary ancestor** that defines this virtual function's interface as the owner class.

Assuming a virtual call site expects a virtual function `Foo::func()`, and the virtual function `func()` is first defined in class `Bar` among `Foo's` all ancestors, then this virtual call site will use `Bar` as the owner class to compute the expected signature. For the function `Foo::func()`, it also will use `Bar` as the owner class to compute its signature.

3) Modularity Support.: To compute the signatures, we only need the ancestor information for the target class. As a result, we can generate signatures for virtual call sites and virtual functions when compiling one single module. We can simply analyze one module at a time, and extract the virtual function's type and compute signatures based on the type. This process is simple and fast. Moreover, it has natural modularity support, allowing us to compile target applications module by module. When the class inheritance changes or a new module is added, the default incremental compilation model is sufficient to update everything.

On the other hand, solutions like SafeDispatch and FCFI all need the descendant classes information for each class to perform the checks. It thus requires a whole-program analysis, either by compile-time analysis or runtime merging, to get the knowledge of the complete class hierarchy. As a result, our solution has a better compile-time analysis speed and modularity support.

4) Dynamic Loading Support: Our defense supports dynamic loading. Whether or not a new library is loaded, each virtual function's signature and each virtual call site's expected signature will not change. In other words, even if the class inheritance graph may change, our solution remains effective without the need to update any runtime information.

Solutions like FCFI (aka VTV) have to load runtime information to update the class inheritance hierarchy, when loading a new module. However, this process can be done with initializer functions themselves, without modifying the dynamic loader.

5) Limitations: Similar to traditional signature-based CFI solutions, this layer of defense is also vulnerable if attackers can forge code with correct signatures, i.e., the target application has writable code. Our solution VTrust can defeat this kind of attack, by deploying an extra layer of defense: VTable pointer sanitization.

Moreover, the way we determine a virtual function's owner class also leaves some attack surface. For example, suppose (1) class `Grand` is `Parent`'s primary base class, (2) `Parent` is `Child`'s primary base class, and (3) the virtual function `func` is first defined in `Grand`, then for a virtual call site which requires `Child::func`, the virtual functions `Grand::func` and `Parent::func` are also allowed, since they share a same top-most primary base class `Grand`. In practice, this should leave only a small attack surface, as all these virtual functions (e.g., `Grand::func`, `Parent::func` and `Child::func`) by design should perform a similar action (i.e., generating similar output and side-effects for the same input) but only in different ways. FCFI also has a similar attack surface [38].

Furthermore, similar to SafeDispatch and FCFI, our solution also faces a compatibility issue when an external unhardened library is loaded into the process (e.g., by invoking the system call `dlopen`). For unhardened libraries, there are no signatures associated with its virtual functions, and the security checks we added to the current application will fail if the external virtual function is used. We also deploy a fail-safe error handler similar to the one used in FCFI. If an unhardened library is used, we build a whitelist of target virtual functions. When the security check fails, the error handler will go through this whitelist, checking whether the target virtual function is in the list. If not, a security violation alert is thrown.

In summary, this layer of defense defeats all VTable reuse attacks, including the COOP attack. It can also defeat all VTable injection attacks if target applications have no writable code. Essentially, it is a C++-aware fine-grained CFI policy. Comparing with other solutions for C++ programs, it is much faster in compile-time analysis and runtime execution. It also has natural modularity support without added complexity, as well as a good dynamic loading support. So, we strongly recommend deploying this layer of defense in practice.

C. VTable Pointer Sanitization

As discussed in the previous section, for applications supporting dynamic code generation, attackers may bypass the first layer of defense by launching VTable injection attacks and utilizing dynamically generated code to forge virtual functions with correct types (i.e., signatures). As a result, we introduce a second layer of defense, to limit the source of virtual functions. More specifically, we sanitize the VTable pointers to enforce that they point to valid VTables, and thus only existing virtual functions can be invoked on a particular virtual call site.

One straightforward solution is to enforce the integrity of VTable pointers. But unlike VTables themselves, VTable pointers cannot be set to read-only, because these pointers are usually members of objects that are on the writable heap or stack. In order to enforce their integrity, we have to track data-flow at runtime, and prevent all memory write operations from overwriting VTable pointers. Obviously, this solution results in unacceptable performance overhead.

Instead, we enforce the validity rather than the integrity of VTable pointers. Our solution enforces the runtime VTable pointers to be `valid`, even if attackers have tampered with them. Existing defenses all fall into this category. VTint [27] checks whether the target VTable is read-only and T-VIP [39] works in a similar way. They both fail to defeat some attacks (e.g., COOP). SafeDispatch [29] and FCFI [23] check whether the runtime VTable is in a pre-computed legitimate VTable set, introducing high performance overhead.

We propose a novel solution to enforce VTable pointers' validity in this paper, by encoding VTable pointers into VTable indexes when initializing them and decoding VTable indexes when they are accessed (e.g., for virtual calls). In this way, even if attackers can control the VTable pointers in objects, these pointers will first be decoded before being used in virtual calls. As a result, only valid VTables can be used to perform the virtual calls.

A straightforward encoding and decoding solution works in this way: we maintain a whitelist of all legitimate VTable pointers, encode each VTable pointer to an index of this whitelist when assigning VTable pointer to objects, and decode the VTable pointers (i.e., indexes) that are read from the runtime objects to the original pointers by using the whitelist. This solution is simple and can defeat all VTable injection attacks. However, it does not support shared libraries. For example, when a VTable is defined in a shared library, its pointer can hardly be encoded to a unique index, because this library may be used in different applications, i.e., this VTable pointer may be recorded in several whitelists.

Instead, we use a separate whitelist (denoted as local VTMap) for each single library, as well as a global map

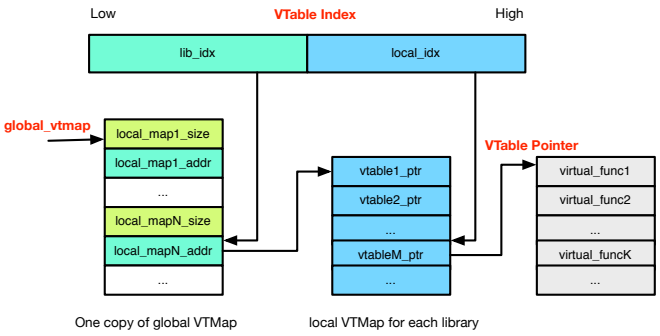

Fig. 3. VTable pointer sanitization solution. Each VTable pointer will be encoded to a VTable index consistinf of a `lib_idx` and a `local_idx`. A global VTable pointer map (pointed by `global_vtmap`) will be used to decode VTable indexes.

(denoted as global VTMap) to perform the VTable pointer encoding and decoding. As shown in Figure 3, each VTable pointer will be encoded into a VTable index (i.e., an integer of the bit width same as the platform), consisting of two sub-indexes of the same bit width: (1) a `lib_idx` that represents the index (to the global VTMap) of the library that uses this VTable; and (2) a `local_idx` that represents the index (to the local VTMap) of the VTable inside the library. When decoding a VTable index, we use its lower half (i.e., `lib_idx`) to retrieve the local VTMap's address from the global VTMap, then use its upper half (i.e., `local_idx`) to retrieve the VTable pointer from the local VTMap.

When compiling a single library, we can build the local VTMap for it, and assign an index `local_idx` to each VTable that is **used** in current library. These indexes are all statically assigned, so that the library can be shared among different applications without modifications. Then we can statically compute the library's index `lib_idx`, either by (1) manually specifying or (2) automatically scanning existing libraries indexes and computing a different one.

The global VTMap of each application must be initialized at runtime. Whenever a library is loaded, its local VTMap's address will be registered in this global VTMap, at the specific entry numbered with `lib_idx`. We also store the size of the local VTMap into the global VTMap, for further bound checks when accessing the local VTMaps. As a result, this solution provides a good support for incremental compilation and dynamic loading. Similar to virtual function type enforcement, it may cause incompatibility issues when working with unhardened libraries that have VTables.

This layer essentially provides a whiltelist protection. Although attackers may overwrite the VTable pointers, the pointers will be decoded (and therefore validity checked) before being used in virtual calls. As a result, this defense defeats VTable injection attacks. We emphasize that, this layer of defense also provides partial protection against VTable reuse attacks, even if the first layer of defense is absent. It enforces that only legitimate VTables can be used for virtual calls. So attackers can only reuse existing VTables, rather than any other data or the middle of existing VTables on which the COOP attack is based. However, in theory, attackers may still launch some VTable reuse attacks by only reusing existing VTables. Therefore, both defense layers are needed to prevent attacks.

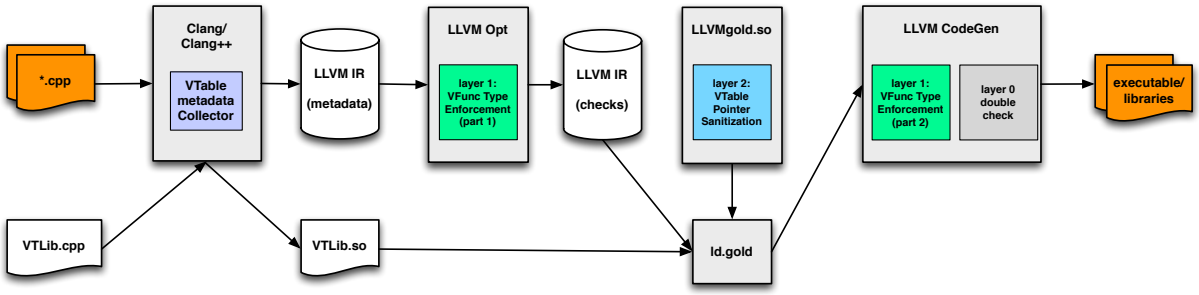

Fig. 4. Illustration of VTrust's workflow. The first layer of defense is implemented as an compile-time optimization pass and a code generation step. The second layer of defense is implemented as a link-time optimization pass.

a) Limitations: Our solution for selecting `lib_idx` can avoid conflicts when all code is compiled on the same machine. If code is compiled on different machines, it is not possible to statically determine a unique `lib_idx` by manual specification or by scanning for other libraries. This is an engineering challenge we plan to address in future work.

One possible solution is to have the library loader resolve the conflicts like it does for relocation. In fact, conflicts should not be that common in practice, as the number of possible values of `lib_idx` is much larger than the number of libraries a typical application links against. Another solution is to use the library name rather `lib_idx` to index the local VTMap, which eliminates the conflicts but makes the runtime decoding a little slower.

b) Alternative VTable Pointer Sanitization Solutions: We also tested a range check solution to validate VTable pointers. It records all legitimate VTable sections' address ranges, updates this information when libraries are loaded and unloaded, and validates whether the runtime VTable falls in any of these address ranges. However, this alternative is not precise as the VTable encoding and decoding solution we discussed earlier, and also has a higher performance overhead.

IV. Implementation

We have implemented a prototype of VTrust using the LLVM [40] compiler suite version 3.4 for x64. Figure 4 shows the workflow of VTrust. In general, there are four steps in our implementation.

First, we collect VTable related information of the current compilation unit in the compiler frontend (i.e., Clang++), including all virtual functions, all VTable constants, all virtual call instructions, as well as all VTable assignment and read operations. It is worth noting that we do not need to collect the class inheritance information during the compilation, unlike SafeDispatch [29] and FCFI [23].

This information is kept in the form of LLVM metadata and function attributes, and passed to the optimizer and linker. Some optimizations may remove or replace some LLVM instructions, e.g., the instruction combination optimization. When instructions are removed, some LLVM metadata may be discarded, which causes problems in our link-time analysis. In our prototype, we keep multiple copies of metadata in different instructions when compiling, as well as in the compilation module. During the link-time analysis, we will perform a cross verification to detect such metadata missing issues, and recover the metadata automatically based on other copies.

Second, we instrument type signature checks before virtual call instructions (i.e., the first layer of defense) when performing compile-time optimization.

Third, we utilize LLVM's link-time optimization support (based on the linker's gold plugin feature) to deploy our second layer of defense. More specifically, we encode and decode the VTable pointers before use.

Finally, we instrument type signatures before the body of each virtual function (for the first layer of defense) during code generation. We also verify that all VTables are placed in read-only sections (i.e., layer 0 defense).

We also provide a runtime library `VTLib.so` for some runtime APIs used by the security checks that we instrumented. All the hardened modules are then linked together by the gold linker, generating the final executable or libraries.

A. Virtual Function Type Enforcement

In the first layer of defense, VTrust will enforce that each runtime virtual function has a matching type (i.e., the name, argument type list, qualifiers, and class information) with the one expected on a virtual call site. To make the runtime check more efficient, we encode the type information into a word-sized hash value. This hash value is used as the signature for each virtual function, and is compared before the virtual function call site. If they do not match, a security violation is detected and the program is terminated.

For each virtual function, we collect its type information from the frontend, compute its signature, and embed this signature right before the function body when generating the native code for this function.

For each virtual function call site, we collect the expected type and compute the expected signature. VTrust will then instrument a security check before this virtual call to match the function's runtime signature against the expected one.

Computing the signature based on a type information is easy, but collecting the type information is not. The qualifiers and argument list type are deterministic. However, the virtual function name and the class information are not.

- First, we cannot get a meaningful function name for virtual destructor functions. The derived class' virtual

```
C++ source code

typedef void (Base::*base_fptr)(void);

void test_foo(Base* obj, base_fptr fptr){
  (obj->*fptr)();
}

void main(){
  Base* obj1 = new Base();
  base_fptr fptr = &Base::foo1;
  test_foo(obj1, fptr);    // virtual call #1

  Base* obj2 = new Sub();
  fptr = &Base::foo2;
  test_foo(obj2, fptr);   // virtual call #2

  obj2->foo2();            // virtual call #3
}
```

(a)

```
executable code (virtual function body)

   dd  signature_without_name
   dd  signature1
Base::foo1:
   ...
   ret


   dd  signature_without_name
   dd  signature2
Base::foo2:
   ...
   ret
```

(b)

```
executable code (virtual call site)

test_foo:
   ...
   ; assume EAX is the target virtual function
   cmp [eax-8], signature_without_name
   jnz ERROR
   call eax            ; virtual call #1 and #2
   ...
   ret


main:
   ...
   ; assume EAX is the target virtual function
   cmp [eax-4], signature2
   jnz ERROR
   call eax            ; virtual call #3
   ....
   ret
```

(c)

Fig. 5. Illustration of the class member function pointer issue and solution. Here, virtual functions `Base::foo1` and `Base::foo2` have a same function type but different name. We instrument an extra signature that is computed without function name before the function body. For virtual call sites that use class member function pointers, we compare this special signature instead of the signature with function names.

destructor function will overload the parent class' virtual destructor function, i.e., they use the same slot in the VTables. But they have different function names. For example, assuming `Foo` is derived from `Bar`, then their virtual functions use a same slot in the VTable, but with different names: `~Foo` and `~Bar`.

- Second, we cannot use the virtual function's owner class' information (e.g., name) to compute the signature. For a virtual function call site, we only know a static class. At runtime, the target virtual function may belong to another class. These two classes are often different and their names do not match.
- Third, we cannot get a meaningful function name for virtual call site that uses class member function pointers. As shown in Figure 5, the virtual call in function `test_foo` does not have a meaningful name.

For the first two issues, similar to choosing owner class for virtual functions as discussed in section III-B2, we choose the destructor name of the top-most primary ancestor class among all its ancestors that defined virtual destructors.

Algorithm 1. VTable Building and Type Collection Algorithm (Python-style pseudocode). Text in orange is the code we instrumented to collect type information for virtual functions.

```
addVTableMethods(TgtClass):
  BaseClass = PrimaryBase(TgtClass)
  # recursive invocation
  addVTableMethods(BaseClass)
  # all overrider virtual functions
  for overrider in TgtClass:
    updateVTableEntry(overrider)
    oldFunc = get_overriden_func(overrider)
    if isVirtualDestructor(overrider):
      ancestorFuncName = get_func_name(oldFunc)
      register_func_name(overrider, ancestorFuncName)
    ancestorClassName = get_class_name(oldFunc)
    register_class_name(overrider, ancestorClassName)
  # all new virtual functions
  for newFunc in TgtClass:
    appendVTableEntry(newFunc)
    if isVirtualDestructor(newFunc):
      register_func_name(newFunc, '~'+TgtClass)
    register_class_name(newFunc, TgtClass)
```

For each virtual function in a class, there is one slot in the per-class VTable. If a derived class overrides a virtual function, then the overrider takes the slot at the same offset in the VTable of the derived class. The text in black in Algorithm 1 shows the basic VTable building algorithm used in LLVM. In general, the primary base class initializes the VTable first, and the derived class then updates the VTable slots with overrider functions and extends it with new functions.

Based on this basic algorithm, for each virtual function, we can easily retrieve the function name and class name of the top-most primary class that defines this function. As shown in Algorithm 1, for each overrider function, we use the overridden function's name and class information. It is worth noting that, we use the information from the top-most primary class that first defines the virtual function, not from the top-most primary class of the static class (declared in the virtual call site). These two may be different since an object may contain several VTables, due to multiple inheritance.

In this way, we can compute the signature of the type without the complete class inheritance tree. We can collect all the information when the compiler builds the VTable. It makes our compile-time analysis very fast. Moreover, if the programmer extends the class inheritance tree in the future, we do not need to recompile existing modules.

For the third issue, we instrument special security checks for virtual call sites that use class member functions. It reads the signature from a different offset to the function body, which is computed without the function name information. This signature still has the class information, as well as the function prototype information, and thus is strong enough.

B. VTable Pointer Sanitization

In the second layer of defense, VTrust will sanitize VTable pointers at runtime, to enforce they are *valid*. More specifically, we will encode VTable pointers when they are assigned to objects, and decode them when they are used.

1) VTable Pointer Encoding: When analyzing a module (i.e., a library or executable since we are working on link-time optimization), we discover all VTable pointer assignment

operations, including (1) assigning VTable pointers to runtime created objects in constructor functions; (2) filling VTable construction tables (VTT, an auxiliary data structure for complex class inheritance) with VTable pointers; (3) assigning VTable pointers to static `typeinfo` objects that are used for RTTI; and (4) assigning VTable pointers to some constant static objects. It is worth noting that, for the latter three cases, there are no assignment instructions. Instead, the compiler will directly put the VTable pointer at the proper location.

For each of these VTable pointer assignments, we will replace the pointer with a (statically computed) constant index. As discussed in Section III-C, this index consists of two parts: an index `local_idx` to the local library's VTMap, and the index `lib_idx` to the global VTMap. We use the order of each VTable pointer in the library as its `local_idx`, and use the build order of each library as its `lib_idx` (after scanning existing libraries and fixing conflicts). As a result, we can compute the constant index for each VTable pointer at link-time. After encoding, the VTable pointer of each runtime object will be a constant integer.

We create a global VTMap array, which will at runtime hold all loaded libraries' local VTMaps' addresses, in the support library `VTLib.so` for each application. For each library, we add an initialization function and a local VTMap array. The local VTMap stores addresses of all VTables used in the current library. The initializer function is invoked automatically when the library is loaded, and registers the local VTMap to the global VTMap. More specifically, it updates the global VTMap's `lib_idx`-th entry to store the size of the local VTMap and its runtime address. This runtime update operation temporarily maps the global VTMap as writable when loading a library. Most applications only load libraries during initialization, therefore the risk of being attacked during this short time window is low.

2) VTable Pointer Decoding: After encoding the VTable pointers, we have to decode them at runtime, when the object's VTable pointer is read out and used for (1) accessing virtual function pointers for virtual calls; (2) accessing offsets of virtual base objects; or (3) accessing RTTI information.

To decode VTable pointers, we first parse the VTable index that is read from the runtime object into two parts: `local_idx` and `lib_idx`. Then we use the library index `lib_idx` to access the global VTMap, and get the library's local VTMap's address and size.

If the `lib_idx` is larger than the global VTMap's size, or the `local_idx` is larger than local VTMap's size, we raise a security violation exception and terminate the program. Otherwise, the `local_idx` is used to access the local VTMap, to get the value of the VTable pointer. Finally, this decoded pointer is used for virtual calls. In this way, even if attackers can overwrite the runtime VTable pointers, e.g., by exploiting use-after-free vulnerabilities, only VTables in the local VTMaps can be used in virtual calls after decoding.

This solution changes the representation of VTable pointers and will cause incompatibility issues if some libraries are not instrumented. But it covers all VTable uses and protects them from attacks. For example, if there are custom virtual calls written in assembly, as shown in the evaluation, traditional defenses may fail to protect them.

TABLE I. VIRTUAL CALL RELATED STATISTICS FOR SPEC CPU2006 BENCHMARKS WRITTEN IN C++. THE UNIT M STANDS FOR MILLIONS, AND B STANDS FOR BILLIONS.

	Runtime Profiling				Static Count	
	#inst	iCall	iJump	Return	vtbl.	vcall
444.namd	39M	1.17%	37.74%	61.08%	3	2
447.dealII	43B	1.01%	27.33%	71.66%	115	200
450.soplex	144M	3.78%	41.03%	55.19%	51	495
453.povray	29B	24.30%	2.43%	73.27%	48	112
471.omnetpp	22B	11.19%	18.63%	70.18%	127	1431
473.astar	15B	32.95%	0.07%	66.98%	2	0
483.xalanc.	36B	24.42%	7.81%	67.78%	29	4284

V. EVALUATION

We evaluate our prototype implementation on the SPEC CPU2006 benchmarks and the open source browser Firefox, to test our prototype's runtime performance and security.

A. Virtual Call Statistics

First, we measure the attack surface of VTable hijacking attacks by gathering virtual call related metrics for these benchmarks, Our results show that a large attack surface exists in real applications. Consequently, defenses like our solution VTrust should be deployed as soon as possible.

1) Statistics for SPEC2006: Table I shows the total number of runtime indirect control flow transfers for SPEC CPU2006 benchmarks that include C++ code, and the static count of VTables and virtual calls. At runtime, there are 1% to 33% of instructions are indirect calls. So, it is important to deploy defenses like VTrust to harden these applications.

We also found that, the proportion of return instructions is high for all benchmarks, calling for an efficient stack protection mechanism (which is orthogonal to our work).

Moreover, we found that many indirect control flow transfers at runtime only have a single target. In our runtime profiling, an average of 26% indirect calls, 85% of indirect jumps, and 44% of return instructions only have one runtime target. Such a high number of control transfers with a single target indicates that a localized caching strategy, e.g., devirtualization and inline caching [41], might be an interesting future opportunity for optimization. This optimization not only improves the runtime performance, but also reduces some attack surfaces. FCFI [23] showed the devirtualization can greatly improve the runtime performance.

2) Statistics on Firefox: Table II shows parts of the VTable-related information in the browser Firefox. We collect these information during compile-time optimization, and thus only libraries or executables are evaluated. The data of all object files are not included here. There are 39 libraries and executables in Firefox, and only 12 of them are listed in this table.

Columns 2-4 shows the VTable related information. The second column shows the count of VTables in each library (or executable). The library libxul.so has 15,801 VTables, indicating that there are thousands of classes in Firefox.

The third column shows the count of VTable pointer assignments. VTable pointers are usually assigned to runtime created objects in the constructor functions. In each constructor

TABLE II. VTABLE-RELATED STATISTICS OF FIREFOX, INCLUDING THE COUNT OF (1) VTABLES, (2) VTABLE POINTER ASSIGNMENTS, (3) VTABLE POINTERS READ OPERATIONS, AND (4) CALL INSTRUCTIONS, AS WELL AS THE RATIO OF INDIRECT CALLS TO CALL INSTRUCTIONS, AND THE RATIO OF VIRTUAL CALLS TO INDIRECT CALLS.

library/	VTable			Call Instruction		
executable	#	assign	read	call	iCall	vCall
makeconv	173	470	1831	62625	6.66%	42.85%
genrb	173	473	1831	68429	6.35%	41.10%
icuinfo	173	470	1844	66600	6.35%	42.40%
genccode	173	470	1831	61037	6.81%	42.95%
gencmn	173	470	1831	61051	6.81%	42.95%
icupkg	175	476	1845	63197	6.89%	41.36%
pkgdata	175	476	1845	64363	6.75%	41.43%
gentest	174	471	1846	66640	6.35%	42.42%
gennorm2	179	478	1837	61831	6.78%	42.73%
gendict	174	472	1831	60896	6.83%	42.94%
js	1420	1991	3626	262502	23.26%	5.87%
libxul.so	15801	26212	72874	1720021	15.21%	27.48%

function, it may assign multiple VTable pointers to the objects due to multiple inheritance. The library libxul.so has 26,212 VTable pointer assignment operations, indicating that Firefox has thousands of constructors. For each VTable pointer assignment, the optional second layer of VTrust statically encodes the pointer to a constant index before it is assigned to the object.

The fourth column shows the count of VTable pointer read operations. Before accessing VTables (e.g., for virtual calls or RTTI), the VTable pointers will be read from the objects first. VTrust will decode these pointers at runtime. The library libxul.so has 72,874 VTable pointer read operations.

Most of these VTable pointer read operations are used for virtual calls. As shown in this table, there are 71,892 virtual calls (=1,720,021*15.21%*27.48%), close to the count of VTable pointer read operations. The remaining three columns in the table show the count of call instructions, the ratio of indirect calls to call instructions, and the ratio of virtual calls to indirect calls. It shows that about 40% of indirect calls are virtual calls, which is very high.

From this table, we can see that there is a large attack surface in Firefox. Attackers can find a lot of useful VTables and virtual call sites to launch VTable hijacking attacks. As the paper [28] discussed, it is practical to launch COOP attack in Firefox.

B. Performance Overhead

We have evaluated the performance overhead of VTrust on the SPEC CPU2006 benchmarks and the Firefox web browser.

1) Runtime Performance on SPEC2006: To evaluate the performance overhead of VTrust we applied it to the SPEC CPU2006 benchmarks that are written in C++. There are seven C++ benchmark programs in the SPEC CPU2006 suite. We ran the benchmarks under Ubuntu 14.04 LTS on a computer with an Intel Core i7-3770 with eight cores @ 3.40 GHz Processor and 16 GB RAM.

Our current prototype implementation of VTable pointer sanitization only works with the C++11 compatible library libcxx provided by LLVM, which has some compatibility issues with the 471.omnetpp and 447.dealII benchmarks. All

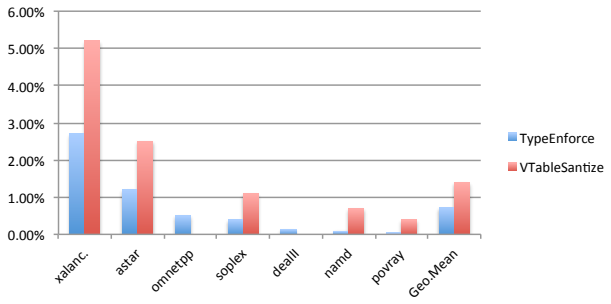

Fig. 6. Performance overhead of VTrust on SPEC CPU2006, when enabling only the first layer of defense (virtual function type enforcement), or only the second layer (VTable pointer sanitization).

other benchmarks and configurations work well. The overhead for virtual function type enforcement is very low, on average 0.72%. The worst case's overhead is about 2.7%. The average overhead for VTable pointer sanitization is 1.4%, and the worst is 5.2%. The average overhead of these two layers together is 2.2%, and the worst case is 8.0%.

2) Performance on Firefox: We also measure the performance overhead on a Firefox (version 34.0). We use six popular browser benchmarks, including Microsoft's LiteBrite [42], Google's Octane [43], Mozilla's Kraken [44], Apple's Sunspider [45], RightWare's BrowserMark [46] and PeaceKeeper [47], to test browsers' performance on JavaScript execution, HTML rendering and HTML5 support.

We tested the browser using Ubuntu 14.04 on a computer with an Intel Core i7 with 12 cores @ 3.7 GHz Processor and 16GB RAM. We tested the performance overhead of VTrust's first and second layer separately.

As shown in Figure 7, the first layer of defense (i.e., virtual function type enforcement) introduces negligible performance overhead. On average, the performance overhead is about 0.31%. In the worst case, its performance overhead is about 1.05%. The overhead is so small that in some cases it is even negative. We attribute these fluctuations to caching effects, code layout, or system noise.

The second layer of defense (i.e., VTable pointer sanitization) introduces higher performance overheads. It has a performance overhead of about 1.81% on average. In the worst case, it introduces a performance overhead of 3.21%.

For applications with dynamically generated code (e.g., Firefox), we enforce both layers of VTrust. The performance overhead is close to the accumulation of the overheads of the first layer and the second layer. For example, if we enable both of these two defenses, it introduces an average performance overhead of 2.2%, close to the sum of the standalone virtual function type enforcement's overhead 0.31% and the overhead of VTable pointer sanitization 1.81%.

a) Practical Experience with Firefox: Firefox uses some tricks to support multiple platforms, and interactions between JavaScript and C++ code. It implements several special virtual functions `nsXPTCStubBase::StubNN`, where `NN` is a number ranging from 0 to 249, by using inline assembly code with mangled names (e.g.,

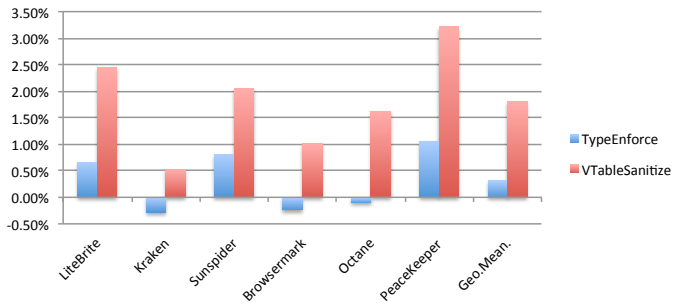

Fig. 7. Performance overhead of VTrust on Firefox, when enabling only the first layer of defense (virtual function type enforcement), or only the second layer (VTable pointer sanitization).

`__ZN14nsXPTCStubBase6StubNNEv`). The linker automatically resolves these virtual functions by name at runtime. So, the frontend compiler (e.g., Clang/Clang++) does not know the existence of these functions. As a result, VTrust fails to include signatures before these functions, causing false alarms when they are used at virtual call sites that have been instrumented with the first layer of defense. As a workaround, we modify the security violation handler to check whether the target is in the set of special virtual functions (for this application), when a security violation is detected.

Moreover, Firefox also has a special virtual call site, which is simulated in a special function `NS_InvokeByIndex`. This function will get an object and a method index as arguments, then do a simulated virtual function call: (1) it reads the VTable pointer from the argument object; (2) it reads the function pointer from the VTable using the method index; and (3) it calls the target method. This in fact is a virtual function call site, but the compiler frontend is not aware of it. As a result, VTrust fails to instruments checks, including the second layer of defense, for this call site. At runtime, encoded VTable pointers will be used here, and will then cause compatibility issues. We can identify this kind of corner cases, and instrument them with VTrust. It is worth noting that, all other existing defenses, including FCFI and SafeDispatch, fail to identify this kind of corner cases, leaving them still vulnerable to VTable hijacking attacks.

C. Performance Comparison

Among all existing solutions that provide a strong protection against VTable hijacking attacks, FCFI [23] has the lowest performance overhead. Other solutions, e.g., SafeDispatch, vfGuard and RockJIT, introduce a much higher performance overhead, which will be discussed in Table IV.

For the worst case SPEC benchmarks `astar` and `xalanc`, FCFI introduces a performance overhead of 2.4% and 19.2% respectively, while VTrust introduces a comparable performance overhead of 3.7% and 7.9% respectively when enabling both two layers of defenses. An important point is that while the FCFI paper reports a "lower bound" of 4.7% overhead on `xalanc`, this is not a valid comparison to VTrust. FCFI's "lower bound" configuration uses profile guided optimization (PGO), devirtualization, and replaces the bodies of the FCFI library functions with stubs. We do not consider PGO practical for complex software or a fair technique for benchmarks with a small number of input datasets, like the SPEC CPU2006 benchmarks. In fact, the FCFI authors admit Chrome cannot be built with with PGO and devirtualization. For the browser benchmark `sunspider` and `octane`, FCFI introduces an overhead of 1.6% and 2.6% respectively when deployed on Chrome, while VTrust introduces an overhead of 2.8% and 1.5% respectively. So, VTrust introduces a similar performance overhead as FCFI.

FCFI validates the runtime VTable against a legitimate set that is updated when loading libraries are loaded. This validation needs to (1) dynamically update the legitimate set for each virtual call when a library is loaded, (2) resolve the *split-set problem* when a VTable set is created, and (3) perform a slow set lookup operation to validate the runtime VTable. Since the library loading usually finished before benchmark testing, and thus its overhead is not easy to evaluate. Moreover, when the legitimate set's size is large, which is the common case for classes with many derived classes, it will take a longer time to do the runtime lookup.

Our solution VTrust has a negligible overhead of library loading. It only validates the signatures of target functions, and decodes the VTable pointers before they are used. It costs a constant time for each virtual call, and is faster than FCFI in general. More important, for applications without dynamic code, VTrust only validates the signatures of target functions, which is much faster. So, in general, VTrust is faster than existing solutions.

D. Memory Overhead

We evaluated the memory overhead on Firefox in two scenarios: after a cold start and after running a sample benchmark.

After a cold start, the original Firefox uses about 130MB memory (resident set size, RSS). The hardened version Firefox uses about 133MB memory. The absolute memory overhead is about 4MB, and the relative memory overhead is about 3.1%.

Most of the memory overheads are from (1) the instrumented security checks including the type enforcement checks and VTable pointer decoding instructions, (2) the instrumented local VTMaps and the global VTMap that are used for VTable pointer encoding and decoding, (3) the instrumented signatures before each virtual function's body, and (4) the runtime supporting library VTLib.so, introduced by VTrust.

For example, there are 71892 virtual calls in the library libxul.so, and each virtual call costs about 40 bytes for the security checks. As a result, the security checks in this library takes about 2.9MB. Moreover, there are 15801 VTables in this library, taking about 128KB memory.

After running Firefox for a while, e.g., after testing the Kraken benchmark, the original Firefox uses about 299MB memory. The hardened Firefox uses about 303MB memory. The absolute memory overhead is still 4MB, close to the memory overhead in the cold start scenery. The relative memory overhead drops to 1.3%. Our solution VTrust does not use runtime allocated memory, so its absolute memory overhead stays constant.

TABLE III. PUBLIC VTABLE HIJACKING EXPLOITS AGAINST FIREFOX.

CVE-ID	Exploit Type	Vul App	Protected
CVE-2013-1690	VTable injection	FF 21	YES
CVE-2013-0753	VTable injection	FF 17	YES
CVE-2011-0065	VTable injection	FF 3	YES

E. Case Studies

1) Real World VTable Injection Attacks: To evaluate the effectiveness of VTrust, we choose three public real world vtable hijacking exploits. These exploits are publicly available and all target the popular browser FireFox by exploiting use-after-free vulnerabilities. They all inject fake vtables and hijack the control flow. This is the most common VTable hijacking attack seen in practice. Table III shows details for these exploits, including the CVE-ID, target Firefox version, and the type of the exploits.

Experiments are carried out in a virtual machine running Ubuntu 14.04. For each exploit, we download the vulnerable Firefox's source code and compile it with VTrust. After hardening Firefox, we drive the browsers to access malicious URLs containing exploits. Results show that all the exploits we collected are blocked. Therefore, VTrust successfully protects applications from VTable injection attacks.

2) Real World VTable Reuse Attacks: Since it is much easier to launch VTable injection attacks than VTable reuse attacks and no defenses against these attacks have been deployed, there are few VTable reuse attacks in real world. We found only one such case, besides the COOP attack published recently. In a recent Capture The Flag event [36], there is one challenge program (i.e., `zhongguancun`) that deploys a similar defense as VTint [27]. It checks if the runtime VTable is writable. If yes, it terminates the program. The only way to hijack its control flow is through VTable reuse attacks.

More specifically, this challenge program allocates a large buffer on the heap, and several objects close to this buffer. When passing a negative number to the program, it will overflow the buffer on the heap. By exploiting this vulnerability, attackers are able to overwrite the adjacent objects that have VTable pointers. They then overwrite this VTable pointer with a pointer to read-only memory, to bypass the deployed defense.

In fact, attackers can overwrite this VTable pointer to reference an offset in an existing VTable (i.e., a COOP attack). In this way, attackers can invoke a virtual function out of context. On the other hand, the program contains a virtual function that writes arbitrary content to the memory pointed by the function argument, allowing attackers to implement write-what-where primitives. Finally, attackers can overwrite control data, e.g., function pointers in the Global Offset Table, to hijack the control flow.

Several teams have solved this challenge, showing that VTable reuse attacks are feasible. The research paper COOP [28] also shows that VTable reuse attacks are practical in larger applications, e.g., Internet Explorer and Firefox.

To evaluate our solution's effectiveness against VTable reuse attacks, we collected several public exploits for this CTF challenge. Then we get the source code of the challenge from the author, and recompile the challenge using our tool VTrust. Finally we modify all these exploits to fit our new environment, and test them against the hardened challenge program. The result shows that all these exploits are blocked when the overwritten VTable pointer is used for virtual calls.

VI. RELATED WORK

A. Control-Flow Hijacking Defense

Control-flow hijacking attacks (including VTable hijacking attacks) usually utilize memory safety bugs to modify the program state by tampering with code pointers (e.g., return addresses and function pointers), causing the control-flow to divert when the broken code pointers are used.

Effective defense mechanisms against these attacks can be classified on *when* they stop an attack [50]: (i) at the memory safety level by, e.g., checking bounds of memory access [8], [51]–[53] or enforcing temporal safety on memory [9], (ii) when code-pointers are written (i.e., protecting a subset of data) [10], or (iii) when corrupted data is used in a computation like for Control-Flow Integrity (CFI) [11]–[23].

Memory safety-based defenses result in fairly high overhead as many memory read and write operations must be protected with additional guards. Some tools reduce the overhead by restricting protection to write operations only [54]. Code-Pointer Integrity [10] restricts the protection in another dimension by protecting a subset of all pointers: code pointers and any data structure that references code pointers.

CFI protects against control-flow hijacking attacks by adding guards before indirect control-flow transfers, which restricts each indirect control-flow transfer to the set of valid targets as determined by a static analysis (usually a type-based points-to analysis). CFI can stop attacks such as return-to-libc [55] and ROP [2]. HyperSafe [13] enforces a fine-grained CFI policy for virtual machine managers. Recent approaches [17], [19] directly rewrite binaries and provide a coarse-grained CFI protection.

However, CFI faced several adoption hurdles: the fine-grained CFI solutions usually do not support separate compilation, few of them provide precision protection for C++ programs, and many of them induce fairly high overhead. Relaxed implementations can be circumvented [56]–[58]. Our solution VTrust provides a fine-grained CFI for only virtual calls. It does not rely on a whole-program analysis and provides modularity support.

B. VTable hijacking Defense

Researchers also proposed some specific virtual call protection solutions. Table IV shows a brief comparison between these solutions and our solution VTrust, including the effectiveness of each defense, the support of incremental building (i.e., modularity support), dynamic loading of external libraries (i.e., mixed code) and dynamic generated code (i.e., writable code), as well as the performance of compile-time class hierarchy analysis and runtime overhead.

The VTint [27], T-VIP [39] solutions are binary-rewriting based defense mechanisms. VTint places VTables in a special read-only section and adds instrumentation before virtual calls to check if the runtime target VTable is in this read-only

TABLE IV. COMPARISON BETWEEN DEFENSES AGAINST VTABLE HIJACKING ATTACKS, INCLUDING WHETHER THEY CAN (1) DEFEAT VTABLE HIJACKING, AND SUPPORT (2) INCREMENTAL BUILDING (I.E., MODULARITY), (3) EXTERNAL LIBRARIES, AND (4) WRITABLE CODE (I.E., DYNAMIC GENERATED CODE). THIS TABLE ALSO SHOWS THE COMPARISON OF (5) SPEED OF CLASS HIERARCHY ANALYSIS, (6) SOURCE CODE DEPENDENCY, AND (7) PERFORMANCE OVERHEAD. THE ABBREVIATION `SD` STANDS FOR SAFEDISPATCH, `FCFI` STANDS FOR FORWARD EDGE CONTROL-FLOW INTEGRITY. IN THE `dynamic loading` COLUMN, `Y/N` MEANS THE DEFENSE SUPPORTS LOADING HARDENED OR ANALYZED LIBRARIES, BUT NOT UNHARDENED ONES.

defense solution	able to defend?		incremental building	external libraries	writable code	class hierarchy analysis speed	source code dependency	performance overhead
	VTable injection	VTable reuse						
VTint [27]	y	partial	N/A	y	y	N/A	N	2%
T-VIP [39]	y	N	N/A	y	y	N/A	N	2.2%
vfGuard [48]	y	partial	N/A	Y/N	y	N/A	N	18.3%
original CFI [11]	partial	partial	N/A	Y/N	N	N/A	N	16%
VTGuard [49]	N	y	y	Y/N	N	N/A	y	$< 0.5\%$
SD-vtable [29]	y	y	N	Y/N	y	slow	y	30%
SD-method [29]	y	y	N	Y/N	y	slow	y	7%
RockJIT [37]	y	y	y	Y/N	y	slow	y	10.8%
FCFI-VTV [23]	y	y	y	Y/N	y	fair	y	about 3%
VTrust	y	y	y	Y/N	y	fast	y	0.72% or 2.2%

section. It can defeat all VTable injection attacks, but only a few VTable reuse attacks (i.e., reusing existing data rather than VTables). Attackers may reuse existing VTables to launch attacks [28] to bypass it. T-VIP works in a similar way, but does not provide any protection against VTable reuse attacks. They both introduce a low performance overhead, and are able to protect applications with writable code.

VfGuard [48] is another binary level defense. It filters virtual functions at runtime based on some features, e.g., the index of the function inside a VTable. It uses dynamic instrumentation tool PIN [59] to validate these filters, and thus has a high performance overhead. Since the filters used by vfGuard are permissive, it only provides a partial protection against VTable reuse attacks. Moreover, all these three binary solutions rely on some heuristics to identify VTable related operations in programs, and may also cause false negatives in some cases, i.e., some virtual calls are not protected.

The original CFI [11] also provides some protection against VTable hijacking. However, it cannot defeat all VTable hijacking attacks, because it does not utilize the type information of virtual functions. Moreover, it does not support writable code and incurs a higher performance overhead.

VTGuard [49] is a lightweight source code level defense, similar to stack canaries, that instruments secret cookies at the end of legitimate VTables. Its performance overhead is extremely low. However, it is vulnerable to information leakage attacks. Attackers may leak the secret cookies and inject fake VTables with correct cookies. So, it cannot defeat VTable injection attacks, nor protect applications with writable code.

SafeDispatch [29], RockJIT [37] and FCFI [23] work on programs' source code too. SafeDispatch resolves the set of legitimate VTables (or virtual functions) for each virtual function call by performing a class hierarchy analysis (CHA) at compile-time, and validates the runtime VTable (or virtual function) against this set. It requires a heavy compile-time class hierarchy analysis, which prevents the incremental compilation. It uses a set lookup operation that is slow to perform the security check, introducing a high performance overhead.

RockJIT is based on a fine-grained signature-based CFI solution MCFI [22] that only protects C code, and extends it to Just-in-Time compiled code and virtual calls. RockJIT also performs a CHA analysis like SafeDispatch, and introduces a very high performance overhead as well. Unlike SafeDispatch, it supports separate compilation by emitting the class hierarchy information into each module and combining them at link time. However, it also has to rebuild the whole program, when the class hierarchy changes. VTrust only uses signature matching (i.e., type enforcement) to protect virtual calls, without the requirement of class hierarchy information, which provides a better compatibility and performance.

FCFI proposes several methods to protect indirect call and jump instructions. Its method VTV also validates if the target VTable is in a legitimate set. But it only analyzes parts of the class hierarchy information when compiling, and utilizes the runtime initializer functions to update the overall class hierarchy. In this way, it supports incremental building with a faster class hierarchy analysis.

However, it also needs a slow runtime set lookup operation to perform the security check. The performance overhead not only depends on the count of virtual calls, but also the size of the legitimate VTabe set. In applications with complex class hierarchy, the performance overhead would be higher. Moreover, it needs to perform an extra check each time a new VTable set is created, to overcome the *split-set problem* [23], causing a high overhead when loading a new library.

FCFI's overhead ranges from 2.4% to 19.2% for SPEC applications, and from 1.6% to 8.4% for the Chrome browser when testing different benchmarks. Our solution VTrust introduces an average overhead of 0.72% and 0.31% for SPEC and Firefox respectively, when only enabling the first layer. Even with the extra second layer of defense, the average overhead of VTrust is about 2.2%, comparable with FCFI. FCFI also introduces a profile guided optimization to perform devirtualization, i.e., translating virtual calls to direct calls. This optimization helps improve the overall performance a lot. It can also be adopted by other defenses, such as VTrust.

Our solution VTrust uses the signature matching to enforce virtual functions' type and provide a most fine-grained

CFI protection, and an optional extra layer of defense to validate VTable pointers' validity in case target applications have writable code allowing attackers to forge functions with correct signatures. It does not need any global class hierarchy information, and thus it has a faster static analysis and natural modularity support. Its performance overhead is also better than most of existing solutions. Moreover, the overhead of each security check instrumented by VTrust is constant, irrelevant to the class hierarchy. It is also able to identify corner cases, and provides a complete protection against VTable hijacking attacks.

All six of these source code level defenses support dynamically loaded libraries. When a hardened library is dynamically loaded into the process, the runtime class hierarchy is updated, so the newly loaded virtual functions are allowed to be called. However, if an unhardened library is loaded into the process, all these solutions may cause false positives because the class hierarchy is missing the classes from the loaded library. Usually, a special fallback failure function that tracks a whitelist can be embedded in the security check to catch such false positives.

VII. Conclusion

VTable hijacking attacks are one of the most critical threats to modern applications written in C++. Few solutions have been deployed in practice. We propose a lightweight compiler-based two-layer defense VTrust, to defeats all VTable hijacking attacks. The first layer enforces that the runtime target virtual function matches the type expected in the source code. The second layer enforces that each VTable pointer references a valid VTable. Modern compilers usually place VTables in read-only sections, forming a default layer of defense (i.e., layer 0).

Combined with the layer 0, the first layer protects applications from all VTable hijacking attacks if target applications have no dynamically generated code (e.g., just-in-time compiled code). Combined with the second layer, it blocks all VTable hijacking attacks even if there is dynamically generated code. Evaluating the performance of our prototype implementation shows that we achieve low overhead for standard benchmarks and browsers. A security evaluation using existing VTable hijacking exploits shows that our prototype successfully protects against attacks. We strongly recommend deploying the first layer of defense in practice, because it provides a very low performance overhead, and a good modularity support and a strong security enhancement.

Acknowledgements

We thank the anonymous reviewers for their constructive suggestions on this work. This material is based upon work supported, in part, by the National Science Foundation under Grant No. CNS-1513783 and CNS-1464155, DARPA award HR0011-12-2-005 and FA8750-14-C-0118, and FORCES (Foundations Of Resilient CybEr-Physical Systems), which receives support from the National Science Foundation (NSF award numbers CNS-1238959, CNS-1238962, CNS-1239054, CNS-1239166). It is also supported, in part, by the National Science Foundation of China under Grant 61402125 and 61572149.

References

[1] Nergal, "The advanced return-into-lib(c) exploits," *Phrack*, vol. 11, no. 58, p. http://phrack.com/issues.html?issue=67&id=8, 2007.

[2] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *CCS: ACM Conference on Computer and Communications Security*, 2007.

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming without Returns," in *CCS: ACM Conference on Computer and Communications Security*, 2010.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: a new class of code-reuse attack," in *ASIACCS: ACM Symposium on Information, Computer and Communications Security*, 2011.

[5] A. van de Ven and I. Molnar, "Exec Shield," https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[6] PaX-Team, "PaX ASLR (Address Space Layout Randomization)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[7] E. Hiroaki and Y. Kunikazu, "ProPolice: Improved stack-smashing attack detection," *IPSJ SIG Notes*, pp. 181–188, 2001.

[8] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *PLDI: Conference on Programming Languages Design and Implementation*, 2009.

[9] ——, "CETS: Compiler Enforced Temporal Safety for C," in *ISMM: International Symposium on Memory Management*, 2010.

[10] V. Kuzentsov, L. Szekeres, M. Payer, G. Candea, D. Song, and R. Sekar, "Code pointer integrity," in *OSDI: Usenix Symposium on Operating Systems Design and Implementation*, 2014.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *CCS: ACM Conference on Computer and Communications Security*, 2005.

[12] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *OSDI: Usenix Symposium on Operating Systems Design and Implementation*, 2006.

[13] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 380–395. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.30

[14] P. Philippaerts, Y. Younan, S. Muylle, F. Piessens, S. Lachmund, and T. Walter, "Code pointer masking: Hardening applications against code injection attacks," in *DIMVA: Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2011.

[15] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating Code-reuse Attacks with Control-flow Locking," in *ACSAC: Annual Computer Security Applications Conference*, 2011.

[16] B. Zeng, G. Tan, and U. Erlingsson, "Strato: A Retargetable Framework for Low-level Inlined-reference Monitors," in *Usenix Security Symposium*, 2013.

[17] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 559–573. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.44

[18] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, and L. Szekeres, "Protecting Function Pointers in Binary," in *ASIACCS: ACM Symposium on Information, Computer and Communications Security*, 2013.

[19] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 337–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534796

[20] B. Niu and G. Tan, "Monitor Integrity Protection with Space Efficiency and Separate Compilation," in *CCS: ACM Conference on Computer and Communications Security*, 2013.

[21] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *IEEE Symposium on Security and Privacy*, 2014.

[22] B. Niu and G. Tan, "Modular Control-flow Integrity," in *PLDI: Conference on Programming Languages Design and Implementation*, 2014.

[23] 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.

[24] Microsoft, "Visual Studio 2015 Preview: Work-in-Progress Security Feature." [Online]. Available: http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx

[25] C. Tice, "Improving function pointer security for virtual method dispatches," in *GNU Tools Cauldron Workshop*, 2012.

[26] Microsoft, "Software vulnerability exploitation trends: Exploring the impact of software mitigations on patterns of vulnerability exploitation (2013)," http://download.microsoft.com/download/F/D/F/FDFBE532-91F2-4216-9916-2620967CEAF4/Software%20Vulnerability%20Exploitation%20Trends.pdf, 2013.

[27] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting Virtual Function Tables' Integrity," in *NDSS: IS Network and Distributed System Security Symposium*, 2015.

[28] 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 *Oakland*, 2015.

[29] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks," in *20th Annual Network and Distributed System Security Symposium*, 2014.

[30] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 81–96. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee

[31] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines," in *the Network and Distributed System Security Symposium (NDSS)*, 2015.

[32] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sep. 2006.

[33] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Usenix Security Symposium*, 2005.

[34] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 177–192. [Online]. Available: http://blogs.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu

[35] "Itanium C++ ABI," http://mentorembedded.github.io/cxx-abi/abi.html.

[36] BlueLotus Team, "Bctf challenge: bypass vtable read-only checks," https://github.com/ctfs/write-ups-2015/tree/master/bctf-2015/exploit/zhongguancun, 2015.

[37] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 21st ACM Conference on Computer and Communications Security*. ACM, 2014.

[38] I. Haller, E. Gkta, E. Athanasopoulos, G. Portokalidis, and H. Bos, "ShrinkWrap: VTable Protection without Loose Ends." ACM Press, 2015, pp. 341–350, 00000. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2818000.2818025

[39] R. Gawlik and T. Holz, "Towards automated integrity protection of C++ virtual function tables in binary programs," in *ACSAC: Annual Computer Security Applications Conference*, 2014.

[40] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Symposium on Code Generation and Optimization*, 2004.

[41] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *ECOOP'91 European Conference on Object-Oriented Programming*. Springer, 1991, pp. 21–38.

[42] Microsoft IE, "LiteBrite: HTML, CSS and JavaScript Performance Benchmark," http://ie.microsoft.com/testdrive/Performance/LiteBrite/, 2014.

[43] Google, "Octane JavaScript benchmark suite," https://developers.google.com/octane/, 2014.

[44] Mozilla, "Kraken 1.1 javascript benchmark suite," http://krakenbenchmark.mozilla.org/, 2014.

[45] Apple, "Sunspider 1.0.2 javascript benchmark suite," https://www.webkit.org/perf/sunspider/sunspider.html, 2014.

[46] RightWare, "Browsermark 2.1 benchmark," http://browsermark.rightware.com/, 2014.

[47] FutureMark, "Peacekeeper: HTML5 browser speed test," http://peacekeeper.futuremark.com/, 2014.

[48] 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.

[49] M. R. Miller, K. D. Johnson, and T. W. Burrell, "Using virtual table protections to prevent the exploitation of object corruption vulnerabilities," 2014, uS Patent 8,683,583.

[50] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy*, 2013.

[51] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for c with very low overhead," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 162–171. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134309

[52] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Softw. Pract. Exper.*, vol. 27, no. 1, pp. 87–110, Jan. 1997. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(199701)27:1⟨87::AID-SPE78⟩3.0.CO;2-P

[53] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247362

[54] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," in *IEEE Symposium on Security and Privacy*, 2008.

[55] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[56] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out Of Control: Overcoming Control-Flow Integrity," in *IEEE Symposium on Security and Privacy*, 2014.

[57] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection," in *Usenix Security Symposium*, 2014.

[58] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *Usenix Security Symposium*, 2014.

[59] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40. ACM, 2005, pp. 190–200.