Practical Control Flow Integrity & Randomization for Binary Executables

Chao Zhang[1], Tao Wei[1,2*], Zhaofeng Chen[1], Lei Duan[1],
László Szekeres[2,3+], Stephen McCamant[2,4+], Dawn Song[2], Wei Zou[1]

[1]Beijing Key Lab of Internet Security Technology
Institute of Computer Science and Technology
Peking University, China

[2]University of California, Berkeley
[3]Stony Brook University
[4]University of Minnesota

***Abstract*—Control Flow Integrity (CFI) provides a strong protection against modern control-flow hijacking attacks. However, performance and compatibility issues limit its adoption.**

We propose a new practical and realistic protection method called CCFIR (Compact Control Flow Integrity and Randomization), which addresses the main barriers to CFI adoption. CCFIR collects all legal targets of indirect control-transfer instructions, puts them into a dedicated "Springboard section" in a random order, and then limits indirect transfers to flow only to them. Using the Springboard section for targets, CCFIR can validate a target more simply and faster than traditional CFI, and provide support for on-site target-randomization as well as better compatibility. Based on these approaches, CCFIR can stop control-flow hijacking attacks including ROP and return-into-libc. Results show that ROP gadgets are all eliminated. We observe that with the wide deployment of ASLR, Windows/x86 PE executables contain enough information in relocation tables which CCFIR can use to find all legal instructions and jump targets reliably, without source code or symbol information.

We evaluate our prototype implementation on common web browsers and the SPEC CPU2000 suite: CCFIR protects large applications such as GCC and Firefox completely automatically, and has low performance overhead of about 3.6%/8.6% (average/max) using SPECint2000. Experiments on real-world exploits also show that CCFIR-hardened versions of IE6, Firefox 3.6 and other applications are protected effectively.

I. Introduction

Many protection mechanisms including DEP (Data Execution Prevention [1]), ASLR (Address Space Layout Randomization [2][3]), GS/SSP (Stack Smashing Protector [4][5]), and SafeSEH (Safe Structured Exception Handling [6]) have gained wide adoption, and they are making it more difficult for attackers to exploit vulnerabilities. These mechanisms can mitigate various standard attacks, but these reactive defenses can often be bypassed by advanced exploitation techniques [7][8]. Attacker countermeasures that originally sounded impossible become easier and easier, and sometimes automatable, over time. A better long term approach is to focus on what we want to protect, and then design protection measures accordingly.

A natural protection against control-flow hijacking attacks is to enforce CFI (Control Flow Integrity [9]): a guarantee that all control-flow transfers in a program will be the ones intended in the original program (i.e., those represented in the compiler's control-flow graph). CFI can stop all control-flow hijacking attacks, including sophisticated ROP exploits (Return Oriented Programming [10][11][12]). CFI provides a guarantee that is strong, and can be easily reasoned about formally; this also makes it useful as a building block for other protections [13][14]. The world would be a much more secure place if every binary was protected with CFI.

Unfortunately, despite its long history (the original paper proposing it was in 2005 [9]), CFI has not seen wide industrial adoption. CFI suffers from a perception of complexity and inefficiency: reported overheads (average/max) have been as high as 7.7%/26.8% [13] and 15%/46% [9]. Many CFI systems require debug information that is not available in COTS applications, and cannot be deployed incrementally because hardened modules cannot inter-operate with unhardened modules.

We propose a new practical and realistic protection method called CCFIR (Compact Control Flow Integrity and Randomization, pronounced "see-see-fur"), which fills most of the gap between existing lightweight protection mechanisms on one hand, and CFI on the other. It introduces low performance overhead, and is compatible with unmodified legacy binaries. These properties address the main barriers to adopting CFI widely.

CCFIR enforces a policy on indirect control transfers that prevents jumps to any but a white-list of locations; it also distinguishes between calls and returns, and prevents unauthorized returns into sensitive functions. These restrictions capture the most important aspects of CFI protection, without requiring difficult and imprecise alias analysis. For efficiency and compatibility, CCFIR performs this enforcement by directing indirect control transfers through a dedicated "Springboard section" that encodes target restrictions via code alignment. The execution time overhead of this checking is low, 3.6%/8.6% (average/max) over SPECint2000. As a further layer of protection, the Springboard section facilitates randomly permuting the allowed jump targets at program startup, further increasing the difficulty of control-flow injection.

We build CCFIR as a purely binary transformation. It does

*Corresponding author. Email: lenx.wei@gmail.com
+Work done while the authors were at UC Berkeley.

not depend on source code or debug information. Instead, it analyzes binary executables based on relocation tables which are available with the wide deployment of ASLR. It can be validated independently and deployed progressively. We have applied it to parts of IE6 and Firefox 3.6 and 5 other applications to stop 10 known vulnerabilities. CCFIR protects binaries as large as the 11MB xul.dll in Firefox completely automatically.

In summary, our CCFIR protection approach has the following key advantages:

- Robust protection: provides strong defense against control-flow hijacking attacks including return-to-libc and ROP. ROP gadgets are eliminated.
- On-site randomization: an additional lightweight layer of protection beyond CFI to frustrate control-flow attacks.
- High performance: low overhead compared to previous CFI implementations, only 3.6%/8.6% (average/max).
- Binary only: no source code or debug symbols required.
- Progressive deployment: protected and unprotected code can inter-operate, without raising false alarms.
- Verifiable: can be verified independently.

The remainder of this paper is organized as follows: We talk about related work in Section II, and then give an overview of our approach in Section III. We describe the design and implementation of our system in Section IV. Section V gives our evaluation of performance and protection. Section VI discusses security topics including remaining possible attacks. Finally Section VII concludes.

II. Related Work

Binary Disassembling and Rewriting. Schwarz et al. [15] cover the disassembly problem in detail, including two standard algorithms and a new combination. Their approach also uses relocation tables, but less extensively. A common challenge for disassembly is mixing of code and data within the code section. Many other approaches [16][17][18] have depended heavily on heuristics which with unjustified assumptions that miss some fraction of code. With the deployment of DEP, compilers are more restricted in the ways they can mix code and data. We believe we are the first to point out that the binaries generated with modern compilers' security-sensitive modes (DEP, ASLR) can be thoroughly analyzed using their relocation tables. Some of the systematic binary rewriting modes we use were previously proposed in systems such as Vulcan [19].

Control Flow Hijacking Attacks and Mitigation. Memory safety enforcement can protect against control-flow hijacking attacks; it also defeats non-control-data attacks [20]. A representative technique is automatic bounds checking [21][22]; however these techniques require recompilation from source, and their performance overheads are too high for practical deployment. Recently proposed techniques, such as SoftBound [23] with CETS [24], also introduce an overhead of 116%. Approaches that enforce data-flow integrity [25] can also stop many kinds of exploits, but cause a 2.5x slowdown. PointGuard [26] use pointer encryption to protect function pointers from tampering. However it causes compatibility issues and has weaknesses [27].

Modern operating systems widely adopt lightweight and efficient protection mechanisms, like DEP [1], ASLR [2], and SafeSEH [6]. However, advanced exploit techniques like return-to-libc and ROP-based exploits [10][11] can defeat these protections. An indication of the power of these techniques is that they can provide attackers a Turing complete language for malicious functionality [10][28].

Some new mitigation techniques have focused on protecting against ROP [29], but this has also spawned newer variants of attacks [12]. More comprehensive ROP protections, such as ROPdefenser [30] and ILR [18] introduce high overhead. IPR [31] uses randomization in basic blocks with minimal overhead, but provides only partial protection.

Control Flow Integrity. Abadi et al. introduced the term CFI [9] and proposed a technique to enforce it. Rather than trying to protect the integrity of function pointers and return addresses, this technique marks the valid targets of these indirect control transfers (i.e. function entry points and landing points for returns) with unique identifiers (IDs), and then inserts ID-checks before each indirect call or return instruction. They propose identifying the set of valid targets (i.e. the points-to set) through a precise control flow graph (CFG) construction and enforcing control flow only to this set for each indirect transfer instruction.

However, a precise CFG construction needs a sophisticated pointer analysis, which is especially difficult without source code or debug symbols. Compatibility is a problem: hardened modules and un-hardened modules cannot inter-operate, preventing incremental deployment which is often needed in real systems. A further challenge is diversity of IDs. The more IDs the code uses, the more restricted jumps are, but any overlapping points-to-sets must be unified to use the same ID. Sharing of jump targets such as library functions can lead to many sites having only one ID.

In the absence of detailed analysis, Abadi et al. suggest that using a single identifier for all sites (a 1-ID approach) or one for calls and another for returns (a 2-ID approach) could still provide substantial protection. Their implementation used a conservative CFG in which any indirect call could target any function whose address is taken. This allows modular transformation of libraries, while still supporting multiple return IDs for directly-called functions. However calls and returns into untransformed code are still prohibited, so this approach does not support incremental deployment. CCFIR implements a 3-ID approach, which extends the 2-ID approach by further separating returns to sensitive and non-sensitive functions. This stops the jumps that would be most useful to attackers, but the three-way separation can be compactly represented in the Springboard section layout without requiring separate ID values and checks.

MoCFI [32] applies CFI to ARM binaries, but due to the imperfection of pointer analysis on binaries, they let statically unresolved calls/jumps transfer to any valid function entry. CFIMon [33] utilizes the Performance Monitoring Unit in modern processors for performance, but it is not reliable in practice because of false negatives and false positives. CPM [34] uses source code analysis to find all possible targets of an indirect call and masks (e.g. bitwise AND) the runtime target with these possible addresses to determine its validity. Since the target information is encoded at the call site statically, shared libraries are not supported. Control-flow locking [35] implements a similar policy and has similar restrictions. HyperSafe [36] provides integrity for supervisor-mode code such as a hypervisor, including a CFI-like technique that replaces jump targets with integer indexes into function-specific tables. This approach can provide finer granularity protection for returns compared to basic CFI, but it can not support modular compilation or dynamic linking. Whole-system overhead was modest for benchmarks dominated by I/O or user-space execution, but HyperSafe's approach would likely be significantly more expensive than CCFIR if applied to CPU-bound user-space applications.

SFI (Software(-based) Fault Isolation [37][38][39]) uses instruction rewriting but provides isolation (sandboxing) rather than hardening, typically allowing jumps anywhere within a sandboxed code region. CFI is also useful as a foundation for enforcing SFI [13], or other higher level policies, such as XFI [14] or write-integrity [40].

III. The CCFIR Approach

The goal of CCFIR is to enforce control-flow integrity in user mode applications by ensuring that the targets of all indirect control transfer instructions are legal. We identify the valid targets in binary modules and rewrite them so that the valid targets can be distinguished from invalid ones efficiently. Then we insert checks before each indirect control transfer instruction to make this distinction. To enforce control-flow integrity fully, all modules have to be rewritten, but this ideal is not always possible. To support incremental deployment, our scheme allows unprotected libraries as well.

A. Assumptions

In this paper, we assume the following properties hold:

- ASLR and a W⊕X protection such as DEP are in use. This usually holds in modern systems. In order to support ASLR, modern compilers generate relocation tables in target binaries, which are used by our binary analysis. With DEP, compilers separate code and data sections and thus ease disassembling. DEP also prohibits attackers from tampering with code (including direct transfer instructions' targets) or executing code in the data section.
- The target executable does not self-modify its code or dynamically generate code. Traditional executables compiled from high level languages always satisfy this. For executables that do not conform to this assumption, our CCFIR scheme is not suitable because static rewriting cannot enforce runtime control flow integrity.
- Limited information disclosure vulnerabilities are available to attackers. If intended functionality or a separate information-disclosure bug allows attackers to read entire memory regions such as the Springboard, or to selectively reveal Springboard stub addresses of their choice, the protection provided by randomization can be negated.

B. Protection Targets of CCFIR

There are several types of control flow transfers in user mode Windows x86 binaries:

- Exceptions. When exceptions occur, the operating system takes control and then transfers to user-defined exception handlers. These handlers are only invoked by the OS and will not be targets of indirect transfers. In addition, these exception handlers are well protected by SafeSEH.
- Direct *jmp/call* and conditional jump (*jo, jz* etc.) instructions. Most *jmp/call* instructions fall into this category. Their targets are fixed in the code, so DEP or W⊕X protection prevents attackers from tampering with them.
- Indirect *jmp/call* instructions, as used for function pointers and virtual method dispatch. Their targets usually are read from memory and may be controlled by attackers.
- All *ret* instructions. Their targets are computed and pushed onto the stack at run-time by corresponding call instructions. Attackers may overwrite the return address on the stack to launch attacks like ROP and return-to-libc.

The first two kinds of transfers are already well protected, so CCFIR protects the last two. As shorthand we refer to an intended target of an indirect call or jump instruction as a **function pointer**, and the target of a return instruction as a **return address**.

C. Identify Indirect Transfer Targets

To protect targets of indirect transfers (i.e., return addresses and function pointers) in binary executables, the binary should be disassembled first. And then, we need to find where all transfer targets are created and used, in order to deploy further protection.

In general, it is challenging to disassemble an x86 PE (Portable Executable format [41]) file correctly, because x86 is a CISC platform with variable-length instructions and a dense encoding. However, we can take advantage of the fact that ASLR and DEP are widely adopted on Windows. These facts result in the following important deductions:

R1. ASLR-protected executables must have relocation tables, because absolute addresses in code must be relocated when loading.

R2. Compilers can freely choose the starting address for a function or a segment.

R3. Programmers get the addresses of functions or instructions only through ways provided by high level

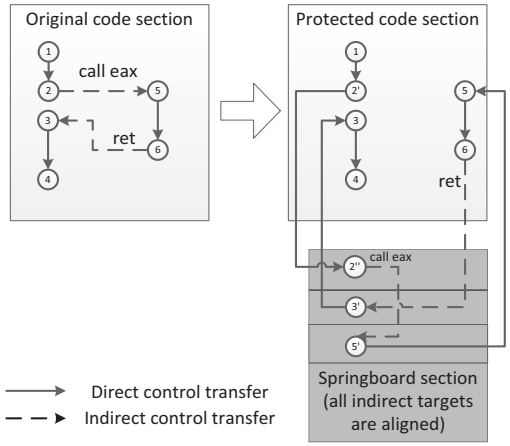

Figure 1: Illustration of CCFIR: a code section is split up into 2 sections, and all indirect control transfers (dashed lines) are only permitted to flow to an aligned address in the Springboard section.

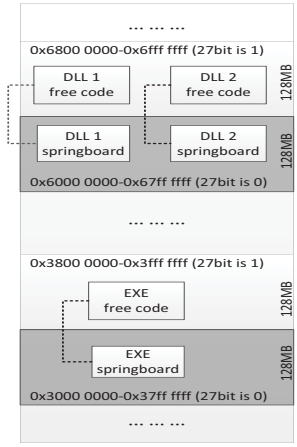

Figure 2: Memory layout for executables hardened by CCFIR: only Springboard sections are placed in a memory area for which the 27th bit is 0.

languages.[1] Programs in high-level languages comply with this rule, and even most inline assembly code does.

R4. If the targets of indirect *call/jmp* instructions are hard-coded in binaries, they must be absolute addresses and can be indexed through relocation tables (as rule R1).

R5. Compilers separate code and data sections (in order to conform to DEP). In code sections, the only data which can appear are special control structures, such as jump tables for switch statements and exception tables.

These rules hold for binaries generated by modern compilers today. Due to the rule R4, we can find most possible indirect code entries. Then with the help of the export table and the `EntryPoint` of the target PE file, it can be disassembled recursively to identify all possible instructions.

Combined with other policies described in Section IV-B, we take an approach that can disassemble a PE file complying with rules R1∼R5 correctly and automatically. For binaries not respecting R5, we can still identify most code and data correctly and tag unidentified parts explicitly for manual review. These remaining parts are usually small even for large binaries, and can be easily reviewed.

As binaries can be disassembled correctly, we can identify where transfer targets are created (i.e., all occurrences of function pointers and return addresses) and where transfer targets are used (i.e., all control-transfer instructions).

D. The Springboard and New Memory Layout

While CFI enforcement techniques have been used to make software fault isolation (SFI) more efficient [13], we conversely use ideas of layout-based checking from SFI [38][39] to make CFI enforcement more efficient.

[1] *getpc()* is a seldom-used method for addressing code and data in normal binaries, although it's more popular in malicious code. In our experiments we find only one case of *getpc* in Windows binaries, *setjmp()* discussed in Section IV-C2.

For each module, we introduce a new code section called the Springboard. As shown in Figure 1, for each valid *indirect* control-transfer target (e.g. nodes 5 and 3 in this figure), the Springboard contains an associated unique stub (nodes $5'$ and $3'$ respectively) containing a *direct* jump to the given target. The nodes $2'$ and $2''$ are used to make sure the node $3'$ is aligned. Using techniques known from SFI, we make sure that *any indirect control-flow transfer instruction can only jump to a code stub inside the Springboard*. As a result, diverting the execution to an attacker-supplied arbitrary target becomes impossible.

The Springboard section is distinguishable from other memory areas through the memory layout. As shown in Figure 2, it is enforced that any executable code section whose address's 27th bit is 0 can only be a Springboard section. In other words we divide the program's virtual memory space into 128MB-large (2^{27}) slices, so that Springboard sections are always in even slices, and other code sections are in odd slices. Data sections are not constrained.

Real-world applications' code sections are typically smaller than 10MB, and they can be placed freely anywhere into an odd 128MB memory slice, as long as the whole section is inside the slice. Multiple Springboards or multiple code sections can be contained in the same 128MB slice but never mixed. Thus one bit testing instruction is capable of checking if an address belongs to a Springboard section.

Make Valid Targets Distinguishable. In order to distinguish valid targets of indirect transfer instructions from invalid targets (e.g. those supplied by attackers), valid targets are all redirected to code stubs in the Springboard. Further, to defeat advanced attacks like ROP and return-to-libc, code stubs within the Springboard are further distinguishable.

First, function pointer stubs and return address stubs are different. Second, return address stubs for return-landing

Executable	Bits				Meaning
	27	26	3	2-0	
no	*	*	*	***	Non-executable section
yes	1	*	*	***	Normal code section
yes	0	*	*	!000	Springboard's invalid entry
yes	0	*	1	000	Springboard's function pointer stub
yes	0	1	0	000	Springboard's sensitive return stub
yes	0	0	0	000	Springboard's normal return stub

Table I: Bit Mask of stubs in Springboard.

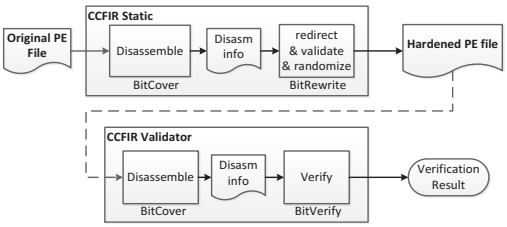

Figure 3: Architecture of CCFIR

points within sensitive library functions (e.g. `system()` in `libc`) are different from return address stubs for normal functions. In other words, there are three kinds of code stubs in the Springboard (i.e., a 3-ID CFI implementation).

As shown in Table I, stubs inside the Springboards are aligned and placed at distinguishable addresses based on their types. Function pointer stubs are 8-byte aligned but not 16-byte aligned. All return address stubs are 16-byte aligned. Further the 26th bits of sensitive return address stubs are 1, while they are 0 for normal return stubs.

With this distinction, the type of indirect transfer targets can be quickly determined at runtime, and a stricter security policy can be enforced on indirect control transfers.

E. Enforcing Control Flow Integrity

Due to the careful design of the Springboard and stubs alignment, one or two bit-testing instructions inserted before an indirect control transfer are capable of validating its target and so enforcing control flow integrity.

Indirect call and jump instructions can only jump to function pointer stubs in the Springboard: In particular, they are enforced to jump to targets within Springboard that are 8-byte aligned but not 16-byte aligned. Moreover, there are no function pointer stubs for sensitive functions in the Springboard because these functions are never used as targets of indirect transfers, as discussed in Section IV-E. With this enforcement, attackers cannot invoke invalid functions or sensitive functions through indirect *call/jmp* instructions.

Return instructions in normal functions can only jump to normal return address stubs in the Springboard, but not sensitive return address stubs: In particular, these return instructions are enforced to jump to 16-byte aligned stubs whose 26th bits are 0. With this enforcement, the capacity of return-to-libc attacks is greatly constrained because exploits cannot jump into sensitive functions. Especially, the Turing-completeness of return-to-libc attacks [28] is broken.

Return instructions in sensitive functions can jump to any return address stubs in the Springboard: In particular, these return instructions are enforced to jump to 16-byte aligned stubs regardless of their 26th bits.

It is worth noting that user programs may invoke sensitive functions, and thus returns within sensitive functions may jump to user functions. On the other hand, we have never observed a need for sensitive functions to call user functions, so CCFIR prohibits returns within user functions from jumping to sensitive functions. While simple, we believe this 3-ID scheme achieves a good balance between compatibility and safety.

In addition, all return instructions' targets are 16-byte aligned stubs in Springboard. And thus, attacks like ROP that jump into the middle of instructions or basic blocks are prohibited. It also greatly raises the bar for exploits which weave together small snippets of code (e.g. gadgets).

CCFIR provides an extra protection to randomize the order of the stubs inside the Springboard at load-time to defeat guessing the addresses of function pointer and return address stubs. Section IV-E will discuss this in detail.

IV. System Design & Implementation

CCFIR consists of three major modules: BitCover, BitRewrite & BitVerify. Its architecture is shown in Figure 3.

The first module BitCover disassembles a given PE file, and identifies all indirect *call/jmp/ret* instructions and all potential indirect control-transfer targets (Section IV-B).

BitRewrite statically rewrites the target PE file. In particular, it inserts Springboard sections for each module, encodes valid transfer targets with pointers to Springboard stubs (Section IV-C1), and instruments runtime checks before indirect transfers to validate the targets (Section IV-C2). BitRewrite also pays much attention to compatibility issues (Section IV-D) to support incremental deployment.

In addition, BitRewrite introduces a further layer of protection, randomization, to increase the difficulty of attacks (Section IV-E).

A separate module BitVerify checks whether a given binary conforms to our defined security policies (Section IV-F). It is the last module before a binary is executed.

A. Background: Relocation Table

The relocation table is a feature of binary code required to support dynamic linking and ASLR, and BitCover also uses it to support disassembly. We use the following terms to describe the structure of a PE-format relocation table:

- **Relocation item:** a 2-byte entry in the relocation table. The lower 12 bits of an item are used together with a page base to compute the address of a `relocation slot`;

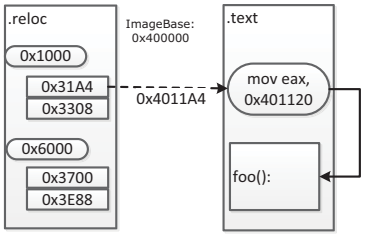

Figure 4: Relocation table

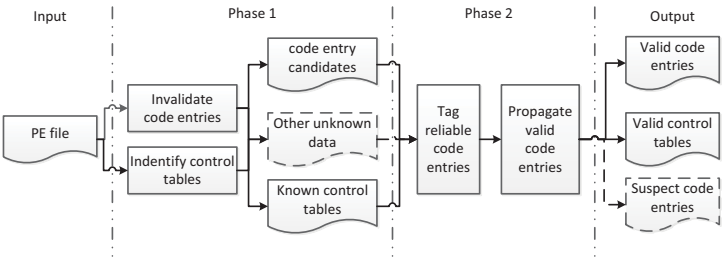

Figure 5: Workflow of BitCover

- **Relocation slot:** a memory area that is to contain a `relocation entry`. For instance on a 32-bit architecture a typical slot will be 4 bytes long.
- **Relocation entry:** the value to be stored in the `relocation slot` at relocation (load) time. This is usually the address of a function or global variable.

For example, in Figure 4, a relocation item 0x31A4 exists in the relocation table (i.e. the `.reloc` section). The highest 4 bits (i.e. 0x3) indicate this is a normal relocation item.

The address of the relocation slot represented by this item is 0x4011A4, which is the sum of the image base of the PE (e.g. 0x400000), the relocatable page's Relative Virtual Address (e.g. 0x1000) and the relocation slot's offset within this page (e.g. 0x1A4, lower 12 bits of the relocation item).

The actual content stored in this relocation slot, i.e. the relocation entry, is 0x401120. This is the address of a function *foo*, which needs to be updated at load time.

B. BitCover

The goal of BitCover is to identify all indirect control transfer instructions and all valid transfer targets. As shown in Figure 5, the workflow of Bitcover consists of two phases.

1) Phase 1: Explore the Code and Data: The EntryPoint and entries in the export table of a PE file are possible code entries. In addition, relocation entries are possible code entries too according to rule R4 in Section III-C. For each possible code entry, BitCover starts disassembling from it recursively. Every executable instruction will be reachable by recursive disassembly from some code entry.

However, not all export entries or relocation entries are real code entries; for instance a relocation entry might represent data rather than code. BitCover uses the following heuristics to determine when disassembly reaches a byte sequence that cannot be a valid instruction. If BitCover encounters an invalid sequence during recursive exploration, it marks the code entry from where it starts disassembling as invalid.

H1. No invalid instruction are permitted.
H2. No instruction overlaps with another.
H3. A valid instruction must lead to other valid instructions.
H4. When disassembling starting from a code entry, all possible paths should stop at return instructions, indirect jump instructions which jump to unknown targets, or instructions which invoke *must-terminate function*s.
H5. All addresses' sizes must be valid.
H6. If an instruction contains a relocation slot, the content of this slot (i.e., a relocation entry) must be a valid immediate value or offset.
H7. Instructions cannot start from relocation slots.
H8. All absolute addresses in code must be relocated, except special hard-coded system values.
H9. I/O instructions and interrupt instructions are permitted only in specific situations.
H10. Only specific segment registers can be used in code.

Here, a *must-terminate function* is one that will never return to its caller, such as `exit` or `abort` in C/C++. During exploration, BitCover also marks any function that calls a must-terminate function unconditionally as a must-terminate function itself. BitCover stops disassembling after a path reaches a call to a must-terminate function, since the bytes after that call would not be executed: they may belong to another function or not be code at all. A function that might call a must-terminate function under some but not all circumstances we call a *may-terminate function*. BitCover also analyzes which function may terminate, and if it encounters an invalid byte sequence after a call to a may-terminate function, it treats that call like a call to a must-terminate function.

In phase 1, we also identify control tables like switch jump tables [42]. We use both instruction and data characteristics to distinguish switch jump tables from instructions. A switch jump table must be an array of relocation slots containing pointers which point to valid code entries. Following H7, BitCover can accurately recognize any switch jump table as data, i.e. not parts of instructions. In fact, heuristic H7 can filter out most control tables, including vtables for C++ objects, but not jump index tables [42] for switch statements, because entries in these tables are not relocation slots.

After this phase, all candidate code entries and some known control tables have been identified. There are still some invalid candidate code entries and some unknown data left. A second phase analysis is needed to remove all those invalid candidate code entries, and identify control tables like switch jump index tables from the unknown data.

2) Phase 2: Refine the Disassembling Result: In this phase, BitCover removes unreachable entries, tags other suspect entries, and identifies remaining control tables.

If a relocation entry does not exist in the export table and is not the target of any direct jump or call (i.e., is not explicitly a function pointer), and all relocation slots containing this relocation entry are parts of some instructions (and thus the relocation entry must be an offset or immediate value according to H6), and this relocation entry will not be assigned to a register (e.g., directly moved to registers), then this relocation entry is called as an ***unreachable*** entry.

For example, if a relocation entry E is only used in instruction `mov eax, E[ebx*4]`, then E is an *unreachable* entry. For *unreachable* entries, there is no chance to transfer their values to any registers or memory; the program cannot use it as a valid indirect target. So, we can claim that:

R6. All *unreachable* entries must not be valid code entries.

Based on rule R6, we can filter out switch jump index tables [42] and other remaining tables in code sections. In addition, after filtering out unreachable entries, the remaining code entries in phase 1 are all candidate entries. If a candidate entry does not point to an entry in a known control table, it must be a valid code entry, according to rule R5 in Section III-C. So, BitCover can disassemble the whole program automatically.

For binaries not obeying R5, BitCover will find unknown data in their code sections. In this case, BitCover tags the location as a "suspected" target, and leave it for manual review. In our experiments, there are limited suspected entries even in big binaries such as `mshtml.dll`, and an expert can tag code entries in them quickly.

C. BitRewrite

The BitRewrite module carries out the central task, instrumenting the binary to enforce control-flow integrity. This is done in two steps:

- All valid indirect control transfer targets, e.g. function pointers and return addresses, are modified by redirecting them to unique stubs located inside the Springboard section. This makes the validity of the addresses verifiable.
- Before each indirect control transfer instruction, e.g. *call/jmp/ret*, a special dynamic check is inserted, which ensures that the transfer target is a valid stub in the Springboard section.

*1) **Redirecting Indirect Control Transfer Targets**:* BitRewrite redirects both indirect *call/jmp* and *ret* instructions' targets, i.e. function pointers and return addresses. As discussed in Section III-E, in order to enforce a better security policy, function pointers and return addresses are redirected to different kinds of stubs in the Springboard. In addition, the ways function pointers and return addresses are created and used are different. So, they are handled differently.

Redirecting function pointers. Function pointers in a compiler-generated binary may be hard-coded in virtual function tables, global variables and instructions. All these occurrences of function pointers can be found, based on the relocation table, as described in Section III.

As shown in Figure 6, as *foo* is loaded into a register and may be a potential target of an indirect call, a unique 8-byte aligned stub *foo_sb* in the Springboard is associated with it. This stub contains a direct jump which will jump to the entry point of *foo*. BitRewrite then replaces *foo* in the instruction `mov ecx, foo` with *foo_sb*.

`Optimizations.` As discussed in Section III-B, function pointers hard-coded in direct *call/jmp* instructions (e.g., *call foo*) and structured exception handlers used by the OS are protected by DEP and SafeSEH and cannot be tampered with by attackers. In addition, these pointers cannot be used by indirect transfer instructions. Thus we can improve performance by no redirecting these pointers, and suffer no loss of security or correctness.

Moreover, function pointers inside jump tables need not be redirected. These pointers cannot be tampered with because of DEP, and can only be targets of jump instructions for switch statements, such as *jmp jtable[eax*4]*. When we confirm that compilers implement jump table lookups correctly, i.e., jumping out of this table is impossible, these pointers can be safely skipped.

Redirecting return addresses. The most frequent indirect control transfer targets are return addresses, which also makes them the most popular targets of attacks. Return addresses are generally pushed onto the stack by corresponding *call* instructions. To redirect valid return addresses, BitRewrite relocates all call instructions.

Figure 7 shows the relocation of a direct call. Like function pointers, a unique 16-byte aligned stub in the Springboard (here *back_sb*) is associated with each call site. A direct jump instruction in this stub will jump back to the original return-landing point (i.e. *back*). To make this stub the new return address, the original call is replaced by a jump to a new call instruction placed right before this stub. This way, when the function is called, the return address pushed onto the stack will be the verifiable address of the stub *back_sb*, which will seamlessly lead back the execution after the original call site.

Figure 6 shows that indirect calls are modified similarly to a direct call. The only difference is because the length of a direct *call* instruction is 5 bytes, while the length of an indirect one is 2. Hence their modified targets are *back_rsb-5* and *back_rsb-2* respectively. Moreover, as discussed in Section III-D, for return-landing points in sensitive functions, the 26th bits of associated return address stubs must be 1, while they are 0 for normal return address stubs.

Having all indirect control transfer targets redirected to their aligned stubs in the Springboard section makes legal targets distinguishable from illegal ones.

*2) **Validating Indirect Control Transfers**:* As discussed earlier, we focus on validating indirect call/jump and return

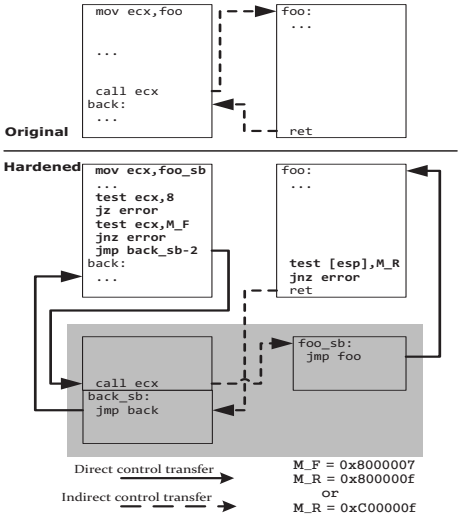

Figure 6: Rewriting of an indirect call and return

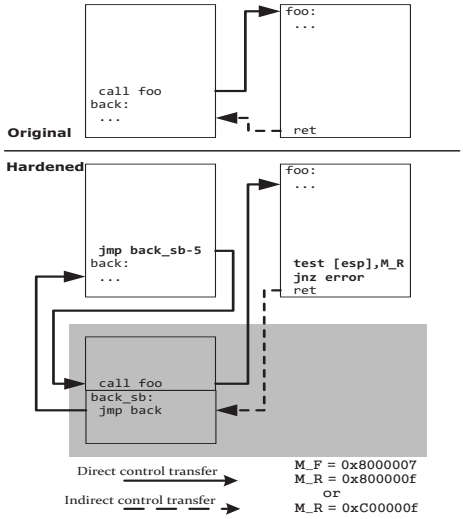

Figure 7: Rewriting of a direct call and return

instructions' targets before the control transfers. The policy our scheme enforces is the following:

- Indirect call/jump instructions' targets must be function pointer stubs (i.e 8-byte aligned but not 16-byte aligned) in the Springboard.
- The target of a return back to a sensitive function can be any valid return stub (i.e. 16-byte aligned).
- Any other return instruction's target must be a valid normal return stub (i.e., 16-byte aligned with the 26th bit 0).

As discussed in Section III-D, this enforcement can be performed using one or two bit-testing instructions.

For any indirect call/jump instruction, its target should be in the Springboard (i.e. the 27th bit is 0) and only 8-bytes aligned (i.e. the 0-2 bits are 0, but the 3rd bit is 1). Thus if the target address is bitwise ANDed with 8, the result should be non-zero. In addition, if the TARGET is bitwise ANDed with the mask 0x8000007 (i.e. `M_F` in Figure 6), the result should be zero. As shown in Figure 6, these bitwise AND operations are performed with the *test* instruction. If one of these conditions is violated, the control flow is directed to a predefined error handler (i.e. `error` in Figure 6). In our implementation, the error handler will log the buggy EIP value and the invalid transfer target, and then terminate the process. (To record the EIP, there is a separate copy of `error` for each indirect call/jump and return.)

Figure 6 also shows how the validation is inserted before return instructions. Before returning, the target of the return is on the top of the stack, pointed to by the *esp* register. The return address is then tested against a mask `M_R`. The mask is 0x800000f for returns from functions called by sensitive functions, and 0xc00000f for all other return instructions.

`An Exceptional Case.` The function *longjmp()* ends with an indirect jump, but its target is a return address saved by a call to *setjmp()*, and so is 16-byte aligned. Thus the check for this special *jmp* instruction matches the check for a return instruction: *test ecx, 0xc00000f*.

`Optimizations.` Indirect jump instructions which are used for switch statements, such as *jmp jtable[eax*4]*, do not need dynamic checks. For any switch statement, regardless of what its control expression is, the control flow in the binary generated by modern compilers (e.g., GCC and VC) is forced to one entry in its jump table. For example, GCC first makes a bound check against *eax* (corresponding to the case value in switch statements). If it exceeds the bound, then *eax* is assigned with a default value (corresponding to the default case). And then, the control flow transfers through *jmp jtable[eax*4]*. In this way, the control flow is always forced to the jump table entries and thus cannot be hijacked by attackers. Thus BitRewrite skips validating these indirect jump instructions, to improve performance.

D. Compatibility Issues

A protected module only allows indirect control transfers whose targets are valid Springboard stubs. But the stubs are not restricted to be within the current module's Springboard section. Stubs within other modules' Springboard sections are also permitted, since their addresses are compatible; they are validated the same way. And thus if every module in a program (i.e. the main program and all DLLs) is rewritten, according to the scheme described in the previous section, the separate modules will be compatible with each other in any combination and the control-flow integrity is enforced.

However, rewriting all modules is not always possible in practice (e.g. system DLLs on Windows 7 cannot be altered). While control transfers from an unprotected module to a protected one cause no problem, if there is an indirect control transfer from the protected module to an unprotected

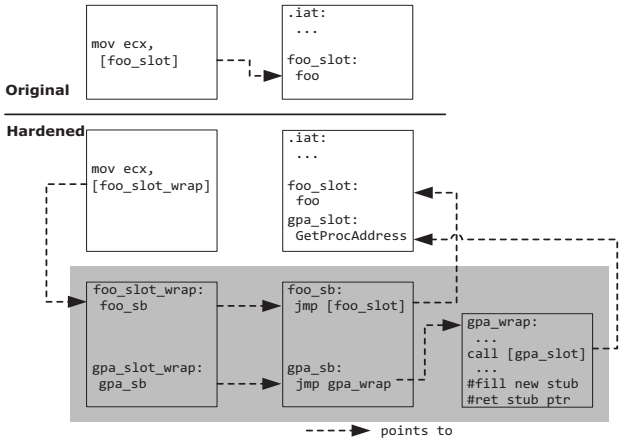

Figure 8: Redirection of imported functions and GPA

one, the check will fail. In order to support the need for incremental deployment, BitRewrite makes special efforts for compatibility.

Compatibility issues come up when a protected module returns to an unprotected module, or calls/jumps to an external function in an unprotected module through

- an imported function pointer,
- a function pointer resolved at runtime by special API, e.g., *GetProcAddress()*, or
- a non-exported function pointer.

1) ***Imported function pointers****:* Most calls to external functions are done through imported function pointers. Imported function pointers are all stored in the import address table (IAT) and then are accessed through IAT entries. For example, in Figure 8, the imported function *foo*'s address is stored in the *.iat* section (i.e., the *foo_slot*). All references to *foo* are accessed through its IAT entry *foo_slot*, e.g., `mov ecx, [foo_slot]`. Imported function pointers will be resolved at load time by the dynamic linker and the IAT entries will be updated.[2] As a result, statically modifying these IAT entries does not work.

To work around this issue, for each IAT entry, BitRewrite generates a read-only and non-executable wrapper to replace it. As shown in Figure 8, for the imported function *foo*, a wrapper *foo_slot_wrap* is generated to replace *foo_slot*. The wrapper stores a function pointer which will jump to the original imported function.

`Optimizations.` Because the IAT is read-only, imported function pointers directly used in *call/jmp* instructions, such as *call/jmp [foo_slot]*, can also skip redirection to improve performance.

2) ***Run-time resolved function pointers****:* Sometimes dynamic libraries are not loaded and linked at load time, but at run-time using *LoadLibrary*. Function pointers in such cases can be obtained by the *GetProcAddress* call (comparable to *dlopen* and *dlsym* in Unix). Since in this case the address of a given function is computed at runtime, if the library is unprotected, the redirection of this function can only be made at runtime as well.

We leave stubs in the Springboard section which can be filled at run-time. For this, write permission has to be given for the page containing the stub, but only for the time of the update. The run-time stub generation is carried out by a special function which wraps *GetProcAddress*. As an exception from the above described redirection technique for imported functions, *GetProcAddress* is redirected to our wrapper function, as depicted in Figure 8. This is possible because the function *GetProcAddress* has to be imported.

The stub code for *GetProcAddress*, i.e. *gpa_sb*, does not jump back to the original *GetProcAddress* function directly, but to the wrapper. The wrapper will call the original *GetProcAddress* function, create a new stub for the returned function pointer in one of the blank slots in the Springboard, and return the pointer to the newly created stub instead of the original function pointer. Only the page containing this stub in the Springboard is writable for the time of this update. This way all function pointers retrieved by *GetProcAddress* are redirected to the Springboard section and 8-byte aligned.

3) ***Non-exported-function pointers****:* The overwhelming majority of external functions are called either through imports or resolved at runtime via *GetProcAddress*, i.e., they are exported by an external module.

However, occasionally an external function that is not exported by any external module can also be called, such as through the *vtable* of an object that exported by an unprotected library. Since this function pointer is never redirected to a Springboard stub anywhere, it will fail the check in the protected module.

4) ***Return to unprotected module****:* It is also possible that a protected function has to return to an unprotected module, e.g. when a function is exported by the protected module and invoked by the unprotected module. When the invocation finishes, the protected function tries to return to unprotected module, and then triggers a false alarm.

We handle these rare cases of 3) and 4) by running BitCover on all libraries which can possibly be loaded by a hardened module, but cannot be protected (e.g. Windows system DLLs). The same algorithm described in Section IV-B is used to collect all valid indirect transfer targets. Instead of using this information for instrumenting the binary, a hash table is built from the valid code pointers. When the error handler is triggered at run-time, in case of a failed check, this hash table is looked up as a final chance to validate the target.

If the target being looked up is not in the hash table, the lookup procedure will terminate the process. Otherwise, the error handler will jump back to the original control flow. To jump back, the error handler for each instrumented

[2]Although the IAT will be updated by the dynamic linker, it is usually read-only and non-executable at runtime.

validation is different. Each error handler saves registers before calling the common hash table lookup procedure, and then restores registers after the lookup and jumps back.

This hash table lookup scheme provides the same level of protection as the previous alignment-based checking. We also emphasize that since the vast majority of unprotected targets are already covered by the first two categories (import tables and *GetProcAddress*), the hash table is seldom used and thus the introduced overhead is negligible. As our experiments show, the hash tables for applications in SPEC2000 [43] and SPEC2006 [44] are never looked up.

When all the involved modules can be rewritten and protected by CCFIR, none of these compatibility features or their overhead are required. However when that is not possible, CCFIR can still be applied to a single module, which can work with other un-hardened modules. This feature allows the incremental deployment of the protection scheme, which we identified as an important requirement of practicality.

E. Security Enforcement and Randomization

BitRewrite enforces that indirect call/jump instructions can only jump to function stubs in the Springboard. Return instructions are constrained to jump to return address stubs in the Springboard, and normal return instructions are prohibited from jumping to sensitive return stubs. Thus it is impossible for an attacker to inject a jump into the middle of an instruction, or to an instruction in the middle of a basic block. This greatly reduces the scope for attack techniques based on stringing together small code snippets such as ROP gadgets. However a buffer overflow could still, for instance, allow an attacker to replace a function pointer with a different legal function pointer, if the attacker guessed its value or caused it to be leaked [45][46][47].

The first countermeasure is to harden sensitive functions. These include:

- `system` and the `execl, execv` family of functions in msvcrt.dll and `WinExec, CreateProcess` in kernel32.dll. They can be used to execute a file or create a process.
- The `LoadLibrary` and `GetProcAddress` functions in kernel32.dll which can retrieve function addresses.
- `memcpy` and other memory operation functions.
- The `VirtualProtect` and `VirtualAlloc` family of functions which can disable memory page protections.
- The `fopen` and `CreateFile` families of functions.
- The `longjmp` function. It is the key to performing branching in Turing-complete return-to-libc attacks [28].
- Other similar functions in application-specific libraries.

We suggest that these sensitive functions should only be used via direct calls, and CCFIR raises an alert if they are called indirectly. Thus we can assume that a binary hardened by CCFIR has no function pointer stubs in the Springboard for sensitive functions, and indirect jumps cannot target them. Similarly, CCFIR's prohibition of normal return instructions from returning into the middle of sensitive functions prevents attackers from accessing parts of their functionality.

The second countermeasure is to introduce randomization. Unlike other recent work [18][31], CCFIR randomizes each stub in the Springboard at load-time. In particular, an extra section is introduced in the PE file to record all redirected stubs' addresses, similar to the relocation table. This section will only be used by the loader, and will not be mapped into the process's address space. So, attackers cannot steal redirected stubs from this extra section.

With this extra section, the loader can reorder those redirected stubs in the Springboard. The stubs are randomly moved to new addresses within the Springboard. And all references to the stubs are updated accordingly. This load-time reordering usually is very fast, as shown in Section V.

In our prototype, this load-time reordering is done by custom bootstrap code planted in the executable. In the future, this could be done using the loader.

The randomization introduced here is an orthogonal layer of protection from the previous CFI-style checking. Even if the randomization is totally disclosed, the original 3-ID CFI still exists. Moreover, the location of each stub in the Springboard is virtually independent. Attackers need a very targeted disclosure (e.g., the stub for *system*) to launch an attack, in contrast to ASLR where attackers can learn the base address of a whole module and reveal all targets.

F. BitVerify

Separate verification provides an independent check of whether the target obeys specified security policies. The CCFIR verifier, BitVerify, checks whether a given binary conforms to the following rules:

- Any executable section whose 27th bit is zero is a Springboard section.
- Code stubs in the Springboard section are all aligned.
- Dynamic checks have been inserted before all indirect *call/jmp/ret* instructions that should not be skipped.
- Function pointers that should not be skipped have all been rewritten and redirected to the Springboard section.
- All *call* instructions have been rewritten to make sure the pushed return address points to the Springboard section.

These rules together guarantee that all indirect *call/jmp* and *ret* instructions in the executable can only flow to valid code entries. Based on the results from BitCover, BitVerify can get all valid code entries and indirect control flow transfer instructions. Also, the Springboard section and stubs in it are all identified. So this validation process is straightforward and fast.

As required, BitVerify can also check extra requirements. For example, we can prohibit sensitive API `VirtualProtect()` from being legal targets, as discussed in Section VI-A.

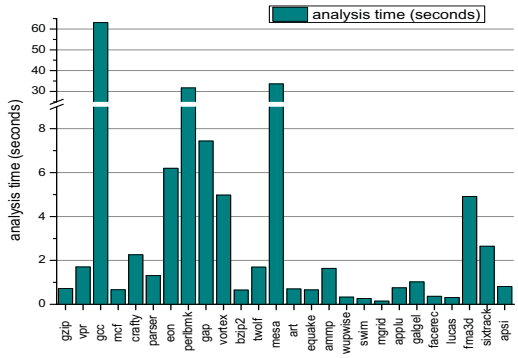

Figure 9: Performance of BitCover and BitRewrite.

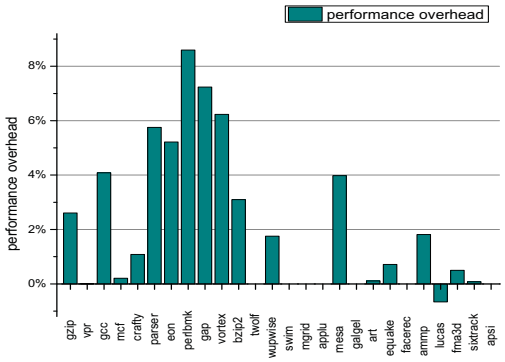

Figure 10: Performance overhead brought by CCFIR.

V. Evaluation

We implement a prototype of CCFIR for x86 PE executables on the Windows platform. In this C++ implementation prototype, BitCover uses an open source disassembler library Udis86 [48] to parse x86 instructions. In addition the 8K LOC of Udis86, BitCover and BitVerify take about 5k LOC, while BitRewrite takes another 5k LOC and an additional custom PE file parser takes 2k LOC.

We test CCFIR with the SPEC CPU2000 (consisting of SPECint2000 and SPECfp2000) benchmark binaries [43], SPECint2006 [44] and several COTS binaries including Firefox 3.6 (denoted as `FF3`) and Internet Explorer 6 (denoted as `IE6`)[3] , to evaluate its overhead and protection.

A. Performance

SPECint2000 consists of 12 applications written in C/C++, while SPECfp2000 consists of 4 applications written in C/C++ and 10 in Fortran. We compile all these 16 C/C++ applications with Microsoft Visual Studio 2010 (abbreviated `MSVC2010`). For the 10 applications written in Fortran we use the GNU Fortran compiler (distributed with the MinGW port of `GCC`). Because CCFIR can also protect the return address, the buffer security check (/GS flag) provided by `MSVC2010` and the stack smashing protection (`-fno-stack-protector`) by `GCC` are turned off. For each application, all modules are statically linked together in order to get the approximate performance overhead of applying CCFIR to the whole system. The experiments are performed on a Windows 7 32-bit system, with an Intel Core2 Duo CPU at 3.00GHz.

Then CCFIR is used to automatically disassemble and rewrite all 26 benchmark binaries. We compare the function pointer information determined by BitCover with the symbol information from the source code, and confirm that BitCover has no false positives or false negatives when parsing the executables. The final binaries rewritten by BitRewrite are then run 9 times. The SPEC harness scripts check that the hardened applications exhibit the same behavior and output as their original counterparts.

For the browser `FF3` and `IE6`, two core modules `xul.dll` and `mshtml.dll` are hardened separately, whereas other modules are left intact, to evaluate incremental deployment. The module `xul.dll` in FF3 is very large (more than 11MB) and has more than 67,000 functions. CCFIR hardens it automatically without any problems. The module `mshtml.dll` in IE6 is also fairly large, 3MB with more than 15,000 functions. While BitCover identified a few hundreds of suspects, an expert can tag code entries quickly. These experiments are conducted in a virtual machine running Windows XP SP3 with 512M memory and 1 core CPU.

Because CCFIR currently does not support dynamically generated code, the JIT (just-in-time compiler) browser option is turned off in the hardened browsers. (To provide a similar protection, the JIT compiler should generate code obeying the same restrictions as CCFIR. In addition, the JIT should protect the generated code from tampering, which is out of the scope of CCFIR.) We check that the hardened browsers work fine and can visit popular websites.

1) Performance of Static Analysis: Figure 9 shows the performance of BitCover and BitRewrite when analyzing the SPEC CPU2000 benchmark. Only three of the benchmark binaries, `gcc`, `perlbmk` and `mesa`, take more than 7 seconds. The other 23 applications take 1.8 seconds on average. The analysis time is positively correlated with the file size (especially the code segment size), the count of function pointers and indirect *call/jmp/ret* instructions. For example, `gcc` is 1,200KB large and takes 63 seconds, while `mgrid` with 70KB takes only 0.14 seconds.

We also evaluate BitVerify's performance. Experimental results show that it can also verify binaries quickly. It takes about 20 seconds to verify the 1.2MB `gcc`, about 37 seconds for the 11MB `xul.dll` in FF3, and less than 10 seconds for other programs in the benchmark suite.

It is worth noting that this static analysis overhead is

[3]For newer browsers, newer OS is needed. But it is difficult to replace modules in newer OS. In addition, there are few public available exploits for newer browsers. So, we chose two old browsers as a benchmark here.

Table II: Statistical data of CCFIR when applying to applications

App.	modifications							performance			#gadgets			file size (KB)	
	redirected fp/ret_addr				validated inst		optimiz.	original run time	new run time	overhead	original	new	valid	original	new
	#fp	#imp	#GPA	#call	#indirect call/jmp	#ret	#skipped fp/import								
SPECint2000										Avrg: 3.6385%					
gzip	77	20	3	976	106	430	171	84.30	86.50	2.6097%	2484	0	0	101	140
vpr	85	20	3	1578	110	768	190	66.00	66.00	0.0001%	4437	0	0	231	301
gcc	1000	26	3	12185	263	5628	612	38.33	39.90	4.0870%	42884	0	0	1181	1642
mcf	73	20	3	896	105	392	173	31.90	31.97	0.2089%	1791	0	0	80	116
crafty	88	23	3	2135	114	930	228	43.03	43.50	1.0845%	7483	0	0	290	368
parser	78	20	3	1600	114	751	173	93.23	98.60	5.7562%	4400	0	0	159	226
eon	1546	28	3	4391	381	2325	316	57.50	60.50	5.2174%	10366	0	0	440	618
perlbmk	924	39	3	7017	203	3229	419	64.47	70.01	8.6002%	30949	0	0	605	829
gap	758	22	3	9991	1352	2672	181	43.30	46.43	7.2363%	20455	0	0	439	639
vortex	164	23	3	3429	124	1715	213	74.33	78.97	6.2334%	13408	0	0	488	648
bzip2	69	20	3	826	103	367	171	68.87	71.00	3.0978%	1824	0	0	91	131
twolf	81	20	3	1385	109	674	183	107.00	107.00	0.0000%	3987	0	0	262	332
SPECfp2000										Avrg: 0.5855%					
wupwise	7	4	0	127	35	31	48	171.00	174.00	1.7544%	255	0	0	67	83
swim	7	4	0	82	30	15	44	347.00	347.00	0.0000%	116	134	0	48	61
mgrid	7	4	0	104	31	21	44	588.00	588.00	0.0000%	161	166	0	49	63
applu	7	4	0	118	29	27	42	484.00	484.00	0.0000%	182	172	0	165	181
mesa	585	22	3	8513	495	3539	345	72.87	75.77	3.9799%	21696	0	0	531	681
galgel	11	4	0	534	62	142	58	265.00	265.00	0.0000%	1515	952	0	281	317
art	73	20	3	895	105	406	174	31.40	31.43	0.1060%	1874	0	0	89	129
equake	69	19	3	862	103	381	154	46.50	46.83	0.7168%	1710	0	0	93	129
facerec	7	4	0	213	55	50	66	239.00	239.00	0.0000%	826	775	0	127	150
ammp	181	20	3	2014	132	901	178	91.87	93.53	1.8143%	5039	0	0	217	279
lucas	7	4	0	98	43	20	55	151.00	150.00	-0.6623%	129	0	0	121	127
fma3d	7	4	0	1341	77	438	72	200.00	201.00	0.5000%	4161	0	0	1429	1633
sixtrack	13	4	0	667	92	208	72	425.00	425.33	0.0786%	3979	3312	0	1463	1618
apsi	7	4	0	372	40	98	54	351.00	351.00	0.0000%	1126	878	0	201	236
Browsers															
mshtml.dll	1,526	139	21	64,662	10,452	15,344	29,557				78,676	0	0	2,995	4,594
xul.dll	145,224	283	34	262,079	55,025	65,359	17,273				273,437	0	0	11,498	15,620

offline and does not influence the runtime performance.

2) Performance of Load-Time Randomization: In our prototype, the load-time randomization is done by bootstrap code placed in the protected executable. Results show that the load time randomization is very fast.

For `mshtml.dll` in *IE6*, there are less than 2^{16} code stubs in the Springboard section, and each stub occupies less than 16 bytes. The whole memory movement when reordering is less than $16 \cdot 2^{16} = 1M$. And the evaluated load time is about 16 milliseconds. Similarly, `xul.dll` has less than 2^{19} stubs and takes about 117 milliseconds.

3) Runtime Overhead on SPEC CPU2000: All the 26 applications in the SPEC CPU2000 benchmark are hardened by CCFIR. Then the median run time over 9 trials is evaluated. Figure 10 shows the performance overhead caused by CCFIR, while Table II shows the detailed run time data.

When protecting targets of all indirect *call/jmp/ret* instructions, CCFIR introduces an overhead of 3.6% on the average over the SPECint2000 benchmark and only 0.59% for SPECfp2000. The largest overhead is 8.6% on `perlbmk`, an interpreter in which every opcode is implemented with an indirect jump. For `lucas`, there is a slight speed-up, maybe due to increased code alignment.

On SPECint2006, the average overhead is about 4.2%. For space reasons, the detailed data are not listed here. Compared with other protections, such as [13][49], CCFIR is capable of protecting all binaries in the SPEC 2000/2006 benchmarks, with a reasonable overhead.

Statistics: Table II also lists the modifications made by CCFIR to the SPEC CPU2000 applications and 2 browsers.

The columns under `redirected fp/ret_addr` in the table represent the count of code entries redirected by CCFIR, including hard-coded function pointers, imported function pointers, pointers returned by *GetProcAddress* and return addresses pushed by *call* instructions. Taking `gcc` as an example, 1000 hard coded function pointers and 26 imported functions are redirected, and *GetProcAddress* is called 3 times. Moreover, there are 12185 *call* instructions in the whole application. All these 13214 (= 1000+26+3+12185) code entries are redirected by CCFIR.

The columns under `validated instructions` record the count of indirect *call/jmp/ret* instructions which are validated by CCFIR. For `gcc`, there are only 263 indirect *call/jmp* instructions and 5628 *ret* instructions. So, targets of 5891 (= 263+5628) instructions are validated by CCFIR.

Performance Analysis: As discussed in Section IV-C, BitRewrite skips redirecting some function pointers and skips instrumenting checks for some indirect jumps. The column under `optimiz.` in Table II counts how many function pointers are skipped. For `gcc`, 612 function point-

ers are skipped, while only 1029 (=1000+26+3) pointers are redirected. So, about 38% function pointers are not redirected and thus the runtime overhead are greatly reduced.

For the original CFI, the attached ID (a potentially slow `prefetchnta` instruction) will always be executed in direct control transfers. But for CCFIR, there are no extra overheads in this case. In addition, the direct control transfers cover most of the control transfers in applications. And thus, CCFIR is much faster than original CFI.

Return instructions play a large part in CCFIR's overall performance. We also repeated our measurements (detailed data omitted) in a mode in which CCFIR protects only indirect *call/jmp* but not *ret* instructions. In this configuration the overhead is 0.79% for SPECint2000, much smaller than the 3.6% overhead when *ret* instructions are also protected.

The 10 Fortran applications in SPECfp2000 have few indirectly used function pointers and imported functions, so the overhead is much smaller than applications written in C/C++, as those in SPECint2000.

4) Runtime Overhead on Real World Browsers.: One core module each of FF3 and IE6 is hardened by CCFIR separately, as described above. We attempted to test each browser against the Sunspider [50] and Google V8 benchmarks [51].

Unfortunately the benchmarks do not support IE6, so we only report results for Firefox (JIT is turned off). The overhead caused by CCFIR was small. When testing with Sunspider, the run time increases from 2130.7ms to 2150.3ms. When testing with the Google V8 benchmark, the score drops from 369 to 361 (larger results are better).

B. Protection Effects

1) Eliminating ROP Gadgets: CCFIR can be used to defend against ROP attacks because it will validate *ret* instructions' targets. Only instructions directly following a call site can be the targets of *ret* instructions. As a result, after applying CCFIR on the target binary, ROP gadgets that do not directly follow call sites are unusable, including any that start from the middle of legal instructions.

To evaluate this protection we count the number of gadgets in our benchmark applications. First, we use the tool Mona [52] to count the gadgets in the original applications and the rewritten applications. As shown in the columns under `#gadgets` in Table II, after hardening, Mona only finds gadgets in 7 out of the 26 applications, but none of these gadgets will pass the validation of CCFIR.

2) Randomization Entropy: CCFIR's load-time randomization makes it hard to guess the address of a target function or a return site, and thus raises the bar for attackers to hijack the control flow, including return-to-libc and ROP attacks. Our Springboard's size is 128MB (i.e. 2^{27}). All code stubs are randomized within the Springboard. Each stub takes less than 16 bytes and is aligned to 8 or 16 bytes.

This degree of randomization makes a brute-force search infeasible. For each stub, there are 2^{23} (=$2^{27}/16$) possible positions after load-time randomization. To chain k target gadgets together, the attacker has to probe $2^{23} \cdot (2^{23}-1) \cdot \cdots \cdot (2^{23}-k+1)$ times in the worst case.

Table III: Real World Exploit Samples Prevented by CCFIR.

ID	App	Vul Type	Vul Module	Protected
CVE-2011-0065	**FF 3**	Use After Free	xul.dll	yes
CVE-2010-0249	**IE 6**	Use After Free	mshtml.dll	yes
CVE-2010-3962	**IE 6**	Use After Free	mshtml.dll	yes
CVE-2011-1260	**IE 6**	Mem. Corrupt	mshtml.dll	yes
CVE-2005-1790	**IE 6**	Mem. Corrupt	mshtml.dll	yes
CVE-2008-0348	coolplayer	Stack Overflow	core exe	yes
CVE-2010-5081	RM-MP3	Stack Overflow	core exe	yes
OSVDB-83362	urlhunter	Stack Overflow	core exe	yes
CVE-2007-1195	XM ftp	Format String	core exe	yes
OSVDB-82798	ComSndFTP	Format String	core exe	yes

3) Protection against Real World Exploits: We also chose 10 publicly available exploits from Metasploit [53] against FF3, IE6 and 5 other applications. These experiments are performed in a virtual machine running Windows XP SP3 within a separate experiment network. Table III shows the 10 vulnerabilities attacked by exploits we used.

Taking CVE-2011-0065 as an example, this vulnerability exists in Firefox 3.x before 3.6.17. It is a use-after-free vulnerability which can cause arbitrary code execution, when exploited by techniques such as heap spray [54].

After hardening the vulnerable module `xul.dll` with CCFIR, we drive Firefox to access the attack URL again, and the error handler added by CCFIR is triggered. The remaining 9 exploits, which target IE6 and other 5 applications, are also prevented by CCFIR in a similar manner.

VI. Discussion

A. Possible Attacks

To attack CCFIR, an attacker may:

(a) forge a valid target.
(b) change memory pages' protection attributes to change instructions directly or to add forged targets.
(c) use a dangerous target that is used by the program.
(d) jump to valid targets or chain them to launch attacks.

For (a), the attacker has to use a page which is writable and executable at the same time. For modern programs protected by DEP, this depends on the attack (b).

Some APIs are inherently dangerous (e.g. `WinExec`), or are dangerous because they can disable page protections (e.g. `VirtualProtect` in the `Virt*` family). These functions are rarely used through indirect calls in regular applications. CCFIR raises an alert if such functions are called indirectly. CCFIR randomizes their entry addresses to make it even harder for attackers to guess or steal them, providing some protection before developers provide a patch.

If a program calls `Virt*` functions directly and only uses constant `flProtect` or `flNewProtect` arguments which do not make the page executable, it will be immune to such attacks as (a) or (b) after being hardened by

CCFIR. If a program calls `Virt*` functions to make a page executable for JIT, attackers still have a chance to utilize these functions, but again they need to penetrate the randomization generated by CCFIR. We still suggest that the program carefully check the arguments before such calls.

For (d), attackers' abilities are greatly constrained by CCFIR. As discussed in Section IV-E, indirect *call/jmp* are enforced to flow to valid and non-sensitive function entry points. In addition, normal return instructions cannot jump into sensitive functions or any function entries. So, the Turing-completeness of return-to-libc attacks is broken. Besides, ROP attacks that to jump to the middle of instructions or basic blocks become impossible, while chaining gadgets also becomes much more difficult.

When CCFIR is only applied to parts of a program, attackers still have a chance to modify pointers flowing to unprotected external modules. In this situation, ASLR and other memory-allocation-based protection methods will provide valuable defense and make attackers spend much more effort to find the vulnerable locations. But of course we still recommend applying CCFIR to the whole system to provide the best protection.

B. Race Condition of Return Address

The code sequence that CCFIR uses to validate a return address has a TOCTTOU (time of check to time of use) race condition in a multi-threaded program. CCFIR checks the value of `[esp]` and then executes *ret* in the next instruction, but the return address is stored in memory in the interim, where it could be modified by another thread.

The race could be avoided by storing the value in a register, but this would have a substantial performance penalty because it would disrupt the CPU's branch prediction. (Modern CPUs use a private shadow stack to predict the targets of return instructions, while other indirect jumps use a less sophisticated prediction mechanism.)

This race condition affects any other return protection scheme that checks the return value in-place, including MSVC's /GS, GCC's SSP, and PittSFIeld [38]. However the time window in the race is extremely small, so in practice the odds of a successful attack will be small. To avoid the possibility of repeated attacks within a process, CCFIR's validation will terminate a process immediately if it detects an illegal return address.

VII. Conclusion

In this paper, we propose a new approach called CCFIR to ensure that indirect control transfers jump only to known targets. It can be used to enforce CFI, which provides a solid base for software protection. It can block various attacks against control transfers, including most ROP attacks.

CCFIR can be applied through binary rewriting on executables generated by modern compilers. Its runtime overhead is low (about 3.6% measured by SPECint2000). CCFIR's techniques can also be used directly in the compilation process to provide protections for software.

Acknowledgments

This research was supported in part by the National Natural Science Foundation of China under the grant No. 61003216 and 61003217; the Chinese NDRC InfoSec Foundation under Grant No.[2010]3044; the NSF grants 0842695, 0831501, CCF-0424422 and CNS-0831298; the ONR grants N000140911081 and N000140710928; an AFOSR grant FA9550-09-1-0539; and a DARPA award HR0011-12-2-005.

References

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

[2] PaX Team, "PaX address space layout randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[3] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *USENIX Security Symposium*, 2003.

[4] 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 attack," in *USENIX Security Symposium*, 1998.

[5] M. Frantzen and M. Shuey, "StackGhost: Hardware facilitated stack protection," in *USENIX Security Symposium*, 2001.

[6] Microsoft Visual Studio 2005, "Image has safe exception handlers," http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.80%29.aspx.

[7] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Symposium on Security and Privacy*, 2004.

[8] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[10] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[11] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[13] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *ACM Conference on Computer and Communications Security (CCS)*, 2011.

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

[15] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Working Conference on Reverse Engineering*, 2002.

[16] M. Prasad and T.-c. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *USENIX Annual Technical Conference*, 2003.

[17] Hex-Rays SA, "IDA Pro: a cross-platform multi-processor disassembler and debugger." http://www.hex-rays.com/products/ida/index.shtml.

[18] J. Hiser, A. Nguyen-tuong, M. Co, M. Hall, and J. W. Davidson, "ILR : Where'd my gadgets go," in *IEEE Symposium on Security and Privacy*, 2012.

[19] A. Edwards, A. Srivastava, and H. Vo, "Vulcan: binary transformation in a distributed environment," Microsoft Research, Tech. Rep. MSR-TR-2001-50, 2001.

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

[21] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2004.

[22] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *International Conference on Software Engineering (ICSE)*, 2006.

[23] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," in *Conference of Programming Language Design and Implementation (PLDI)*, 2009.

[24] ——, "CETS: compiler enforced temporal safety for C," in *International Symposium on Memory Management (ISMM)*, 2010.

[25] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[26] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting pointers from buffer overflow vulnerabilities," in *USENIX Security Symposium*, 2003.

[27] S. Alexander, "Defeating Compiler-level Buffer Overflow Protection," *The USENIX Magazine ;login:*, Jun. 2005.

[28] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[29] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *International Conference on Information Systems Security*, 2009.

[30] L. Davi, A. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[31] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," *IEEE Symposium on Security and Privacy*, 2012.

[32] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[33] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012.

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

[35] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[36] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE Symposium on Security and Privacy*, 2010.

[37] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient software-based fault isolation," in *Symposium on Operating Systems Principles (SOSP)*, 1994.

[38] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *USENIX Security Symposium*, 2006.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy*, 2009.

[40] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *IEEE Symposium on Security and Privacy*, 2008.

[41] MSDN online library, "Microsoft Portable Executable (PE) and Common Object File Format (COFF) Specification," http://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.

[42] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," in *International Workshop on Program Comprehension*, 1999.

[43] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, Jul. 2000.

[44] ——, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sep. 2006.

[45] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *European Workshop on System Security (EUROSEC)*, 2009.

[46] D. Blazakis, "Interpreter exploitation," in *Workshop on Offensive Technologies (WOOT)*, 2010.

[47] F. J. Serna, "The info leak era on software exploitation," in *Blackhat USA*, 2012.

[48] V. Thampi, "Udis86 disassembler library for x86," http://udis86.sourceforge.net/.

[49] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[50] WebKit.org, "SunSpider JavaScript Benchmark," http://www.webkit.org/perf/sunspider/sunspider.html.

[51] Google Inc., "V8 Benchmark Suite - version 6," http://v8.googlecode.com/svn/data/benchmarks/v6/run.html.

[52] Corelan Team, "MONA: a PyCommand plugin for Immunity Debugger," http://redmine.corelan.be/projects/mona, 2012.

[53] Metasploit Open Source Commitment, "Metasploit Penetration Testing Software & Framework," http://metasploit.com.

[54] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with JavaScript," in *Workshop on Offensive Technologies (WOOT)*, 2008.