A CFI Countermeasure Against GOT Overwrite Attacks

SEUNGHOON JEONG¹, JAEJOON HWANG², HYUKJIN KWON³, AND DONGKYOO SHIN¹

¹Department of Computer Engineering, Sejong University, Seoul 05006, South Korea
²Hanwha Systems, Seongnam 13524, South Korea
³Information Planning Bureau, Korea Ministry of National Defense, Seoul 04383, South Korea

Corresponding author: Dongkyoo Shin (shindk@sejong.ac.kr)

This work was supported by the National Research Foundation of Korea through the Basic Science Research Program, Ministry of Education, under Grant 2018R1D1A1B07047395.

ABSTRACT In the Unix-like system, the Global Offset Table (GOT) overwrite attack is a long-lasting control flow hijacking attack. The attack, by leveraging the dynamic symbol binding mechanism, overwrites any GOT entry into the attacker’s target address to take the execution flow on the library function call. Recently, Full Relro (Relocation Read only), which arranges the GOT section as read-only at program startup, is regarded as most useful against the threat. However, it entails nontrivial loading overhead and is not applicable to libraries. Furthermore, many software packages are currently distributed without Full Relro. As a result, programs are still exposed to the risk of GOT attacks. In this paper, we propose a CFI-based protection scheme against the GOT overwrite attack. Using dynamically bound function symbols as branch identifiers, the scheme secures inter-module function calls on PLT (Procedure Linkage Table) effectively with little performance overhead. Our LLVM based implementation and evaluation on binutils-gdb show that the branch protection scheme is difficult to bypass, fast, and compatible with existing library programs.

INDEX TERMS Control flow integrity, GOT/PLT, linker and loader, security.

I. INTRODUCTION

Control flow hijacking is the primary goal of software vulnerability attacks. It takes control of a program by changing the execution flow of the program to an intended program code address. The Global Offset Table (GOT) overwriting attack [1] is a traditional control flow exploitation technique for exploiting software privileges in a Unix-like system environment. It uses a dynamic binding mechanism of the Executable and Linkable Format (ELF) program. Each ELF program has a branch table named PLT (Procedure Linkage Table) for library function calls, and the branch instruction of the entry references its own GOT entry value that the dynamic linker finds and binds to when the library function is called. The GOT attack overwrites this GOT entry into the attacker’s branch target address and takes control flow when the library function is called in the program.

Several techniques for defending against GOT overwrites have been proposed, and Relro (RELocation Read Only) is now regarded as the most effective one. Relro blocks the runtime GOT tampering attacks by setting the ELF data section used by the dynamic linker to be read-only when loading the program. There are two types of Relro techniques: Partial Relro, which protects .ctors, .dtors, and .dynamic sections except the .got.plt (hereafter GOT) section, and Full Relro, which protects the GOT section as well. In a program with Full Relro applied, the GOT becomes read-only after all library function calls are bound at loading time and GOT modification is blocked at runtime. Table 1 is an excerpt from the security features of the latest versions of Ubuntu. Starting from 18.04 LTS, Full Relro was applied by default to the X86-64 (amd64) system through the gcc compiler tool.

However, Full Relro involves the performance problem of startup delay. Dynamic binding includes time-consuming tasks such as searching dependent library lists and comparing strings in library function names [2]. In a program with multiple libraries, the loading time by symbol binding may be slower in proportion to the square of the library number when a large number of calls to library functions placed later in the library list occur [3]. For libraries, large numbers of functions can be included, which are not called in normal execution flows, and without lazy binding cascading loading delays can occur due to library dependency. Therefore, Full Relro is not generally applied to libraries. Moreover, in the legacy systems, programs are built and distributed with Partial Relro, and some compilers, including the Low Level Virtual Machine (LLVM) [4], still default to Partial Relro.
Thus, many programs in the execution environment are still exposed to GOT attacks.

As the code reuse attack [5] has become the mainstream of the control flow exploitation attack, the control flow integrity (CFI) scheme has been actively studied as an effective method for defending against the code reuse attack. A CFI applied operating system was released [6], and some compilers are equipped with CFI functionality [7]. Since the CFI method was first proposed, CFI related studies for safe inter-module control transfers [8], [9] have also been conducted. However, defense techniques that assume weak attack models are easily bypassed, and those requiring real-time code conversion are not used for performance and compatibility issues. Even in the inter-module CFI scheme provided by recent LLVM versions, protection for PLT/GOT calls depends on Full Relro, so control redirection by PLT/GOT is currently out of the CFI protection scope.

In this paper, we propose an inter-module control flow protection scheme on PLT using the Control Flow Integrity (CFI) technique. The function symbol dynamically bound between the calling program and the library is used as the branch identifier. The calling program includes instructions on PLT to check the branch identifier, and the library includes a jump table which contains the branch identifiers of the called functions. A proof of concept tool was implemented based on LLVM 10 in the Ubuntu 18.04 LTS environment for the X86-64 system. Security, performance, and applicability were evaluated by applying the scheme to binutils_gdb program group [10]. The evaluation shows that the branch protection scheme is difficult to bypass, fast, and compatible with existing libraries, making it highly applicable. The contribution points of this paper are as follows.

- Proposal of CFI-based protection scheme defending against the GOT overwrite attack
- Implementation of the scheme as a compiler tool on the X86-64 based system
- Evaluation of the scheme on the binutils_gdb program group

# II. BACKGROUND

The Control Flow Integrity (CFI) [11] technique attempts to limit the branches outside the normal execution flow at runtime. While defense schemes against data-oriented attacks [12]–[14] focus on protecting non-control data affecting a program’s benign behavior, the CFI schemes seek to block abnormal control redirection by breached control data. The protected branches are indirect function calls, indirect jumps, and function returns [11], [15]. In the case of direct call and direct jump statements, the offset value between the call origin and the target branch point is fixed at compile time and generated by direct branch instruction that does not require a memory pointer dereference. Since the Data Execution Prevention (DEP) [16] protects code pages from tampering at runtime, the CFI technique tries to protect indirect branches.

The CFI technique consists of an analysis step of generating a normal control flow graph and a step of inserting the branch validation code such that the runtime control flow conforms to the constructed control flow graph. In the analysis step, the indirect branch instructions and the branch target group for each branch source is identified to generate the control flow graph. The branch validation point is the branch origin where the aforementioned indirect call occurs. In general, the accuracy of static analysis is limited and CFI conservatively creates branch target groups for each branch origin. Thus, control flow graphs are generally over approximated. It is known that as the size of the control flow branch group increases, the reachable branch target points increase, thereby decreasing CFI security [15]. In the code insertion step, the branch validation instructions are inserted according to the generated control flow graph. Validation methods may vary, but most CFI schemes use the comparison of static branch identifiers at the branch origin. If those match, the branch is allowed. Otherwise, execution is abnormally terminated. Thus, if the branch target address is changed by an attacker at runtime, program execution is interrupted and the attack fails [11], [15]. Branch identifiers are also varied from one CFI implementation to another. Unique identifiers can be generated randomly for each branch, or based on the type information of the called function.

Dynamic library linking is the runtime symbol binding process between a calling program and a library where the static and dynamic linkers cooperate with each other. At module creation, the static linker generates indirect call instructions such that they dereference pointers to their respective library functions at runtime. The dynamic linker fills the pointer with the actual address of the library function loaded into memory at runtime. The calling program can be a library that calls another library which it depends on. In order to support dynamic library linking, the Unix-like static linker creates two structures, PLT and GOT, including them as sections of the calling program’s ELF file (Figure 1). PLT is a function call table that connects a calling program with a library and is included by the static linker in the executable or the library module. The static linker creates a PLT entry for

| Ubuntu Version | Built with Relro | Built with BIND_NOW |
|----------------|------------------|---------------------|
| 16.04 LTS (Xenial Xerus) | Gcc patch | gcc patch (s390x), package list for others |
| 18.04 LTS (Bionic Beaver) | Gcc patch | gcc patch (amd64, ppc64le), s390x, package list for others |
| 19.04 (Disco Dingo) | Gcc patch | gcc patch (amd64, ppc64le), s390x, package list for others |
| 19.10 (Eoan) | Gcc patch | gcc patch (amd64, ppc64le), s390x, package list for others |
After the GOT overwrite attack was relatively easy. With the introduction of the ASLR and especially the Full Relro technique, GOT overwrite attacks are no longer considered as practical as they used to be. However, GOT overwrite attacks are still threatening. Advanced code derandomization techniques facilitate the identification of GOT addresses, and it should be noted that Full Relro is not applicable to the library. Therefore, it is necessary to review the threat of the GOT overwrite attack in the current system environment and seek to complement the defenses against it.

III. RELATED WORK

This chapter addresses some seminal researches on CFI. Since the CFI technique was first introduced by Abadi et al. [11], many early CFI techniques focused on the performance improvement and application of the CFI in a practical manner [8], [9], [17], [18]. Later, security issues on the CFI techniques, also called coarse-grained CFIs, have been raised by several studies [19]–[21]. Subsequently, fine-grained CFI techniques [22]–[24] were studied but proven to be breached by the limitation of CFG precision and intelligent attack techniques [25]–[27]. However, researches are ongoing to improve the security and performance of CFI and make it practical. In particular, studies to achieve performance improvement of CFI through hardware support deserve attention [28]–[34]. For the comprehensive understanding on CFI, Burow et al. [15] compare the precision, security, and performance of various CFI implementations. The following are the main proposed studies to defend against GOT overwrite attack and CFI related works.

A. GOT HIDING TECHNIQUE

In order to protect against GOT tampering attacks, a technique of hiding the GOT section and the location of each entry was proposed [35]. Under the basic ASLR scheme, the GOT section is located at the beginning of the data segment and at a fixed offset from the code area, so an attacker can locate the GOT entry through a runtime information leak technique. To make the GOT difficult to locate, SecGOT [35] randomizes the relative offset of the GOT section from other sections at startup and rearranges the order of the GOT entries. The address of the GOT entry paired with each PLT entry is dynamically determined. The scheme makes it difficult for attackers to predict the address of the GOT entry, which provides a defense against the GOT attacks. However, during the loading process, additional time is spent to relocate the GOT section with its entries, and the instructions dereferencing the GOT entry also need to be corrected, leading to further delayed loading time. Therefore, the scheme is less effective in reducing loading time compared to Full Relro. In terms of security, the attacker can still track the relocated code section and locate the changed GOT table entry address using sophisticated information leak techniques. Therefore, the expected security by hiding the GOT address is not large.

B. EXISTING CFI-BASED INTER-MODULE CONTROL FLOW PROTECTION SCHEMES

Since the original CFI [11] was first introduced, several methods have been proposed to protect the control flow between modules. Bin-CFG [8] proposed a way to modify all indirect branches, including inter-module branches, into direct branches via a branch validation function. The branch check function maintains the reachable target table to check whether the branch target address exists in the table. However, the approach, classified as a coarse-grained control flow integrity, has the problem of allowing branches to other addresses in the table, and has been proved to allow generalized attacks that exploit the weakness [21]. Modular CFG [9] attempts to block abnormal control flow by assigning a branch group ID to each branch origin and managing the mapping table between the reachable address group and the branch group ID. For each dynamic symbol binding, the control flow graph is updated to maintain the validity of the control flow graph. Furthermore, in order to improve the accuracy of branch validation, the control flow branch group also maintains the function type information of the branch target point. However, according to the branch group size,
the scheme is still classified into the coarse-grained category and is vulnerable to sophisticated attacks. Non-negligible performance overhead can also be incurred while updating the control flow graph and branch address group table.

**C. LLVM CROSS-DSO TECHNIQUE**

LLVM has introduced a CFI-based protection scheme called cross-DSO since version 4 [7]. The LLVM cross-DSO extends the CFI Indirect Function Call Check (IFCC) [22] towards inter-module indirect function calls, which was originally applied within a single module. Branch validation is done by a type identity check function implemented in the called library. A type checking function (\_cfi\_check) is called with the function pointer to check and the function’s type identifier (CallSite\_TypeId) as arguments. Accordingly, the type checking function is implemented for each loaded library. In order for the proper type checking function of the library to be called from the call origin, it is necessary to maintain a mapping table between the address of the called function and the address where its library is placed at runtime as in the Bin-CFG scheme. However, this branch validation at branch targets is accompanied by performance overhead and the scheme has yet unresolved functional problems in inter-module calls. The developer notes the problem as an incomplete function: ‘Shared library support-EXPERIMENTAL’ [36]. In addition, LLVM’s cross-DSO only protects indirect calls with function pointers, relying on Full Relro for the protection of function calls by PLT/GOT. Therefore, the scheme does not defend against the GOT overwrite attack if the module is built with Partial Relro.

**IV. ATTACK MODEL AND ASSUMPTIONS**

In the modern operating system environment, launching the GOT overwrite attack is not simple. In the presence of the address space layout randomization (ASLR), attackers have to find the location of GOT entries as well as the address of the target code. Under the current ASLR protection, the GOT overwrite is useful as the step of the control flow hijacking to execute the gadget chains in the attack process rather than a standalone attack; in a code reuse attack, after an attacker constructs a gadget chain using leaked pointers, she needs a way to redirect programs’ flow into the constructed payload [37]. With the help of the gadget search tool, finding PLT and GOT can be facilitated. For example, the code harvest technique [38] is a runtime process for dynamically searching and GOT can be facilitated. For example, the code harvest technique utilizing the GOT entries in the executable. The code harvest technique can be used to obtain a leaked code pointer to derandomize the memory layout, to locate PLT/GOT using advanced ROP attacks such as the code harvest process mentioned above [38]. In case the executable is hardened by Full Relro, the attacker targets at GOT entries in the libraries instead of those in the executable. The GOT entries in dependent libraries are found with the help of the recursive search technique utilizing the GOT entries in the executable. The address to be written to the GOT entry could be that of a full attack function such as sh in libc or the entry address of a gadget chain. In the latter case, it is assumed that the gadget chain for the code reuse attack is already constructed and ready to run. Manipulated library loading by attackers is not assumed because it implies that the attacker has successfully completed control flow hijacking and can already control the execution of the process. Such assumptions are not excessive since existing defense mechanisms assume the attack model where attackers obtain a leaked code pointer and redirect the control flow to the gadget chain using advanced ROP techniques. Moreover, according to the research [37], the code pages are highly connected by code pointers, and
identifier assignment should take the library into account. As mentioned above, for the inter-modules CFI, the branch DYNAMIC BINDING SYMBOLS suggests that the uniqueness of branch target for each branch requires review in terms of security. Recent research [51] the selection of branch identifiers based on function type vulnerable to defending against motivated attacks. Therefore, are performance practical for large programs, they can be loaded.

In the runtime environment, this study proposes the control flow protection scheme against the GOT overwrite attack. Indirect calls using function pointers without PLT or dynamic library loading using dlopen() and dlselect() are not the scope of this study.

V. BRANCH PROTECTION SCHEME WITH CFI

CFI-based branch validation requires branch identifiers that the branch origin and target points share. Care should be taken in the selection of branch identifiers in the inter-module branches. If a random identifier is assigned with respect to the branch origin, the shared identifier in a library function valid in one program address space may not be valid as the identifier in another address space since the library can be shared among multiple programs. This can be a problem because a library needs to be created separately according to the identifiers assigned to each branch origin of the calling programs.

Furthermore, the consideration of the size of the branch target group for each branch identifier is also important. In general, at compile time, the reachable target point is not determined decisively for each call point. The smaller the size of the target group for the branch origin, the higher the security of the CFI. However, due to the inaccuracy of CFG generation, there is a limit to reducing the size of the branch target group for one branch identifier [15]. The weakness of coarse grain CFI techniques that employ large target groups have been addressed in many studies [20], [21], [27]. For the fine-grained CFI scheme, it can ensure an enhanced security level. However, if the scope of the branch extends to the inter-module scope, the security level can be degraded again; branch identifiers that were unique within a single module can conflict in the address space where multi-modules are loaded.

Since the discussion on the appropriate branch group size became widespread, branch validation based on function types have been widely used. The CFI scheme implemented in LLVM is also based on type validation using built-in type checking functions. However, a recent study has reported that CFI techniques based on type checking are not safe [50]. The study shows that in a large code, duplication of function types occurs frequently, enabling attacks to bypass CFI techniques based on the Runtime Type Checking (RTC). Although RTCs are performance practical for large programs, they can be vulnerable to defending against motivated attacks. Therefore, the selection of branch identifiers based on function type requires review in terms of security. Recent research [51] suggests that the uniqueness of branch target for each branch identifier is an important factor for improving CFI security.

A. SHARING STATIC BRANCH IDENTIFIERS USING DYNAMIC BINDING SYMBOLS

As mentioned above, for the inter-modules CFI, the branch identifier assignment should take the library into account.

Dynamic binding function symbols are statically shared identifiers between the calling program and the library. As such, the binding process of the dynamic linker uses the library list and symbol table of the calling program. Different programs that call the same library function have an entry for the same library file in the .dynamic section and an entry for the same function symbol in the .dynsym section (Figure 2). The function symbol can be used as the branch identifier for validation on the PLT entry since the function symbol is the one and only one symbol that the corresponding GOT entry is associated with. Note that unlike other branch origins, the branch target function for each PLT entry is uniquely determined at least in terms of function symbols.

The minimized branch target group size guarantees a high level of security for the branch validation on the PLT. In general, library function symbols have a strong symbol property and are generally unique except for few functions in SymbolVersioning [52]; the static linker causes duplicated symbol errors for multiple input functions of the same symbol. If a call function symbol has a weak linkage property, the function symbol may be duplicated in multiple libraries. In this case, different addresses can be bound to the same function symbol according to the runtime environment of the calling program. In summary, branch identifiers for few functions for symbol versioning and weak functions may generate somewhat larger branch target groups. However, the impact of the function symbols on the security aspects is limited and discussed later in detail in the Evaluation section. On the other hand, a function symbol of an external linkage attribute defined in one module may be defined again as a local linkage attribute in another module. However, those functions of the local linkage are not bound by the dynamic linker and do not belong to the branch target group. Therefore, dynamic binding function symbols have good properties as branch identifiers for branch validation.
Dynamic binding function symbols should be coded to be used as branch identifiers in CFI. Library function symbols are strings of variable length and not suitable for direct use as the branch identifier. In addition, in the C++ language, template declaration enables the same function name of another type. Therefore, the dynamic binding symbol goes through the following encoding process to be used as a branch identifier of the CFI scheme. Symbols are generated as fixed-length bit codes by a hash function that can be loaded into a single instruction code, while the symbol names are preprocessed by the name mangling [53] rule as input to the hash function. Since the PLT is generated by the static linker and uses symbols that have already been mangled by the compiler, no extra mangling process is necessary for branch identifier generation in our scheme.

### B. CHECKING BRANCH IDENTIFIERS IN THE PLT OF THE CALLING PROGRAM

When the CFI technique is extended between modules, the branch validation process may change. Branch validation is commonly performed either by directly comparing the branch identifier of the branch target point with that of the branch origin, or by executing instruction code to check the valid offset range that the branch address should be located within. However, in the inter-module scope, the branch target group may be enlarged due to duplication of the branch identifier. For the library functions, a valid offset range may be dynamically changed, so that a separate validation function may be required. Therefore, the existing inter-module CFI techniques require dynamic table management and dedicated function calls for validation, which causes a lot of performance overhead. In particular, branch validation becomes more complex when branch validation functions must be performed at the library as in the case of LLVM cross-DSO.

In our approach, since almost unique function symbols are used as the branch identifier, branch validation can be performed simply by executing a comparison instruction at the branch origin. Algorithm 1 shows the linker’s pseudo algorithm that adds branch validation instructions to PLT. Note that the validation instructions are added only if the branch target module has a jump table for the branch validation. At runtime, the branch identifiers encoded in the branch origin instruction and the branch target instruction are compared. If those are matched, the control flow goes to the library function. Otherwise, the control flow follows to the dynamic binding procedure and the GOT entry is updated. Dynamic binding uses the Lazy Binding, or Lazy Symbol Resolution, without modification, that is, the entry sequence number of the relocation table is pushed on the stack, and the flow jumps to plt[0] to run the dynamic linker. The two instructions following the je branch instruction in Algorithm 1 are those for the dynamic binding.

Branch validation of PLT entries is simple and effective. The instruction code of the PLT entry was changed from the unconditional branch referencing a GOT entry value to the conditional branch checking the branch identifier. Only three instructions were added. Branch validation would fail if the first call is made or the GOT entry is tampered by the attacker; at the first call, the branch identifier would not exist at the address pointed by the initial value of the GOT entry. In the case, the lazy binding process of the dynamic linker updates the GOT entry with the correct function address. An attacker can manipulate the GOT entries to the desired branch target addresses, but it is difficult for the attacker to bypass the branch validation. Under the DEP protection, the attacker cannot tamper with the branch validation code and the branch identifier. Even if the attacker succeeded to overwrite the GOT entries, branch validation of the PLT entry would fail, and the modified GOT entry would be updated with the correct function address by the dynamic linker. Eventually, the control flow takeover attack by GOT overwrites will end in failure.

### C. CREATING THE JUMP TABLE IN LIBRARIES

Branch validation in PLT entries requires the insertion of the branch identifier into the library. Inserting the branch identifier directly into a library function can result in offset changes in the function, leading to additional relocations of code addresses and compatibility issues. Thus, a separate jump table is created for the branch validation so that existing code sections in the library are not modified. The jump table

```
Algorithm 1 Algorithm for Generating Branch Validation Instructions to PLT by the Linker

Input: GOT, PLT, and a relocation entry index
Output: Assembly instructions of a PLT entry

Function writePlt (got, plt, relIndex):
    symbol ← get the function symbol using plt and relIndex
    module ← get the library module using symbol
    gotOffset ← get the offset value to the GOT entry using got and plt
    targetID ← get MD5 hash value using symbol
    idOffset ← get the constant offset value to the targetID from the branch target
    if module does NOT have a jump table then
        instructions ←
        jmpq *<gotOffset>(%rip)
        pushq <relIndex>
        jmpq <plt[0]>
    else
        instructions ←
        mov <gotOffset>(%rip), %rax
        cmpl <targetID>,<idOffset>(%rax)
        je <offset to the ‘jmpq *%rax’>
        pushq <relIndex>
        jmpq <plt[0]>
        jmpq *%rax
    end if
    return instructions
End Function
```
becomes the external interface of the library and each entry in the jump table becomes the entry point to the library function.

Figure 3 conceptually illustrates the call chain from the calling program to the library. Each entry code of the jump table consists of a direct jump instruction to the associated library function and an instruction encoded with the branch identifier. The entire jump table instruction code is created as a separate function so that there are no additional address relocations in the existing library. The symbol address of the library function is modified to point to the jump table entry so that the address of the jump table entry is bound when the library function is called.

Library modules with jump tables enable efficient branch validation with a small number of instructions. The library has one entry in the jump table per a defined global function, and each entry includes only two instructions. Performance overhead is low because only one direct branch instruction is additionally executed; the instruction encoding the branch identifier is fetched at the branch origin and not executed in the library. We measured the instruction overhead in the evaluation section.

VI. IMPLEMENTATION

Our CFI-based protection scheme was implemented based on the LLVM 10 of the Pass framework [54] and the LLD linker project [55] for the X86-64 architecture. The jump table generation code for the library was implemented based on a Module Pass, built into a Link Time Optimization (LTO) [56] library (LLVMgold.so), and input into the LLD linker as plug-in. The branch validation code of the calling program was implemented by modifying the PLT generation code of the LLD linker.

The LLVM provides the Pass framework for code analysis and transformation in units of Module, Function, and BasicBlock. Our scheme is implemented in the ModulePass stage where input file analysis and code optimization are performed. The implemented Pass lists the defined global functions in the module, constructs a jump table, and creates jump table entries that branch to the beginning of each function. Jump table entries are generated to include branch identifiers using dynamic function symbols. Some global functions in the library may not have any associated symbols if they are called only with function pointers. Since our scheme protects function calls on PLT by the runtime symbol binding process, jump table entries are generated only for global functions that have associated function symbols.

The jump table is constructed in a similar way to the jump table used in RTC-based CFI implementation of LLVM. The symbols of the jump table entry are named as aliases of the defined function symbols and have external linkage attributes. The existing function symbols are suffixed with .cfi and converted to have a local linkage attribute so that any external code that references defined function symbols references to the jump table entries. Unlike the jump table entry of the LLVM CFI, a branch identifier-encoded instruction is additionally placed after the jump instruction to enable branch validation at the origin of the call. The branch identifier is obtained by taking the MD5 hash value from the symbol of the mangled function and converting the upper 4 bytes into the little endian format. This value is placed in the lower 4 bytes of the prefetchnta instruction used to encode the branch identifier. prefetchnta is an 8-byte X86-64 instruction code that preloads the addressed instruction into the cache and has no functional side effects at runtime. Figure 4 shows the example code of the library containing two global functions [57] and the result of dumping the generated jump table by the objdump utility. Each jump table entry consists of 16 bytes, and the three int3 instructions following the prefetchnta instruction are the padding bytes for 16-byte alignment.
A jump table per module is generated at the LTO stage. Since the code transformation stage requires an input module as the LLVM IR bitcode [58] format, in our implementation, the linker’s input module was compiled using the Clang 10 compiler with the −flto option. For the non-bitcode object modules, the code transformation is not performed, so their functions are not included in the jump table of the generated library. Because our scheme protects the dynamic branching on PLT, the LLVM IR format is necessary only for the object module to be linked to the dynamic library, and the LLVM IR bitcode is not required for the module to be linked to the executable or to the static archive.

B. CALLING PROGRAM

Branch validation in the calling program was implemented by modifying the LLD’s code that generates the PLT. For each PLT entry, instructions are generated for validating the PLT. The branch identifier is obtained by taking the MD5 hash value in the same manner as in the jump table validation, the linker adds branch validation code on the PLT entry size increases to 32 bytes with the scheme (see Figure 4). Therefore, the binary file of the calling program can increase in size by \( \text{number of PLT entries} \times 16 \text{ bytes} \). The binutils-gdb version 2.33 was used in the environment of Ubuntu 18.04 LTS on the AMD Ryzen™ 7 3700X CPU.

A. FILE SIZE

In order to verify the effectiveness of our approach, we measured the increased file size for the binutils-gdb program group and evaluated the security, performance, and backward compatibility. The binutils-gdb program group was chosen because it is highly available on Linux and includes a wide variety of programs suitable for evaluation in terms of complexity and scale. The binutils-gdb version 2.33 was used in the environment of Ubuntu 18.04 LTS on the AMD Ryzen™ 7 3700X CPU.

VII. EVALUATION

Changes in the PLT entry code can affect the initialized value of the GOT entry. In the case of Partial Relro, the GOT entry is initialized to point to the address of the \texttt{pushq} instruction that stores the relocation index for the dynamic linker’s use. In our scheme, the PLT code first validates the branch identifier of the target address before dereferencing the GOT entry, and if branch check fails, the flow jumps to the \texttt{pushq} instruction for dynamic linking. Therefore, initializing the GOT entry value is not necessary. Meanwhile, when checking branches in a PLT, the jump table may not exist because the protection scheme was not applied to the library. In this case, unnecessary branch validation can cause performance overhead. Our scheme supports selective branch validation for each PLT entry. The linker can determine whether a library in the compilation environment was built with a jump table or not. If the library supports branch validation, the linker adds branch validation code on the PLT entry of the library function call. In our implementation, for simplicity, the linker generates the branch validation code only for the function whose library includes a specific string in the library path.
The library creates jump table entries for each global function defined. Each jump table entry consists of a 5-byte unconditional branch instruction code, an 8-byte instruction code with the branch identifier encoded, and 3 bytes of padding for 16-byte alignment (see Figure 4). Therefore, the library contains a jump table sized of \( \text{number of global functions} \times 16 \text{ bytes} \), which increases the file size. If the library itself contains other dependent libraries, the file size can increase even more by the PLT branch validation code. Table 2 shows file size change in the dynamic library libinproctrace.so included in the binutils-gdb program group. Since the binutils-gdb program group contains only one dynamic library with the small number of PLTs (53) and defined global functions (6), additional experiments were carried out for large sized libraries. Figure 7 shows the calculated file size increment for the dynamic libraries, which the binutils-gdb program group depends on, based on the number of PLT entries and defined global functions. The actual file size may vary around the page size due to segment boundary alignment. The library module usually contains more global functions than executables, but the size increment is not significant. In particular, for most cases, the larger the original file size is, the slower the increment rate will be. Therefore, the file size increment of the calling program and the library by our scheme is acceptable.

### B. SECURITY

This section analyzes the possibility of attackers disabling or bypassing the scheme. Branch validation compares the 4-byte branch identifier hard-coded at the call origin with the one located 0x9 bytes away from the call target (see cmp1 instruction in Figure 5). If an attacker finds code in the process’s memory space that contains the same value as the branch identifier, and modifies a GOT entry value to the address located at -0x9 bytes, the branch validation succeeds with the control flow diverted to the address. However, given the probability of MD5 hash collision, there are little chances that contiguous 4 bytes equal to the hash value may occur. Furthermore, there is even little possibility that the memory area starting from the -0x9 offset coincidentally consists of a set of executable attack code. In the general DEP environment, the attacker is not able to insert executable code.

Therefore, the security of the scheme depends on the probability that the function of the same symbol as the called library function appears in the address space, and the possibility of exploiting the function as attack code. Local linkage attribute functions are not considered because branch identifiers and jump table entries are not generated for them. Instead, attackers can target global linkage attribute functions with the same symbol name. In fact, some strong and weak function symbols can be duplicated in the address space. In general, the linker triggers a symbol redefinition error for multiple strong symbols from the input object files. Nevertheless, SymbolVersioning [52] enables the multi-version of symbol definitions to coexist in libraries. The mechanism is used to import a library function of a specific version. As mentioned earlier, weak functions can also be duplicated in the address space; they are used for the resolution of an otherwise unresolved reference. However, these two classes of functions are likely to perform similar operations in the group due to the nature of the symbol versioning mechanism and of the weak linkage properties. It is likely that there is only a difference in implementation or optimization, and little difference in execution results. Since the GOT overwrite under the ASLR defense is meaningful as the final step of the code reuse attack to launch a full attack function or redirect to the constructed gadget chain, those functions are not suitable as the exploitable attack code.
TABLE 3. Number of total and unique global function symbols in dependent libraries.

| Programs            | Total | Unique | Duplicated (maximum) | Unique ratio |
|---------------------|-------|--------|----------------------|--------------|
| readelf, sysinfo    | 2153  | 2127   | 26(2)                | 98.79        |
| elfedit             | 2225  | 2199   | 26(2)                | 98.83        |
| addr2line, ar, as-new, bfdtest1, bfdtest2, cxxfilt, ld.gnu, gprof, nm-new, objcopy, objdump, ranlib, size, strings, strip-new | 2162  | 2136   | 26(2)                | 98.80        |
| gdbreplay           | 7211  | 7126   | 85(2)                | 98.82        |
| gdbserver           | 7220  | 7135   | 85(2)                | 98.82        |
| dwp, incremental-dump, ld.gold | 7461  | 7274   | 175(4)               | 97.49        |
| gdb                 | 9151  | 8964   | 175(4)               | 97.96        |

We analyzed the probability of duplication for function symbols and branch identifiers in the binutils-gdb program group. Table 3 shows the number of total and unique global function symbols included in the dependent libraries for each program and the ratio of unique function symbols. Programs with the same dependent library group have the same number of global functions. Note that on average, fewer than 3% of library function symbols are duplicated; the maximum number of duplicated symbols is 4 for pthread_cond_* functions. The high proportion of unique library function symbols indicates that an attacker, under our branch protection scheme, has few GOT entries that he can modify for flow diversion in the address space. Even for the modifiable GOT entries, attackers can only divert to few library functions sharing the same symbol. As mentioned earlier, those are practically of no value to attackers as the target of the GOT overwrite. In case of branch ID collision among different function symbols, we also examined the existence of any duplication of branch identifiers for unique function symbols, but there were no MD5 hash collisions or duplicated branch identifiers.

We also describe the measurement of false positive / negative rates, which are important security indicators. In most CFI implementations, the rates are largely affected by the accuracy of the CFG. However, the branch target of the PLT is determined by the symbol associated with the GOT entry, and our scheme does not need CFG. Thus, in our approach, the rates are affected by the presence and redundancy of the called symbols in the address space. The false positive rate is the rate at which this defense scheme blocks normal branches, which is 0% because the linker causes the undefined symbol error for a library symbol that does not exist. The false negative rate is the rate at which the defense can be bypassed. Since an attacker can call other functions with the same symbol name, assuming all library functions are called at the same frequency, the rate can be calculated as the symbol’s average duplication ratio. Therefore, as for the binutils-gdb program in Table 3, the false negative rate is under 3%, which may look high. However, as mentioned earlier, even if an attacker bypasses branch checking, the called functions have similar operations to the original function and are not suitable for the target of the GOT overwrite attack.

C. PERFORMANCE

We compared Full Relro with our scheme for the loading and execution delay using the binutils-gdb program group. Full Relro is the most widely implemented technique for GOT protection of executables. In comparison with Partial Relro, the .got.plt section is fortified as read-only to block GOT overwrite attacks. Therefore, for executables Full Relro and our scheme share the same goal to protect control flows of library calls on PLT. In terms of performance, Full Relro incurs loading delay due to the batched binding of library function symbols at program start up. Our scheme can get fast loading time by Partial Relro, but there may be a time delay due to dynamic binding overhead and branch validation at runtime.

In order to estimate the startup delay, the shell command 'LD_DEBUG = statistics applicationName' was utilized. With the LD_DEBUG environment variable, you can look into debugging information about the operation of the dynamic linker. With the statistics option set, the shell command displays the relocation statistics of the dynamically linked executable, applicationName, by the dynamic linker ld.so. The statistics output consists of two parts: one is about relocations during startup and the other is about relocations by the lazy bindings and dlopen(). For the loading time delay, two kinds of values were extracted from the output text lines in the first part: ‘number of relocations’ and ‘total startup time in dynamic loader.’ Our script iterated one hundred runs for two groups of binutils-gdb programs, one with the protection scheme applied and the other without, and averaged to reduce the time deviation. In order to eliminate the caching effect, ‘echo 3> /proc/sys/vm/drop_caches’ command line was inserted between the run in the script.

Figure 8 shows the relocation numbers and the startup time in the dynamic loader for the binutils-gdb program group. In the left figure, the number of relocations includes those for relative relocation and the global data symbol as well as for the function symbol. Therefore, the relocation number increment in Full Relro is for the number of relocated function symbols that lead to startup loading delay. As the right figure shows, a small number of the additional function symbol relocations in Full Relro lead to remarkable loading.
time delay. Considering the small size of the binutils-gdb program group, it is expected that the loading time overhead could be worse for the large scale programs.

Meanwhile, for our scheme, there is a time delay caused by lazy binding at runtime and performance overhead due to branch validation. As for the individual dynamic binding for the first call, our scheme incurs two additional instructions overhead (mov and cmpl instructions in the Figure 5). In the subsequent calls after the symbol binding, branch validation and the indirect jump in the library jump table incur four more instructions overhead (mov, cmpl, je, and jmpq). We measured the absolute time delay by the four instructions, which also include the two instructions for the dynamic binding. In order to increase the accuracy of time measurement, we utilized the chrono library [59] included in C++11 STL with the clock type of high_resolution_clock, which provides precision in nanoseconds. The execution time of a for loop running 1 million times for the foo function in Figure 4 was measured except for the first call for dynamic binding. The average for one hundred iterations was $2.44 \times 10^{-3}$ sec, and, hence, $2.44 \times 10^{-9}$ sec per each call, while the average for the non-secured version was $1.98 \times 10^{-9}$ sec; the time difference $4.6 \times 10^{-10}$ sec is less than three clocks for Ryzen™ 7 3700X CPU, and is plausible enough when high floating point and measurement as a unit of functions are considered. Therefore, we can expect that the additional time overhead should be less than $4.6 \times 10^{-10}$ sec for the first call (dynamic binding) and not much larger than that for each subsequent library function call under the similar system environment.

The total spent time for dynamic binding depends on the number of library functions actually called, and may vary depending on the execution environment of the program. It is known that the actual call ratio of the library functions included in the executable program is low [60]. Since dynamic binding is run at most once for each library function and the additional overhead by our defense scheme is one-time low overhead, with our scheme, the lazy binding effect of Partial Relro can be achieved. On the other hand, the same library function can be called repeatedly. In this case, the overhead caused by the branch validation is not one-off and may not be negligible. Nonetheless, the additional time delay by several CPU clocks is not expected to noticeably affect the performance of typical programs given the overall overhead of a function call. However, library call intensive programs can be affected if time delays accumulate. In this situation, hardware aided protection schemes can be effective for better performance [32]–[34].

D. BACKWARD COMPATIBILITY

Depending on the program execution environment, the inter-module call may occur with only one of the two modules applied by our scheme. If the protection scheme is not applied to the calling program, the execution follows the legacy flow without branch validation. However, the branch target changes to the jump table instead of the target function because the symbol table of the library points to the entry address of the jump table. Thus, one direct jump overhead is incurred. On the contrary, if the protection scheme is not applied to the library module, the branch validation fails for each call and dynamic symbol binding is performed. In this case, if the function is called frequently, performance may be degraded. In summary, if our scheme is applied to only one module, there is a concern about performance overhead, but no functional difference in the execution. In terms of performance, it is preferable to apply the protection scheme to the both sides. In case the libraries are unable to be modified for reasons such as in operation, our scheme provides an apparatus for this as mentioned earlier; it is possible to insert the branch validation code selectively for each PLT entry by checking whether or not the library supports our scheme. Therefore, our scheme enables modular build with performance overhead minimized, and it is highly compatible with existing libraries.

E. ROBUSTNESS TO ATTACK

When an attacker tries GOT overwrite, Full Relro terminates the program with a SIGSEGV error due to permission violation on the read-only segment. However, in our scheme, branch validation failures trigger the dynamic binding process. Since the modified GOT entry is corrected to the address
of the library function, the result of the attack does not affect the program execution flow.

VIII. DISCUSSION AND FUTURE WORK

A. EFFECTIVENESS OVER FULL RELRO

The latest version of Linux recommends Full Relro as the default compilation option, which may raise questions on the usefulness of this scheme. However, as the size and complexity of the code grows, the program is increasingly modular. As such, the software trend demands us to consider the proper application of Full Relro. Full Relro entails non-trivial loading time delay for the large programs. When an excessive number of libraries are imported, it is known that a significant performance drop occurs [3]. In addition, many of the library calls in the program code include those that are not called in the normal execution flow as in the case of exception handling. As the program scale grows, the effect of lazy binding is getting bigger. In particular, care should be taken when applying Full Relro to programs whose startup time should be considered.

GOT of the library is another attack target difficult to protect with Full Relro. Applying Full Relro to the library is not very useful due to many functions that are not called at runtime, and the performance overhead in case of chained loading. As a result, libraries are being built with Partial Relro, and the GOT overwrite attack on the library is possible. We already showed the plausibility of exploiting GOT entries of the library with the help of advanced ROP techniques such as the code harvest. In addition, recent studies have addressed the potential for attacks on library modules [2], [61]. Attackers can use the DT_DEBUG entry in the .dynamic section of the executable to get the link_map address of the program, which is the structure for the dynamic linker. Next, they could use the link_map pointer to traverse the library and locate GOT entries. In particular, static library searching order and fixed symbol layout in the library enable this kind of attack to be more plausible.

B. PIE COMPILATION

Position Independent Executables (PIE) is the security technique that makes code reuse attacks difficult by randomly placing memory loading addresses for executables. If a function pointer is used in a source code to reference a library function, the address pointed by the pointer varies depending on the PIE option. If the source file is compiled with the PIE option, the function pointer is bound to the address of the library function; otherwise, it points to the address of the PLT entry of the executable program. In the former case, the call does not go through the PLT, which is not the protection scope of our approach. In the latter case, the call is via PLT and is in the scope of our protection scheme. The PIE option is recommended for security, but it increases indirect branches, and care should be taken when applying it to the programs that are sensitive to runtime performance overhead [62]. Therefore, our protection scheme can provide alternative protection for inter-module calls with function pointers in non-PIE executables. For example, LLVM cross-dso requires the PIE option as mandatory. If not applied, branch validation for a pointer directed to a PLT entry will always fail and the program execution will abort. Our scheme can be a good alternative for inter-module control flow protection in programs where PIE is difficult to be applied.

C. INTER-MODULE CALLS WITH FUNCTION POINTERS

Although not covered by our scheme, function calls between modules can be made without using PLT/GOT. Indirect function calls can be made using function pointers. In addition, library dynamic loading and function calls using the dlopen() and dlsym() API functions do not use PLT/GOT as well. However, our scheme can be extended to the call site where the symbol of the function is uniquely determined. This requires additional static analysis to find the symbol of the called function. The former needs pointer alias analysis to find the referenced function symbol, and the latter can extract the function symbol entered as a parameter of the dlsym(). The branch identifier is generated from the function symbol, and the branch validation code is added where the library function is called. For the extended application of our scheme, only calling programs need to be modified and the modification of libraries is not necessary. We leave the implementation and evaluation of the extended scheme for future studies.

IX. CONCLUSION

In this work, we presented a CFI-based branch protection scheme against the GOT overwrite attack. With the sophisticated attack techniques and from the limitations of Full Relro’s inapplicability to libraries, the GOT overwrite is still threatening despite various defense mechanisms in the system environment to date. Furthermore, Full Relro entails non-trivial loading delay, limiting its application to a program sensitive to startup performance. Using dynamically bound function symbols as branch identifiers, the proposed CFI scheme prevents manipulated control redirection on PLT with the verification of coded identifiers in libraries’ jump table. To the best of our knowledge, this is the first work where the CFI technique is leveraged to defend against the GOT overwrite attack. The implementation on LLVM and the evaluation to the binutils-gdb group show the effectiveness of our scheme over Full Relro. As for future work, we plan to extend the protection scope of our model to include indirect function calls by code pointers and function calls by the library dynamic loading.

REFERENCES

[1] c0ntex. How to Hijack the Global Offset Table With Pointers for Root Shells. Accessed: Jan. 31, 2020. [Online]. Available: http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf
[2] M. Zhang and R. Sekar. 2015. Squeezing the Dynamic Loader for Fun and Profit. Accessed: Jan. 31, 2020. [Online]. Available: http://seclab.cs.sunysb.edu/secure/libs/secslab15-12.pdf
[3] P. Pluzhnikov. Dynamic Linking With Large Number of DSOs Degrades Into Linear Lookup. Accessed: Jan. 31, 2020. [Online]. Available: https://sourceware.org/bugzilla/show_bug.cgi?id=16709
SEUNGHOON JEONG received the B.S. degree in computer science and mathematics from POSTECH, South Korea, in 2005, and the M.S. degree in computer science from the University of California at San Diego, San Diego, CA, USA, in 2008. He is currently pursuing the Ph.D. degree with Sejong University, South Korea. Since 2013, he has been a Researcher with the Agency for Defense Development (ADD), South Korea. His research interests include networked systems and information security.

JAEOON HWANG received the B.S. degree in electrical engineering and the M.S. degree in control and instrumentation engineering from Seoul National University, South Korea, in 1980 and 1982, respectively, and the Ph.D. degree in computer science from KAIST, South Korea, in 2004. From 1988 to 2019, he worked as a Principal Researcher with the Agency for Defense Development. He is currently a Principal Research Fellow with Hanwha Systems. His research interests include cybersecurity and data mining.

HYUKJIN KWON received the B.S., M.S., and Ph.D. degrees in industrial engineering from Sungkyunkwan University, Seoul, South Korea. He was the Director of the Research Planning Division, Korea Institute for Defense Analyses, where he researched information system assessment and information system development, from 1991 to 2017. He is currently the Director General of the Information Planning Bureau, Ministry of National Defense, South Korea. He has authored or coauthored a number of scholarly articles. His research interests include cybersecurity and performance for information systems.

DONGKYOO SHIN received the B.S. degree in computer science from Seoul National University, South Korea, in 1986, the M.S. degree in computer science from the Illinois Institute of Technology, Chicago, IL, USA, in 1992, and the Ph.D. degree in computer science from Texas A&M University, College Station, TX, USA, in 1997. From 1986 to 1991, he worked with the Korea Institute of Defense Analyses, where he developed database application software. From 1997 to 1998, he worked as a Principal Researcher with the Multimedia Research Institute, Hyundai Electronics Company, South Korea. He is currently an Associate Professor with the Department of Computer Engineering, Sejong University, South Korea. His research interests include information security, data mining, and ubiquitous computing.