VPV: Enforcing Secure C++ Dynamic Dispatch by Vtable Pointer Verification

Xiaokang Fan, Sifan Long, Chun Huang, and Canqun Yang

***Abstract*—C++ is a very popular object-oriented programming language. Due to its abstraction and high performance, C++ has been widely used in performance critical applications. During the last several years, vtable hijacking attack has become a major attack vector. By corrupting the vtable pointer of a C++ object, an attacker can hijack a virtual call and compromise the control flow of a C++ program.**

This paper proposes VPV (Vtable Pointer Verification), a new method to mitigate vtable hijacking attacks. The novelty of VPV is that VPV enforces virtual call integrity by verifying the legitimacy of the vtable pointer, which is the key part in vtable hijakcing attacks. We use class hierarchy analysis to build a fine-grained control flow graph, which determines the legitimate targets precisely for each vtable pointer. We designed an efficient runtime verification technique which requires only a range check. The average (maximum) runtime performance overhead incurred is only 1.52% (2.30%). VPV provides precise and efficient protection against vtable hijacking attacks. It is an ideal technique to be applied to production software.

***Index Terms*—C++, vtable hijacking attack, vtable pointer**

I. Introduction

C++ is an object-oriented language with direct low level memory access. C++ provides both high level abstraction and high performance. A lot of performance critical applications has adopted C++ as the main language. Examples of large C++ applications include major web browsers like Chrome and Firefox, language runtimes like Oracle's Java Virtual machine, and so on.

Unfortunately, programs written in C++ is neither memory safe nor type safe. Therefore, programs written in C++ are prone to memory errors like buffer overflow and use after free (UAF). Attackers now increasingly exploit memory vulnerabilities to compromise C++ programs. Although some well-known mitigation techniques like Data Execution Prevention (DEP) [1], stack canaries [2] and Address Space Layout Randomization (ASLR) [3] have been broadly deployed. Attackers can still compromise a program by the so-called *vtable hijacking attack* [4], [5], [6], which is among the most popular control flow hijacking attacks against C++ programs. For example, over 80% of attacks against Google's Chrome and more than 50% of known attacks against Windows 7 exploit UAF vulnerabilities and virtual calls [7].

C++ relies on dynamic dispatch to implement polymorphism. C++'s dynamic dispatch is implemented using a data structure called virtual function table (*vtable*) in modern compilers, such as LLVM [8] and GCC [9]. A vtable is a table containing a set of pointers to all the virtual functions defined by a polymorphic class. The compilers put vtables in the read only section of a program. Thus they can not be modified. However, vtable pointers (*vptrs*) are stored in objects which are allocated in the heap or stack, both of which are writable sections. An attacker can hijack a virtual call through the writable vptr in the following three steps: (i) exploiting an initial memory corruption error already existed in the program; (ii) corruption of the vptr, redirecting it to a counterfeit or an existing vtable; (iii) hijacking the program to a piece of malicious code.

Enforcing full memory safety [10], [11], [12] prevents both temporal and spartial memory errors in the program. A vtable hijacking attack will be stopped in its first step. However, the high runtime overhead (over 100% performance slowdown) has prevented memory safety measures from being widely deployed.

Control Flow Integrity (CFI) [13] significantly raises the bar against vtable hijacking attacks by keeping the control flow of the program in a pre-computed Control Flow Graph (CFG). Every virtual call is verified by runtime checks that a legitimate target function, instead of malicious code controlled by the attacker, is called. However, no existing CFI work is capable of stopping the full spectrum of vtable hijacking attacks.

Insights Observe that mitigating a virtual hijacking attack from neither its first step (memory safety measures) nor its last step (CFI) can ensure full safety of dynamic dispatch. Thus the correctness of vptr is the key factor for virtual call integrity.

Solution We propose VPV (**V**table **P**ointer **V**erification), a lightweight approach to enforce secure dynamic dispatch by stopping a vtable hijacking attack from its second step. The novelty of VPV is twofold: (i) VPV enforces a precise CFG constructed by using *Class Hierarchy Analysis* (CHA). (ii) VPV implements a lightweight runtime verification which only consists a range check instead of a time consuming membership test, thus making performance slowdown negligible.

Contributions In this paper, we make the following contributions:

- We propose VPV, an effective and efficient technique for mitigating virtual call hijacking attacks in C++ programs.

Manuscript received December 3, 2020; revised May 28, 2021. This work was supported by the National Key Research and Development Program of China (No.2020YFA0709800), and the National Science Foundation of China (No.62002367).

X. Fan is an Assistant Professor of School of Computer, National University of Defense Technology, Changsha, 410073, Hunan, P.R.China (corresponding author, phone: +86 13787208712, email: fanxiaokang@nudt.edu.cn)

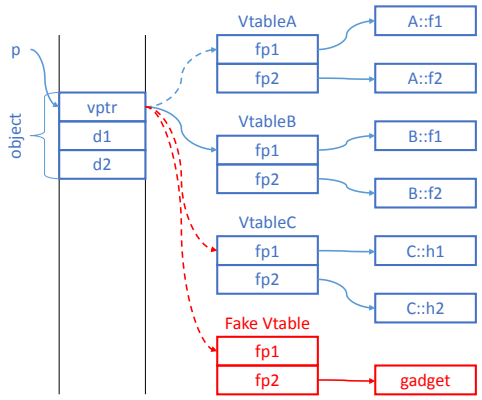
S. Long is a PhD candidate of School of Computer, National University of Defense Technology, Changsha, 410073, Hunan, P.R.China (email: 164712110@csu.edu.cn).

C. Huang is a Professor of School of Computer, National University of Defense Technology, Changsha, 410073, Hunan, P.R.China (email: chunhuang@nudt.edu.cn).

C. Yang is a Professor of School of Computer, National University of Defense Technology, Changsha, 410073, Hunan, P.R.China (email: canqun@nudt.edu.cn).

```
 1:class A {
 2:    public: double d1;
 3:        virtual void f1();
 4:        virtual void f2();
 5:};
 6:class B: public A {
 7:    public: double d2;
 8:        virtual void f1();
 9:        virtual void f2();
10:};
11:class C {
12:    public: double d3;
13:        virtual void h1();
14:        virtual void h2();
15:};
16:
17:A *p = new B;
18:p->f2();  //callee: B::f2
```

(a) C++ source code



(b) Memory layout

Fig. 1. An example illustrating the dynamic dispatch mechanism of C++.

- We present a prototype implementation of VPV upon LLVM-10.0.0.
- We conduct a thorough evaluation using nine C++ programs from SPEC CPU2000/CPU2006/CPU2017. The average (maximum) runtime performance overhead is 1.52% (2.30%).

The rest of the paper is organized as follows: Some background knowledge on dynamic dispatch and vtable hijacking attack is provided in Section II. We define the threat model in Section III. The design and implementation details of VPV are described in Section IV. Followed by a thorough evaluation in Section V. Section VI discusses the related work. We conclude the paper in Section VII.

II. Background

In this section we provide some background knowledge including: the dynamic dispatch mechanism of C++ (Section II-A) and vtable hijacking attack (Section II-B).

A. Dynamic Dispatch

C++ relies on dynamic dispatch to achieve polymorphism. For a virtual function call in C++, *static type* denotes the type of the pointer that invokes the virtual call, while *dynamic type* denotes the runtime type of the object that the pointer refers to. The actual function invoked at runtime by a virtual call is determined by the dynamic type rather than the static type, thus achieving polymorphism.

Figure 1 illustrates dynamic dispatch through a simple example. There are three classes: class `A`, class `B`, and class `C`. Class `B` is a derived class of class `A`. Each class defines two virtual functions. At line 17, the program creates an object of type `B` and then assigns its address to a pointer `p` of type `A*`. At line 18, the pointer `p` is used to invoke a virtual call. At runtime, the static type and dynamic type of the virtual are `A` and `B`, respectively. Therefore, the actual function called should be `B::f2`.

Dynamic dispatch is implemented using a data structure called virtual function table (vtable) in Modern compilers (e.g., LLVM and GCC). As shown in Figure 1(b), The compiler generates one vtable for each polymorphic class (a class that defines virtual functions or inherits virtual functions from its parent classes). A vtable is an array of function pointers. Each entry of a vtable holds the address of a virtual function defined by the corresponding class. For example, class `B` defines two virtual functions `B::f1` and `B::f2`. Therefore `VTableB` has two entries: `VTableB[0]` points to `B::f1`, and `VTableB[1]` points to `B::f2`. For each object of a polymorphic class, the compiler generates a vtable pointer and put the pointer right in the head of the object. The vptr points to the vtable of the object's class. As depicted in Figure 1(b), vptr locates at the first field of the object created at line 17 in Figure 1(a), followed by two data fields: `d1` and `d2`.

The virtual call at line 18 is dispatched in the following four steps: (1) obtaining vptr, which is the address of `VtableB`; (2) obtaining vptr[1], the address of `VtableB[1]`; (3) loading the function pointer stored in `VtableB[1]`, and that function pointer points to the target function `B::f2`; (4) transferring the control flow to the target function `B::f2` and continuing execution.

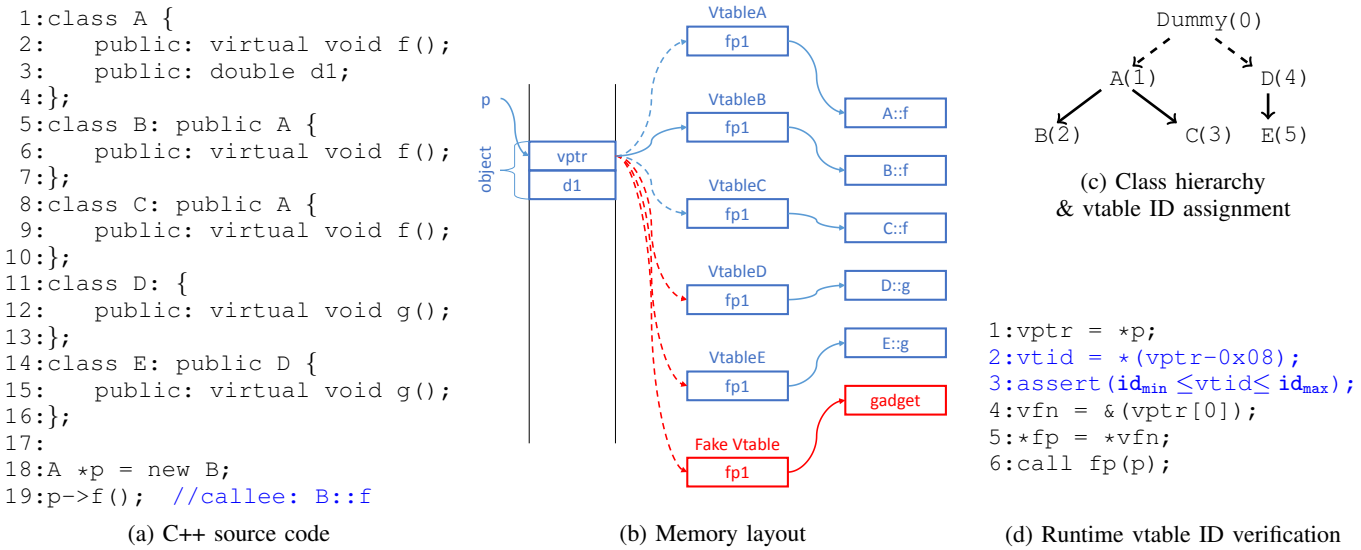
B. Vtable Hijacking Attack

The first step of a vtable hijacking attack is to direct the vptr of an object to a location of the attacker's choice. Attackers can often achieve this goal by exploiting memory corruption bugs or type confusion bugs. Whenever the corrupted object is used to invoke a virtual call, the attacker can redirect the virtual call to some malicious code through the corrupted vptr. Thus compromising the program.

The dashed lines in Figure 1(b) presents an example. The attackers can make the vptr point to `VTableA`, `VTableC`, or even a fake vtable created by the attacker instead of `VTableB`, by corrupting the object created at line 17. Therefore, function `A::f2`, `C::h2` or a code gadget selected by the attacker can be invoked at the virtual callsite `p->f2()`.

III. Threat Model

The attackers can perform arbitrary reads of memory and write to all writable memory addresses. However, we assume that the attackers can not perform arbitrary writes. This work exclusively focuses on protecting virtual calls in C++ programs. Other vulnerabilities such as memory corruption or type confusion are out of scope. Auxiliary defense mechanisms for these vulnerabilities are assumed to be deployed.



(a) C++ source code (b) Memory layout (c) Class hierarchy & vtable ID assignment (d) Runtime vtable ID verification

Fig. 2. An example illustrating VPV: (a) C++ source code; (b) Memory layout; (c) Class hierarchy & vtable ID assignment, dashed lines represent artifical inheritence from the Dummy node, while solid lines represent real inheritence relations; (d) Runtime vtable ID verification in pseudo IR, the code with blue color (line 2 and line 3) are the instrumented runtime verification.

IV. VTABLE POINTER VERIFICATION

In this section, we will describe our vtable pointer verification method. Our key idea is to verify that the vptr points to a legitimate vtable during a virtual call. By enforcing the legitimacy of the vptr, a vtable hijacking attack can be stopped in its second step.

A mitigating technique often consists of two parts: i) a static analysis to construct the CFG enforced during runtime; and ii) a runtime verification technique to verify the legitimacy of the control transfer targets. The precision of the static analysis determines the effectiveness of the mitigating technique, which is how much safety insurance the protection technique provides. The more precise the static analysis is, the stronger protection will be provided. While the performance overhead incurred by the runtime verification technique determines the efficiency of the mitigating technique. Lower runtime performance overhead usually means wider deployment of the protection technique to production software.

The success of VPV relies on two key components: (i) a precise static analysis to determine the legitimate target vtable sets for each vptr (Sectoin IV-A); and (ii) an efficient runtime verification technique to enforce the legitimacy of vptrs (Section IV-B).

A. Class Hierarchy Analysis

According to the semantics of C++, the types of objects pointed to by a pointer to type `T` include `T` or any subclass of `T`. As a result, when a pointer of type `T` is used to invoke a virtual call, the vptr is allowed to point to vtables of `T` or any subclass of `T`.

The legitimate target vtable sets of vptrs can be constructed by the Class Hierarchy Graph (CHG). A CHG is Directed Acyclic Graph (DAG). Each node in the CHG represents a class. While each edge represents an inheritance relation. For each node in the CHG, its predecessor nodes and successor nodes represent its base classes and derived classes, respectively.

We use Class Hierarchy Analysis (CHA) to construct the CHG. A node for each polymorphic class is created in the CHG. Then we use constructor-destructor analysis to build edges between nodes.

An object of a polymorphic class is created by calling its constructor. The constructor creates the vptr and makes it point to the VTable of the object's class. Then the constructor stores the vptr in the object.

For a class with multiple base classes, the "primary base class" denotes its first base class, while "secondary base classes" represent all the remaining base classes. An object of a derived class is made up of several sub-objects, each sub-object corresponds to a base class. Similarly, the VTable of a derived class is made up of several sub-VTables, each corresponding to a VTable of a base class.

During the construction process of an object, constructors of its base classes are first invoked to initialize all the sub-objects before initializing its own vptr. The constructor of the primary base class will be the first one to be invoked, followed by constructors of secondary base classes in the declaration order.

Similarly, The destructors of all the base classes are called in a reverse order (from the last secondary base class to the primary base class) in the destructor of a derived class. Besides, unlike the construction process which initialize the vptr after all base classes are constructed, the destructor assigns the vptr before invoking the destructors of base classes.

We scan all the constructors and destructors in the program to build the edges in the CHG. For each constructor, all the constructors called before the initialization of the vptr are extracted. While for each destructor, we extract all the calls to destructors after the assignment of the vptr. For each call extracted, an edge from the class of the callee to the class of the caller is created in the CHG, representing a base→derived relation.

B. Runtime Verification

We designed a highly efficient runtime verification technique to validate the legitimacy of vtable pointer targets. The key idea is an efficient ID range check instead of a time consuming membership test.

Traditional protection techniques verify control transfer targets by membership test (i.e., by comparing the runtime target to the statically computed legitimate targets one by one.) A vptr of a virtual call in the program may have hundreds or even more legitimate targets. Such large number of legitimate targets means high performance overhead incurred by membership test. Larger C++ applications will exacerbate this problem since larger applications usually have larger class hierarchies and more legitimate virtual call targets.

Our runtime verification transforms an expensive membership test into an efficient ID range check. Two phases are needed: (a) a static phase to assign a unique ID to every vtable in the program; and (b) a runtime phase to verify the ID of a target vtable during a virtual call.

a) VTable ID Assignment: In C++, the CHG must be a DAG. Every edge in the graph represents an inheritance relation, pointing from a base class to one of its derived classes. By adding a dummy class and connecting the dummy node to root nodes of every subtrees, the whole graph can be transformed into a single tree with the dummy node as the root. We can then assign a unique ID to every node in the tree continuously by traversing the tree in a preorder. This ID will be assigned to the vtables corresponding to each class in the tree.

b) VTable ID Verification: For every subtree, the IDs are continuous. While for subtrees with different root, their IDs do not overlap. Therefore, for each virtual call, the ID of the legitimate vtables must be in a unique range. During a virtual call, by checking whether the ID of the target vtable is in the legitimate range, which comprises only two comparisons, the legitimacy of the target vtable can be verified.

Example 1: An example that explains VPV is presented in Figure 2. As shown in Figure 2(a), there are five classes: `A`, `B`, `C`, `D`, and `E`. Classes `B` and `C` inherit from class `A`. While class `E` inherits from class `D`. In line 18, the program creates a pointer `p` of type `A *` and then make it point to an object of type `B`. Then pointer `p` is used to invoke a virtual function `f` in line 19. As annotated in line 19, the callee of the virtual call is `B::f`.

Figure 2(b) depicts the memory layout. The object created in line 18 has two fields: the implicit vptr which points to `VTableB`, and a data filed `d1`.

Figure 2(c) presents the class hierarchy constructed using our Class Hierarchy Analysis described in Section IV-A. There are two subtrees (rooted by `A` and `D`, respectively) in the class hierarchy. With the help of the `Dummy` node, different subtrees can be connected into a single tree with the `Dummy` node as the root. A unique ID starting from 0 can be assigned to each node continuously by traversing the whole tree in a preorder.

The runtime verification is presented in Figure 2(d). Line 2 and line 3 are the instrumented verification code. We use a 64-bit word to store a vtable's unique ID. A vtable's ID is inserted right before it. Line 2 loads the vtable ID from the address which is 8 bytes before the starting address of the vtable. Line 3 checks whether the ID of the target vtable is in the legitimate range. The type of the pointer that invokes the virtual call determine $\texttt{id}_{\texttt{min}}$ and $\texttt{id}_{\texttt{max}}$. According the semantics of C++, the legitimate vtables of the vptr are those whose type are derived from the type of the pointer that invokes the virtual call. Therefore, $\texttt{id}_{\texttt{min}}$ and $\texttt{id}_{\texttt{max}}$ are the min ID and the max ID of the subtree rooted by `p`'s type. As can be seen from Figure 2(c), $\texttt{id}_{\texttt{min}}$ and $\texttt{id}_{\texttt{max}}$ are 1 and 3, respectively.

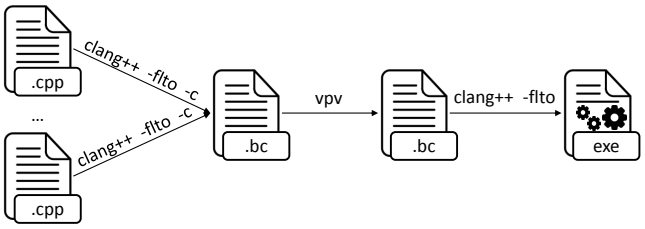


Fig. 3. Compilation process of VPV.

TABLE I
PROGRAM STATISTICS.

Program	KLOC	Application Area
blender	1577	3D rendering and animation
dealII	198	Finite Element Analysis
eon	41	Computer Visualization
leela	21	Artificial Intelligence: Monte Carlo tree search (Go)
omnetpp	134	Discrete Event simulation - computer network
parest	427	Biomedical imaging: optical tomography with finite elements
povray	170	Ray tracing
soplex	41	Linear Programming, Optimization
xalancbmk	520	XML to HTML conversion via XSLT

V. EVALUATION

In this section we present the details of our experiments. Results of nine C++ programs from SPEC CPU2000/CPU2006/CPU2017 show that VPV can provide strong protection with very small runtime performance overhead.

A. Compilation process

VPV is a tool built upon LLVM-10.0.0. VPV takes a C++ LLVM bitcode file as input and exports an instrumented LLVM bitcode file as output. Figure 3 presents the compilation process of VPV. First, all c++ source files are compiled into an LLVM bitcode file using `clang++` with the `-flto` option and the `-c` option. Then, VPV applies class hierarchy analysis and inserts runtime verification into the bitcode file. Finally, the instrumented bitcode file is transformed into the final executable binary file by using `clang++` with option `-flto`.

B. Experimental Setup

The platform on which we conducted all our experiments consists of a 3.60GHz Core i7-9700KF CPU with 16 GB memory. The OS used is Ubuntu Linux 16.04. The compiler we used is Clang-10.0.0. All programs are compiled at optimization level `-O3`. In our evaluation, we consider the following two different configurations:

TABLE II
OVERHEAD OF BINARY SIZE AND INSTRUCTION NUMBER.

Program	# of virtual call	Overhead	
		Binary size (%)	Inst Num (%)
blender	293	8.09	0.10
dealII	365	5.72	1.03
eon	59	4.04	0.44
leela	1	3.56	0.19
omnetpp	8952	1.70	11.54
parest	2432	0.54	1.83
povray	283	4.05	0.39
soplex	625	9.05	3.19
xalancbmk	27743	2.49	12.40
Average	N/A	4.36	3.46

1) **baseline**: All programs are compiled with Clang-10.0.0 at optimization level `-O3`. No runtime checks are instrumented.
2) **VPV**: All programs are compiled in the process described in Section V-A with runtime verification enabled.

C. Programs

We use nine C++ programs from SPEC CPU2000/CPU2006/CPU2017: blender(2017), dealII(2006), eon(2000), leela(2017), omnetpp(2017), parest(2017), povray(2017), soplex(2006), and xalancbmk(2017) to evaluate. The rest C++ programs from SPEC CPU2000/CPU2006/CPU2017 are not selected because there is no virtual call in them. Table I lists the size and application area of each program.

D. Static Statistics

First, we measure the static statistics in Table II, including the number of virtual calls, the overhead in binary size, and the overhead in the number of LLVM instructions.

Column 2 of Table II presents the number of virtual calls in each program. The largest program blender (1577 KLOC) is a C and C++ mixed program. Therefore it has only 293 virtual calls. The program xalancbmk is heavily dependent on dynamic dispatch, as it has the largest number (27743) of virtual calls. The smallest program leela (21 KLOC) has the least number (1) of virtual call.

The overhead of binary size is presented in column 3. VPV incurs the maximum overhead in binary size in program sopolex (9.05%). While the minimum overhead in binary size incurred is in program parest, which is only 0.54%. On average the overhead in binary size incurred by VPV is 4.36%.

Column 4 lists the overhead of the number of LLVM instructions. The number of LLVM instructions are collected in the final bitcode file generated using *link time optimization (lto)*. For programs omnetpp and xalancbmk, VPV incurred over 10% overhead in the number of LLVM instructions. While for programs blender, eon, leela and povray, the overheads incurred are less than 1%. The average overhead of LLVM instruction number is 3.46%.

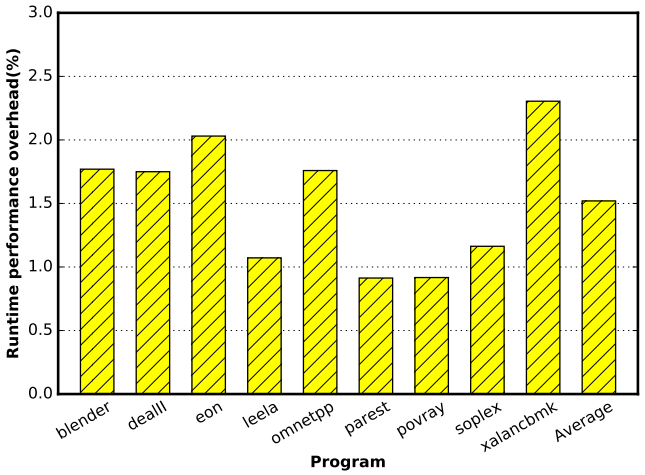


Fig. 4. Runtime performance overhead incurred by VPV compared with native runs.

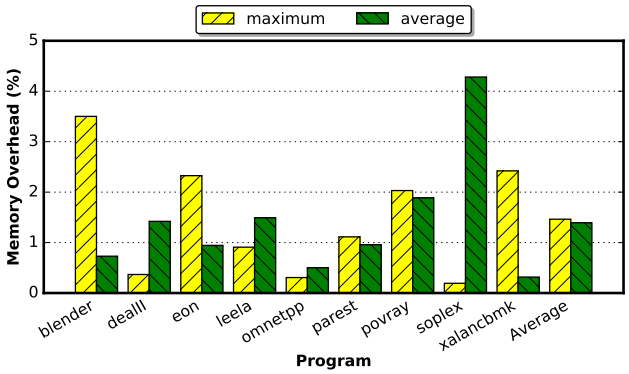


Fig. 5. Memory overhead (maximum and average resident set size (RSS)) incurred by VPV.

E. Performance Overhead

Figure 4 presents the runtime performance overhead of VPV. The overhead measures the slowdown in the execution time of the VPV configuration compared with the baseline configuration. The reference inputs are used for the SPEC programs.

As shown in Figure 4, for CPU intensive SPEC CPU C++ programs, the runtime performance overhead incurred by VPV is quite small. The maximum overhead comes from program xalancbmk (2.30%) as it has the largest number of virtual calls and those virtual calls are called frequently. The average runtime performance overhead is only 1.52%. Such low runtime performance overhead means that VPV is a suitable protection technique to be applied to production software.

F. Memory Overhead

The runtime memory overhead of VPV is also measured. We use the resident set size (RSS) to represent the memory footprint. Figure 5 shows the memory overhead in the maximum and average RSS. For all the programs, the largest overhead in maximum and average RSS are 3.50% and 4.28%, respectively. While on average, the overhead in maximum and average RSS incurred by VPV are 1.46% and 1.39%.

VI. Related work

The security of software is a major research topic and has drawn a lot of research attention during the last several decades [16], [17], [4], [18], [14], [19]. Memory safety techniques [10], [11], [12], [20] have been studied extensively. With memory safety techniques, control flow hijacking attacks can be stopped in the first step. Softbound+CETS provides full memory safety [10], [11]. However, the high runtime performance overhead ($> 100\%$) has prevented the widely application of it.

Abadi proposed Control Flow Integrity (CFI) [13] in 2005. CFI enforces a statically computed Control Flow Graph (CFG) during the runtime. Thus it can prevent an attacker from compromising the control flow of a program even if memory corruption errors took place. CFI has drawn a lot of attention since it was proposed. General CFI techniques [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [19] provide protection for both forward (indirect function call) and backward (return instruction) control flows. However, as they are often more focused on C programs and return instructions, the CFG of general CFI techniques are usually more coarse grained and class hierarchy information is generally missing. Therefore, general CFI techniques are still vulnerable to sophisticated vtable hijacking attacks [31]. Virtual call protection techniques [16], [17], [4], [15], [32], [5], [7], [14] usually focus more on the protection of virtual calls. The CFGs are more fine grained with class hierarchy information taken into consideration. Both source-level and binary-level mitigation techniques have been proposed.

VII. Conclusion

Vtable hijacking attack is a popular attack surface used by attackers to compromise C++ programs. The research community has proposed a lot of mitigating techniques. However, most of these works suffer from the problem of high runtime performance overhead. VPV is an effective and efficient technique against vtable hijacking attacks. Legitimate targets for virtual calls are determined by class hierarchy analysis, which is precise enough to provide a strong protection. The runtime verification of each virtual call only requires a range check, which incurs negligible runtime performance overhead. The effectiveness and efficiency of VPV make it an appropriate technique to be applied into production software.

References

[1] *MicroSoft data execution prevention*, https://support.microsoft.com/en-au/kb/875352.

[2] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[3] *PaX*, https://pax.grsecurity.net/docs/aslr.txt.

[4] D. Bounov, R. Kici, and S. Lerner, "Protecting C++ dynamic dispatch through vtable interleaving," in *NDSS '16*, 2016.

[5] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: securing C++ virtual calls from memory corruption attacks," in *NDSS '14*, 2014.

[6] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS '15*, 2015.

[7] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "Vtrust: Regaining trust on virtual calls," in *NDSS '16*, 2016.

[8] *The LLVM Compiler Infrastructure*, https://llvm.org/.

[9] *GCC, the GNU Compiler Collection*, https://gcc.gnu.org/.

[10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542504

[11] ——, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. New York, NY, USA: ACM, 2010, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1806651.1806657

[12] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS' 05*. ACM, 2005, pp. 340–353.

[14] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for c++," in *ITTSA '17*. ACM, 2017, pp. 329–340.

[15] 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 '14*, 2014, pp. 941–955.

[16] I. Haller, E. Göktas, E. Athanasopoulos, G. Portokalidis, and H. Bos, "Shrinkwrap: Vtable protection without loose ends," in *ACSAC '15*. ACM, 2015, pp. 341–350.

[17] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *S&P '16*, 2016, pp. 934–953.

[18] X. Fan, Z. Xia, S. Long, C. Huang, and C. Yang, "Accelerating type confusion detection with pointer analysis," *IAENG International Journal of Computer Science '20*, vol. 47, no. 4, pp. 664–671, 2020.

[19] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *CCS '15*. ACM, 2015, pp. 927–940.

[20] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746

[21] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *USENIX Security Symposium*, 2013, pp. 337–352.

[22] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *NDSS '15*, vol. 26, 2015, pp. 27–30.

[23] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses." in *USENIX Security Symposium*, 2014, pp. 385–399.

[24] 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*, vol. 14, 2014, pp. 401–416.

[25] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 575–589.

[26] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 941–951.

[27] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 577–587. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594295

[28] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 585–598.

[29] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient cfi enforcement with intel processor trace," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 529–540.

[30] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing." in *USENIX Security Symposium*, 2013, pp. 447–462.

[31] 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 *S&P '15*. IEEE, 2015, pp. 745–762.

[32] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries," in *NDSS '15*, 2015.