A Tough call: Mitigating Code-Reuse Attacks On The Binary Level

Anonymous

Institute of Informatics
Technische Universität München, Germany
anonymous@in.tum.de
December 15, 2018

Abstract. Binary-level control-flow integrity, unlike source-level solutions which can accurately infer the targets of indirect callsites and detect malicious control-flow transfers, is weak in determining the set of valid targets for indirect control flow transfers on the forward edge. However, considering that source code is not always available, offering similar quality of protection at binary level is of importance.
In the paper, the researchers propose binary-level analysis techniques to considerably reduce the number of possible targets for indirect callsites. Furthermore, they reconstruct a conservative approximation of target function prototypes by means of use-def analysis at possible callees. What's more, they come up with liveness analysis at each indirect callsite to derive a many-to-many relationship between callsites and target callees with a much higher precision compared to existing binary-level solutions. This prototype, *TypeArmor*, has achieved some goals such as it successfully breaks published COOP exploits.
In this report, I will describe this powerful binary-level CFI solution published by *S&P* in 2016, and then compare it with other binary-level solutions as well as source-level solutions.

1 Introduction

Extracting an accurate Control Flow Graph from the binary level is an undecidable problem. As the CFG is the main requirement for control flow integrity, this makes CFI at the binary level hard in practice. In this case, most existing binary-level CFI methods choose to base their invariants on an *approximation* of the CFG, which leaves enough chances for attackers to launch successful exploits.
The goal of this solution is *not* that all possible attacks can be stopped: even the strictest CFI solutions *with* access to source code are unable to guarantee one hundred percent protection against all possible attacks. Nevertheless, the solution, *TypeArmor*, the binary-level protection prototype, can stop all COOP attacks published to date and considerably raise the bar for an adversary. It is worth noting that. *TypeArmor* provides strong mitigation for many types of code-reuse attacks for programs binaries, without requiring access to source

code.
TypeArmor deploys a forward-edge CFI strategy that relies on conservatively reconstructing both callee invariants and callsite invariants and it uses this information to restrict that each callsite only targets matching functions *strictly*. For instance, it is not allowed for a callsite to call a function that the number of argument the function consumes is more than the number of arguments the callsite prepares. Additionally, *TypeArmor* deploys a novel protection policy, term as Control-Flow Containment, which further reduces the possible target set of callees for each callsite.
In this report, I will first give a high-level overview of this solution *TypeArmor*, then I will explain in more detail about it when it comes to static analysis as well as runtime enforcement. Next, I will list its achieved success including how completely it stops COOP exploits against IE, Firefox and Chrome. Also, I will display some COOP extension, which could be further work of *TypeArmor*. Then I will further my report to have a comparison between *TypeArmor* with other binary-level CFI solutions as well as source-level prototypes. Finally, I will summarize this prototype and come to a conclusion.

2 Overview

In this section, I will first introduce a high-level overview of *TypeArmor* and then explain how *TypeArmor* impacts COOP exploits.

2.1 What is *TypeArmor*

As the author introduces that *TypeArmor* deploys a combination of two type-based control-flow invariants: target-oriented invariants and callsite-oriented invariants, resulting in a strict forward-edge protection strategy. It is worth noting that the callsite-oriented invariants have not been explored at binary level before, while target-oriented invariants are based on traditional CFI policies. Here *TypeArmor* enforces callsite-oriented invariants through a novel containment technique which termed as Control-Flow Containment(CFC). [1]
To be more specific, *TypeArmor* only allows that indirect callsites that set at most *max* arguments cannot call functions that consumes more than *max* arguments. Additionally, *TypeArmor* ensures that indirect callsites that expect a return value, which in other words, is non-void callsites, can never target a callee of type void.
In order to be conservative and support existing program functionality, *TypeArmor*'s callsite analysis may only report an overestimation of the number of prepared arguments, on the other hand, the callee analysis should report only underestimation.

2.2 How does it impact COOP

TypeArmor's CFC enforces a maximum number of arguments prepared at a callsite and scrambles the unused registers, resulting in a severe impact on the

ability of an attacker to enable data flow between gadgets. [2]
Furthermore, *TypeArmor*'s CFI implementation reduces the target set of the virtual function calls by the main-loop and recursive gadgets considerably. Generally, it prohibits any forward edges to functions that expect more arguments than the callsite prepares.

3 Static Analysis

Static analysis in *TypeArmor* aims to detect (i) the minimum number of consumed arguments at possible callees, (ii) the maximum number of prepared arguments at indirect callsites, and (iii) non-void callsites and void callees.

3.1 Callee Analysis

The callee analysis focuses on collecting state information on each registers to determine if they are used for passing arguments or not. The state of a register can be distinguished as the following classes: *read-before-write*(R), which means data are always read from this register before new data are written to it, *write-before-read*(W),which indicates that this register is always written to before it is read, or *clear/untouched*(C), which means this register is never read or written to.

Step 1) Forward Analysis
The analysis starts at the entry basic block of an address-taken function and iterates over the instructions to determine the state of registers. If all argument registers are determined either R or W, the analysis terminates. Otherwise, a recursive forward analysis starts until the block has no outgoing edges.
A recursive analysis loops over all outgoing edges of the basic block to get a pointer to the next basic block to analyze. Depending on different types of edge, which can be categorized into direct calls, indirect calls, return instructions, and regular outgoing edges, different operations will be followed.

- *Direct calls:* For direct calls, the next basic block to analyze is the entry block of the target function.
- *Indirect calls:* The analysis cannot statically infer the target of the indirect calls. Due to conservative principle, it is assumed that the target writes all arguments, which means all registers are in W state, and thus stops the recursion.
- *Returns:* For return instructions, we pop a fall-through basic block from the stack and use it as the next basic block in the analysis. An empty stack indicates the end of the analyzed function and terminates the recursive analysis.
- *Other:* We handle other edge types in the same way: the targets of the edge are set as the next basic blocks in the analysis.

Step 2) Merging Paths
The set of states $S_i (i = 1, 2, ...n)$, which is returned by *TypeArmor* static analysis for a basic block B, means that it has n outgoing edges. Each state, which is a vector, represents argument registers' states for each edge i. *TypeArmor* combines these vectors into a *superstate* S. Due to conservative principle, the state of a certain register can only be R in S if the state is R in every S_i .

Step 3) Counting Arguments
"Once the recursive analysis converges to a definite state for the entry basic block of a function, the argument count is set using the highest argument register that is marked as R." The author describes this step very simple, and it is pretty straightforward. Due to calling conventions in x86, where the sequence of using is rdi, rsi, rdx, rcx, $r8$, $r9$, it means that if $r9$ is in state R, the previous five registers are all used, thus we conclude this function expects 6 arguments, otherwise we continue determining r8's state.

3.2 Callsite Analysis

TypeArmor detects over each indirect callsite and does a backward static analysis to detect how many argument registers will be set at a specific callsite. The states of argument registers can be categorized into two types: set(S) or not(T,*trashed*).

1) Backward Analysis
TypeArmor starts the analysis at the basic block that contains the indirect call, and iterates over preceding instructions for determining the argument registers' states. If all argument registers are S, *TypeArmor* terminates the analysis and assumes that the callsite prepares the maximum number of arguments, otherwise *TypeArmor* starts a recursive backward analysis.
Similar to what happened to forward analysis in callee, which basic block will be detected next depends on different edge type.
   - *Direct calls:* For direct calls, the preceding basic block to analyze next is the basic block where the direct call originated. Once the backward analysis reach the entry block of the function which contains the inspected callsite, an inter-procedural backward analysis at all the callers of this function is initiated.
   - *Indirect call:* Since indirect call targets cannot be resolved statically, there are no indirect call edges.
   - *Return:* When there is a return edge, it means the currently analyzed basic block has a predecessor that performs a function call. As a result, traversing further in this path is stopped and all remaining argument registers are marked as T.

2) Merging Paths
Path merging for the callsite backward static analysis is simpler than what needs to be done in callee analysis. For all collected states of the incoming

basic blocks, T always supersedes S.
Similar to the forward analysis, once the recursive analysis is finished, the number of prepared arguments is set based on the states of the last write operations.

3.3 Return Values

It is more precise for *TypeArmor*'s CFI implementation to add information about return value. "If we find a callsite that expects a return value (a non-void callsite), it should never target a callee that does not prepare a return value (void functions)," It will add protection strength for *TypeArmor* to restrict the incompatible matching.
Defining return usage information of a certain callee and callsite analysis is also conservative, which means a void callsiite is allowed to target both void and non-void callees.

1) *Non-void Callsites:* By seraching for *read-before-write* operations on the register that holds return values (rax for the System V ABI), a callsite is void or not can be defined. In short, by applying the forward analysis starting from the callsite and only for rax.
2) *Void Callee:* Contrast to detecting non-void callsites, we apply the previously introduced backward analysis at the exit points of a function. The backward analysis only searches for write operations on rax which may indicate a set return value.

4 Runtime Enforcement

After discussing how *TypeArmor* analyze callee and callsite statically, we have a look at how *TypeArmor* provide security guarantees at runtime in this section. The runtime enforcement is composed by three parts:(i) shadow code memory preparation, (ii)CFI enforcement, and (iii) CFC enforcement.

1) Shadow Code Memory Preparation
   The shadow code is an exact copy of the original code that also contains the instrumentation of the callsites, where program execution is actually performed.
   Whenever reaching an indirect callsite during normal program execution, the instrumentation code at this location performs an integrity check between the type of the callsite and the type of the callee. If the types are compatible with each other, the callsite is allowed to target the callee, otherwise it is not allowed to perform.
   We perform the integrity check by retrieving and processing the function's label, located right before the function entry point in the original code region. By using this shadow code memory, there is no need to worry the label will overwrite code.

2) CFI Enforcement

"*TypeArmor* instruments binaries for enforcing that callsites can only target functions with a compatible *type*." This essentially means *TypeArmor* only allow (i) a callsite with a higher number of prepared arguments target all the functions that any callsite with a lower number of prepared argument can also target, but not vice versa, and (ii) a callsite that expects a return value can only target functions that return a value, however, a callsite that does not set a return values can target both functions that are void and non-void. I will describe callee instrumentation and callsite instrumentation separately.

1) *Callee instrumentation:* We add a label of each address-taken function. There are seven possible labels: from no argument (0) to all arguments (6). Therefore, we use a 3-bit representation. In addition, we use one more bit at the lowest position to represent whether the function returns a value: we use 1 to encode *void* functions and 0 for *non-void* functions. For example, we represent the bits of a *non-void* function that has four arguments as 1000.

   As the author describes, in order to have a unique combination of four bytes that does not occur at any other code location, they choose 0xCCCCCC40 as the base label and use the four least signicant bits to encode the function type.

2) *Callsite instrumentation:* At each callsite, *TypeArmor*'s runtime component inserts a check to determine if the callee is of compatible type for the callsite to target. It does so by retrieving the callee's label, decoding the type and check if the result is compatible with the callsite. To be more specific, the instrumented check does the following:

   Step 1) Get the address of the function.
   Step 2) Point into the original code region.
   Step 3) Read the target's memory.
   Step 4) Perform *xor* instruction with the base label to get the function type.
   Step 5) For non-void callsites, make sure that the last bit is 0.
   Step 6) Using an unsigned comparison, check compatibility.

3) CFC Enforcement

As a new terminology, the author introduces that CFC is what they use for scrambling unused registers at indirect callsites. For example, consider an address-taken function *f* that accepts five arguments, but for which *TypeArmor* conservatively concludes that it accepts at least two arguments. Now, consider an indirect callsite *cs* for which *TypeArmor* assumes that it sets no more than three arguments. By enabling CFC, *TypeArmor* instruments *cs* in such a way that the last three argument registers are initialized with a random value at the callsite, which we term it as *scrambled*. Without enforcing CFC, *cs* is allowed to target *f*. However, with CFC in place, it will not change the fact that it is still allowed for *cs* to target *f*, but what it really does is that when the function set the last three argument registers, which are already been scrambled, the program is about to crash.

5 In practice

In this section, we first discuss advanced code-reuse attacks in more detail, COOP in particular. Then, we will have a specific example of how *TypeArmor* stops practical COOP exploits for Internet Explorer, Firefox and Chrome. Next, we will discuss about the Control Jujutsu exploits. And then we will dive into other possibilities of COOP exploitation. Finally, we will talk about pure data-only attacks.
From this table below provided in the paper, we can conclude a short summary of how *TypeArmor* addresses recently published code-reuse attacks.

Exploit	Stopped?	Notes
IE(32-bit)	✗	Out of scope
IE(64-bit)	✓	Argcount mismatch
Firefox	✓	Argcount mismatch
Chrome	✓	Argcount mismatch, Void target where non-void was expected
Apache	✓	Target function not address-taken
Nginx	✓	Void target where non-void was expected

1) Effectiveness against COOP
   As summarized in the COOP paper, there are three ways for data flow as explained as concluded in COOP paper:(i) data flow using unused argument registers, (ii) data flow using overlapping counterfeit object fields or global variables, and (iii) data flow by relying on arguments actually passed to the callee. The first way is called *implicit* data flow, and the remaining two ways are called *explicit* data flow.
   In COOP paper, the last two methods are proved hard to practice in reality, so we can conclude that COOP relies on unused argument registers to enable data flow between gadgets. We are interested in how many of those spurious arguments remain when *TypeArmor* is in place.
   From a test of accuracy of *TypeArmor* compared to the ground truth for different server applications, it can determine the exact number of prepared arguments for 103 out of 130 indirect callsites (geometric mean). Even though this percentage is fairly promising enough, the missing 27 callsites are still dangerous and could be used by attackers to allow data flow. But with *TypeArmor* in place, the attack surface is limited drastically.

2) Stopping COOP Exploits in Practice

   1) *Exploit on 64-bit IE:* There are two ways for exploiting against 64-bit IE which are published in COOP paper. Both exploits start with the same

main loop. For the callsite, *TypeArmor* will detect that it set at most one argument, however, for the callee, which is a series of gadgets, most of them will need two arguments. In this case, *TypeArmor* will not allow this callsite to target this callee, which successfully stops the exploit.

2) *Exploit on 64-bit Firefox:* We examined COOP's exploit on Firefox and also we can find the main loop gadget prepares only one argument, however, functions always expect at least two arguments. This means that *TypeArmor* successfully stops the Firefox COOP exploit.
3) *Exploit on Chrome:* There are two reasons why *TypeArmor* successfully stop exploit on Chrome. One is that we find that three consecutive gadges use rsi to pass data. However, the second indirect call prepares only one argument, which means that *TypeArmor*'s CFC enforcement scrambles data stored in rsi and thus stops the exploit.
   Another is the first indirect callsite is non-void, but it tries to target a void function, which is not allowed with *TypeArmor* in place.

3) Control Jujutsu
   The two Control Jujutsu exploits combine data and control-flow diversion attacks: the authors assume a restricted memory write to prepare a certain state, followed by overwriting a function pointer. The new function pointer still targets a function entry, but one that can use the prepared state to give the attacker control over the program.
   With *TypeArmor* in place, first, the attack against Nginx diverts a non-void callsite to target a void function will be not allowed. Second, the attack against Apache HTTPD diverts a callsite to invoke a target function that does not have its address taken, which is also what *TypeArmor* does not allow.

4) COOP possible Extensions
   As we are already armed with the knowledge that there are three ways for data flow in COOP: using unused registers, using overlapping counterfeit object fields or global variables as well as using arguments actually passed to the callee. According to the COOP paper, *implicit* data flow is always key to successful exploitation: in many cases, main-loop gadgets and recursive gadgets prepare only few arguments for the callsite, leaving the attacker with many registers she can use for date flow. On the other hand, *explicit* data flow is characterized by enabling data flow using *actual* arguments to the vfgadget.
   TypeArmor does effectively prevent implicit data flow. However, if *TypeArmor* fails to determine the exact argument count a callsite prepares, an attacker might be able to use the discrepancy to enable data flow, the attack surface is limited drastically.

5) Pure Data-only Attacks
   The Control-Flow Bending (CFB) paper evaluates the general effectiveness of ideal CFI solutions and evidences their limitations against sophisticated

CFG-aware attacks. As any other CFI solution, *TypeArmor* cannot stop pure data-only attacks. Through author's communication with the author of CFB, author says in paper:"the CFB authors shared their exploit notes for the presented Apache and Wireshark attacks; two attacks that work even in the presence of a runtime shadow stack and ultimately overwrite a function pointer at some point during the exploit".Obviously, *TypeArmor* as any other CFI solution, it cannot stop pure data-only attacks.

6 Performance and security analysis

TypeArmor is implemented on Linux for x86_64. In order to evaluate the impact of *TypeArmor*s instrumentation on runtime performance, they measured the time to complete the execution of the benchmarks and compared against the baseline. The baseline refers to the original version of the benchmark with no binary instrumentation applied. The result as published is that conguration introduces a noticeable performance impact (7.6% on average, geometric mean), owing to about half of the applications executing millions of indirect callsites per second.
In all, " *TypeArmor* imposes a relatively low runtime performance impact on all the test programs considered", as the author puts it. And we are safe to say that this lightweight instrumentation is successful in producing a runtime overhead that is comparable to, or even faster than existing binary rewriting-based CFI solutions.
The security analysis covers the following concerns:(i) callee and (ii) callsite analysis, (iii) the median number of legal indirect callsite targets as enforced by existing (binary-level) address-taken-based solutions and *TypeArmor*s policies. The static analysis results turns out to be very accurate in identifying the exact number of used arguments (79% for callsites and 83% for callees, respectively, geometric mean).

7 Other binary-level solutions

By learning other binary-level solutions which are published, there is an automated method that is published to identify virtual function call sites in C++ binary applications based on an intermediate language and backward slicing, which enables us to determine the potential attack surface for use-after-free vulnerabilities in binary executables implemented in C++. [3]
It presents a generic binary rewriting framework for PE executables with low overhead called PeBouncer that we utilize to implement integrity policies for virtual call sites.
It is worth noting that it is the first to present virtual table integrity protection for binary C++ code without the need for source code, debugging symbols, or runtime type information. Furthermore, it shows that towards vtable integrity

protection (T-VIP) protects against sophisticated, real-world use-after-free remote code execution exploits launched against web browsers, including zero-day exploits against Microsoft Internet Explorer and Mozilla Firefox. A performance evaluation against GCCs virtual table verification feature with micro- and macro-benchmarks demonstrates that our approach introduces a comparable performance overhead.

8 Source-level solutions

With the access to source code, there is no doubt that solutions at source level will be more precise and more reliable. Here I summarize two main source-level solutions compared to solutions at binary level. I will first start with SAFEDISPATCH, and then describe another source-level solution,

1) SAFEDISPATCH
   SAFEDISPATCH[4] addresses the growing threat of vtable hijacking, an enhanced C++ compiler that prevents such attacks. SAFEDISPATCH first performs a static class hierarchy analysis (CHA) to determine, for each class c in the program, the set of valid method implementations that may be invoked by an object of static type c. It uses this information to instrument the program with dynamic checks, ensuring that all method calls invoke a valid method implementation according to C++ dynamic dispatch rules at runtime. By carefully optimizing these checks, it is likely to reduce runtime overhead to just 2.1% and memory overhead to just 7.5% in the first vtable-safe version of the Google Chromium browser which we built with the SAFEDISPATCH compiler.
   In summary, this solution makes the following contributions:
   (a) It is a comprehensive defense against vtable hijacking attacks. We detail the static analysis and compilation techniques to efficiently ensure control flow integrity through virtual method calls.
   (b) The detail of implementation of SAFEDISPATCH as an enhanced C++ compiler is already applied to the entire Google Chromium web browser code base to evaluate the effectiveness and efficiency of this approach.
2) Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM
   This paper [5] presents implementations of two mechanisms that provide forward-edge CFI protection, one in LLVM and one in GCC. We also provide a dynamic CFI analysis tool for LLVM which can help find forward edge control-flow vulnerabilities. These CFI implementations are fully integrated into their respective compilers and were developed in collaboration with their open source communities. They do not restrict compiler optimizations, operation modes, or features, such as Position Independent Code (PIC) or C++ exceptions. Nor do they restrict the execution environment of their output binaries, such as its use of dynamically-loaded libraries or Address Space Layout Randomization (ASLR). The main contributions of this paper can be concluded in three:

(a) It is the first CFI implementations that are fully integrated into production compilers without restrictions or simplifying assumptions.
(b) It shows that our CFI enforcement is practical and highly efficient by applying it to standard benchmarks and the Chromium web browser.
(c) It resolves the major challenges in the development of a real-world CFI implementation that is compatible with common software engineering practices.

Constraining dynamic control transfers is a common technique for mitigating software vulnerabilities. This defense has been widely and successfully used to protect return addresses and stack data; hence, current attacks instead typically corrupt vtable and function pointers to subvert a forward edge (an indirect jump or call) in the control-flow graph. Forward edges can be protected using Control-Flow Integrity (CFI) but, to date, CFI implementations have been research prototypes, based on impractical assumptions or ad hoc, heuristic techniques. To be widely adoptable, CFI mechanisms must be integrated into production compilers and be compatible with software-engineering aspects such as incremental compilation and dynamic libraries.

9 Conclusion

In this report, I describe *TypeArmor* in detail according to the paper. In general, *TypeArmor* relies on binary-level static analysis to derive both target-oriented and callsite-oriented control-flow invariants and efficiently apply security policies at runtime, and thus stop code-reuse attacks bt disallowing calls between incompatible types.
In addition, TypeArmor relies on callsite-oriented invariants to invalidate illegal function arguments at each callsite and contain attacks that rely on type-unsafe function argument reuse, using a protection technique dubbed Control-Flow Containment. CFC further improves the quality of our target-oriented invariants, resulting in the strictest binary-level CFI solution to date.
The author of COOP paper questioned whether it is even likely to mitigate advanced code-reuse attacks at binary level, apparently, *TypeArmor* contrasts this doubt. According to testing, *TypeArmor* is able to stop all published COOP exploits.

References

1. V. van der Veen et al. *A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level.* In *S&P*, 2015
2. F. Schuster et al. *Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications,* In *S&P*, 2015
3. R. Gawlik, T. Holz. *Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs* Anual Computer Security Applications Conference (ACSAC), 2014.

4. D. Jang, Z. Tatlock, S. Lerner. *SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks* Symposium on Network and Distributed System Security (NDSS), 2014.
5. C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, G. Pike. *Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM* USENIX Security Symposium, 2014