FPGate: The Last Building Block For A Practical CFI Solution

Tao Wei[1,2], Chao Zhang[2], Zhaofeng Chen[2], Lei Duan[2],

Laszlo Szekeres[1], Stephen McCamant[1], Dawn Song[1]

[1]UC Berkeley, EECS

[2]Beijing Key Laboratory of Internet Security Technology(LiST), Peking University

April 1, 2012

Abstract

We propose and evaluate a new protection mechanism for indirect call and jump instructions in binaries, which we call FPGate. FPGate stops attacks targeting function pointers by limiting indirect transfers to only those targets that are legal in the original program. When deployed together with other existing lightweight protections, FPGate can provide a level of protection comparable to CFI (Control Flow Integrity), stopping almost all control-flow hijacking attacks including ROP. We observe that with the wide deployment of ASLR, Windows/x86 PE executables contain enough information in relocation tables which FPGate can use to find all legal jump targets reliably, without source code or symbol information. FPGate can be applied to a single module at a time, as well as the whole system, and it provides a clearly specified protection scheme so that it can be checked separatley if the whole binary is protected; we provide an example binary with a function pointer vulnerability which shows the protection. We evaluate our prototype implementation on the SPECint2006 suite: FPGate protects applications as large as the 3MB GCC completely automatically, and has an average time overhead below 0.4%.

1 Introduction

Many binary-level protection mechanisms including DEP, ASLR, GS/SSP, and Safe-SEH have gained wide adoption, and they are making it more difficult for attackers to exploit vulnerabilities. But even together these protections have significant gaps through which attack is still possible. In particular the current state of practice does not adequately protect function pointers and other values used in general indirect call and jump instructions. For instance many recent exploits against use-after-free vulnerabilities work by overwriting class vtables to turn benign method calls into jumps to shellcode, perhaps with intermediate use of ROP.

We propose a new protection method called FPGate to limit indirect call and jump instructions to only those targets that are legal in the original program, preventing these function-pointer misuse attacks. We build FPGate as a purely binary transformation. Our system finds all indirect transfer instructions and the set of valid targets based just on information available in a stripped binary, primarily relocation tables used for ASLR. We then rewrite the binary to add checks before each indirect call and jump. The added checks and supporting information require just a few additional bytes per jump and target, typically less than 100 kilobytes even for a large application. The execution time overhead is extreamly low, less than 0.4% on standard benchmarks.

Beyond its abilities on its own, FPGate is a major step towards making strong Control-Flow Integrity (CFI) guarantees available in practice. Currently deployed mechanisms such as ASLR and stack cookies provide what initially sound like broad protections against standard kinds of attack, but because they are designed in a reactive style, they can often be bypassed by variations and more complex approaches. Attacker countermeasures that originally sounded impossible become all too easy, and sometimes even automatable, over time. A better long term approach is to focus on what we want to protect, and then design protection measures accordingly.

Thus the natural protection against control-flow hijacking attacks is Control-Flow Integrity [1]: 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 defeats a broad range of techniques for shellcode injection, including sophis-

ticated return-oriented programming (ROP). 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 kinds of protection [4]. 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), CFI has not seen wide industrial adoption. CFI may suffer from a perception of inefficiency: the original approach had overheads as high as 40% [1], though recent systems are significantly improved. Another limitation is that CFI systems usually require recompiling all of a program from source code. A recent system [3] replaces this requirement with disassembly via IDA Pro, but when used to on Windows/x86 binaries IDA Pro is heuristic and incomplete.

FPGate fills most of the gap between existing lightweight protection mechanisms on one hand, and CFI on the other. Combining FPGate with robust protection of return addresses as we describe in section 5.2 is a sweet spot for security and usability. This approach provides protection for indirect calls and jumps that is almost as strong as classic CFI (omitting sub-classification of indirect targets), and protection for return addresses that is stronger. At the same time, it has low overhead and can be applied directly to a binary.

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

- ◇ Robust protection: prevents misuse of function-pointers and other indirect calls and jumps
- ◇ Low overhead: under 0.4% on SPECint2006
- ◇ Binary only: no source code or debugging symbols required
- ◇ Progressive deployment: protected and unprotected code can inter-operate
- ◇ Clear guarantee: based on what we want to protect, not particular attack mechanisms

The remainder of this paper is organized as follows: We give an overview of our approach in section 2. We describe the design and implementation of our system in section 3. Section 4 gives our evaluation of performance and protection. Section 5 discusses security topics including remaining possible attacks, and proposed approaches for CFI-level protection. Finally section 6 concludes.

2 Approach Overview

The goal of our FPGate protection mechanism is to ensure that indirect call and jump instructions (collectively "indirect transfer instructions") only jump to targets that are known to be legal. Our general approach is to find where function pointers are created and used in the program, replace the original values of function pointers with encoded values, and modify the instructions that use function pointers to decode the values before jumping. Within this general approach there are two key choices: how to find indirect transfer instructions and their potential targets, and what encoding/decoding scheme to use. We make the following choices:

(1) We use binary analysis to identify all indirect call/jmp instructions and their potential targets.

    Section 2.1 explains how we can perform this identification using only information present in binaries.

(2) We replace function pointers with pointers to code stubs which in turn include a jump to the original target.

    Section 2.2 describes this encoding and the corresponding decoding (checking) process.

Existing mechanisms, such as W⊕X, Safe-SEH and GS/SSP, have protected return instructions and exception handlers, but no commonly-used mechanism systematically protects indirect call and jump instructions. Adding protection for indirect calls and jumps is the last major step towards enforcing that a program may only execute along the paths in its CFG, i.e, to enforce CFI.

We describe more details of our approach in the next subsections.

We use the terms "indirect code entry" and "function pointer" interchangeably with "indirect call/jump target" in the following sections.

2.1 Identify indirect call/jump targets

In general, it is challenging to disassemble an x86 PE file correctly, because x86 is a CISC platform. However, we can take advantage of the fact that ASLR is widely adopted in Windows/x86 executables, particularly those whose developers care about security. This results in the following important deduction:

(1) ASLR-protected executables must have relocation tables (see section 3.1 background on relocation tables).

Further, we make just two more assumptions:

(2) Compilers can freely choose a starting address for a function or a segment.

(3) Most programmers get the addresses of functions only through ways provided by high level languages. Programs written in high-level languages and even most inline assembly code complies with this rule. Malicious/encrypted code may violate this rule, which is out of the scope of this work.

These rules hold for most binaries generated by modern compilers today. We call such binaries "***normal***" PE files in this paper.

Based on these rules, plus the fact that all indirect control transfers in x86 must use absolute addresses, we can deduce that:

(4) All indirect code entries must be reachable from relocation tables or export tables.

We make the observation that due to rule (4), we can cover all possible instructions in a normal PE file by disassembling it recursively from all possible indirect code entries.

DEP is also an important security technology adopted widely today. According to the DEP policy, a compiler should respect the following rule:

(5) In DEP-protected executables, application level data should not exist in code sections. Compilers only put instructions and several types of control tables (such as jump tables for switch statements) into code sections.

Combined with other policies described in section 3.2, we will take an approach that can disassemble a *normal* PE file complying with rule (5) correctly and automatically. For binaries not respecting rule (5), 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 we can disassemble a normal PE file correctly, we can easily identify all valid control-transfer instructions, and all of the target instructions they might legally jump to.

2.2 Protecting function pointers

After finding all indirect control-transfer instructions and possible targets, we can replace all indirect call/jump target addresses with *encoded* values. These encoded addresses should satisfy three conditions:

◇ Their legality can be checked quickly;

◇ They can be used to invoke the original functions quickly;

◇ For maximum compatibility, their numeric values should be similar to the numeric values of non-encoded addresses. For instance, since function entry points are typically aligned, encoded address values should be similarly aligned, in case the program checks this.

Many encoding approaches are possible; the Windows API even provides a pair of functions EncodePointer/DecodePointer with a similar purpose. However most encoding methods give an encoded value that cannot be used as a function pointer by unmodified code.

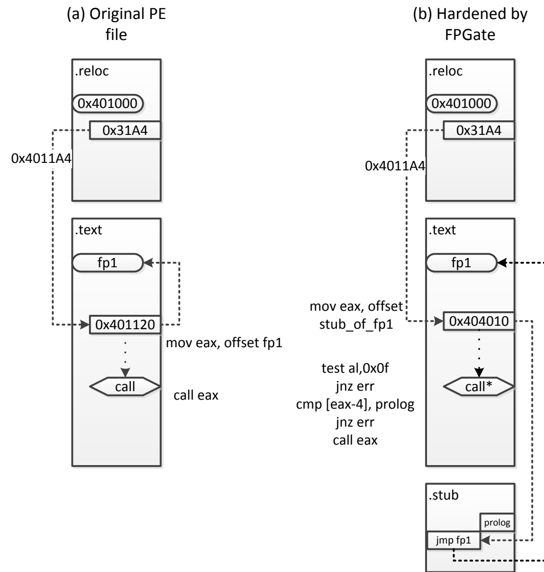

Figure 1: Reloc table & fp encoding mechanism

We propose an encoding scheme similar to the method used to mark legal jump targets in the classic CFI approach [1]. Each possible indirect jump target is represented by a 16-byte-aligned stub function. Valid stub functions are identified by the inclusion of a fixed 4-byte constant sequence we call the "prolog." Following the identifying prolog sequence, the stub has a jump to the original target address. We choose the value of the prolog to be a 4-byte sequence that appears rarely or not at all in the code section otherwise. We can use binary rewriting to ensure that the value is unique.

A key advantage of this approach is that if the stub function is executed directly, as it would be by an unprotected indirect call instruction, it still has the correct behavior. As described in more detail in section 3.3.2, this allows us to apply FPGate to just a subset of the modules in a program.

2.3 From existing protections to C-FI enforcement

On modern x86 platforms, attackers can use only the following four methods to violate control-flow integrity in user space:

(a) Modify instructions directly.

(b) Modify return addresses.

(c) Modify system exception handlers.

(d) Modify indirect call/jmp targets, including function pointers.

There are relative mature mechanisms to protect against the first three attacks.

In order to stop attack (a), W⊕X is a policy most of modern platforms support and obey. It makes sure that every page in a process' address space is either writable or executable, but not both simultaneously.

For attack (b), technologies such as GS cookie protection or Stack Smashing Protector (SSP) are adopted by most modern compilers. When applied only to vulnerable functions, their overheads (based on our benchmarking) are about from 0.14%(gcc) to 1.78%(vc 2010). When applied to all functions, their overheads are about from 2.38%(llvm) to 2.77%(gcc). There are also other technologies proposed for similar purposes but more secure, such as shadow stacks. Their overheads are less than 5% too.

For attack (c), there are methods such as SafeSEH which can make sure that only registered exception handler can be invoked. Most application level exception mechanisms are implemented by indirect jump instructions, so attacks against them belong to type (d).

Only attack (d) has no systematic defense in current systems. There are EncodePointer/DecodePointer APIs provided by the system, but they must be used manually and cannot handle vtables and many other situations.

If we can protect indirect call/jump instructions' targets from arbitrary modifications, we can enforce CFI or quasi-CFI combined with existing protection methods. We have designed FPGate to play this role of protecting against attack (d); we will return to the discussion of modes of combination in section 5.2

3 System Design & Implementation

FPGate includes two major modules: BitCover & BitRewrite. BitCover disassembles a given PE file, and tags all potential indirect control-transfer targets. BitRewrite rewrites the PE file with indirect transfer targets hardening mechanisms. The architecture of FPGate is shown in Fig.2.

For *normal* PE files which respect rule (5) described in section 2.1, FPGate can harden them automatically. For other PE files, FPGate could generate suspect lists for manual review.

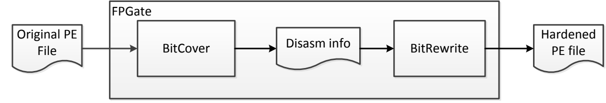

Figure 2: Architecture of FPGate

Relocation table is the important basis for this proposal, but it has some confusing concepts. We clarify these concepts in the first subsection. After that we describe the algorithms of BitCover and BitRewrite.

3.1 Background:Relocation table

Relocation mechanism of the PE format relates to several addresses. We define them as the following:

- ◇ **Reloc item:** exists in the .reloc table. Each item is 2-bytes long, where the highest 4 bits describe the relocation type and the lower 12 bits are used to compute the *reloc slot*;
- ◇ **Reloc slot:** address of the memory which contains a *reloc entry*, must be 4-bytes long in 32bit systems;
- ◇ **Reloc entry:** content which needs to be updated when loaded by the system loader, it is usually the address of functions or global variables etc.

For example, in Fig.1 (a), a reloc item existed in the relocation table (i.e. the .reloc section) is 0x31A4, where, its type is 3 and means this reloc item is a normal one.

The reloc slot represented by this item is 0x4011A4, it is computed through $0x400000 + 0x1000 + 0x1A4$, where, 0x400000 is the image base of the PE file, 0x1000 is the relocatable page's Relative Virtual Address (RVA) and it is stored in the relocation table too, 0x14A is the lower 12 bits of the reloc item and it means the reloc slot's offset in the relocatable page.

The actual content stored in this reloc slot is 0x401120, i.e. the reloc entry, it is the address of a function $fp1$, which will be updated when loading.

3.2 BitCover

The workflow of BitCover is shown in Fig.3. There are two phases in BitCover.

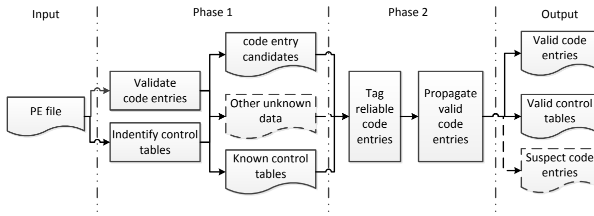

Figure 3: Workflow of BitCover

In the first phase, BitCover checks all possible code entry points to determine if they are valid. It also identifies known control tables based on their special characteristics.

For a given address in the code section (i.e. a possible program counter (PC) value), BitCover uses the following rules to determine whether it is a valid instruction:

◇ No invalid instruction permitted

◇ No instruction overlaps with another

◇ A valid instruction must lead to other valid instructions

◇ The set of code reachable from an entry point should stop at a return, a *terminating function* or an indirect jump to an unknown target

◇ All addresses' size must be valid

◇ If an instruction contains a .reloc pointee, the pointee must be a valid immediate value or offset.

◇ All absolute addresses in code must be relocated, except special system values

◇ IO/interrupt instructions are permitted only in specified situations

◇ Only specified segment registers can be used in code

(A *terminating function* is one that is not expected to return to its caller, such as `exit` or `abort` in C).

Due to rule (4) in section 2.1, all indirect code entries must be in reloc entries or in the export table. For each code entry in these tables, BitCover disassembles from it recursively. If it meets an invalid instruction during the recursive exploration process, it invalidates the entry. During the process, BitCover needs also to propagate information about terminating functions in order to stop properly.

For binaries respecting rule (5) in section 2.1, a candidate entry must be a valid code entry or an entry for known control table. So BitCover can disassemble the whole program automatically.

But for binaries not obeying rule (5), BitCover will find unknown data in their code sections. In this case, BitCover needs to tag such "suspect" parts for manual review. In order to do that, we need to filter suspect parts from reliable parts in phase 2.

We observe a phenomenon that those entries between data and code are most prone to be identified wrongly. At phase 2, we first tag code entries not behind unknown data as valid code entries. Then we tag all its callees as valid. In such a way, we propagate the validity of code entries. Finally we tag code entries left as suspect for manual review.

For the convenience of BitRewrite, BitCover also indentifies padding areas and other entries.

3.3 BitRewrite

In this paper, we call all instructions or data which could generate or represent the address of an indirect call/jmp target *function pointer sources.* We call all indirect call/jmp instructions *function pointer sinks.* As this terminology suggests, function pointer values are introduced into the program's data at function pointer sources. They may then be stored in other variables and data structures until eventually they reach a function pointer sink when they are used to choose the next code to execute.

With information provided by BitCover, BitRewrite tries to encode each potential indirect transfer target at each source, and then decode the function pointers at each sink.

Several encoding and decoding schemes exist. Here, we present a scheme which has good compatibility and security. Each valid function pointer value is represented by a small new piece of code which we call a *trampoline stub.* The trampoline stub is in fact valid code which if executed directly will jump to the original jump target. But the stub also contains an additional identifier which allows the inserted code to verify that it is a legal encoded target.

More specifically, as shown in Fig. 1,

(a) Each function pointer source, such as the instruction *mov eax, offset fp1* in Fig. 1(a), is encoded to *mov eax, offset stub_of_fp1*, where, *stub_of_fp1* is the address of a trampoline stub.

(b) Each function pointer sink, such as the instruction *call eax*, is replaced with a new code block which decodes the function pointer it

uses. In our scheme, the decoding is to validate the transfer target and then call/jump to the transfer target, i.e. the encoded function pointer or the address of a trampoline stub, and then jumps to the original target function.

Each trampoline stub is 16 bytes long and starts at a 16-byte aligned address. At the beginning of the trampoline (offset 0) is an instruction that implements the jump to the original target. Four bytes before the beginning, or equivalently at the end of the previous trampoline (offset −4 or 12), there is a special 4-byte identifier we call the prolog. The prolog is what identifies the trampoline as valid; its value is chosen so that it does not occur elsewhere in the code segment.

In a real executable, there are different types of function pointer sources and sinks. The rest of this section describes in more detail how FPGate handles each kind of source and sink.

3.3.1 Encode function pointer sources

According to the assumption (3) in section 2.1, we can deduce that function pointers sources include such categories:

(a) Hard-coded function pointers in the executable file, such as those stored in the vtable (belonging a C++ classes with virtual methods) and those stored in global variables. The Following function pointers do not need to be encoded:

- ◇ Entries in switch statement jump table. They are usually stored in the code section and cannot be altered by attackers.
- ◇ System exception handlers. They are directly used by the system and protected by Safe-SEH. So they should not be encoded.
- ◇ Function pointers directly used in call/jmp instructions such as *call fp*.

(b) Entries in the import table. These function pointers will be updated when loading. If an import function is only directly used in call/jmp instructions, such as *call [imp_slot]*, it does not need to be encoded too.

(c) Function pointers generated at runtime by *GetProcAddress()*.

(d) Function pointers generated at runtime by *setjmp()*.

In our scheme, each function pointer is replaced with a new pointer, i.e. a trampoline slot, which holds the address of a trampoline stub.

Listing 1: Encode sources: hard-coded fp

```
1 // original:    reloc_slot:fp
2 // Hardened:     reloc_slot:tramp_stub
3 // tramp_stub: 0xE9 4bytes-rel32 (jmp rel32
   )
4 //              7bytes-0xCC 4bytes-
   PrologForNext
```

For function pointers of type (a), as shown in List. 1, each reloc slot which holds a function pointer is filled with the address of a trampoline stub. The corresponding stub jumps to the original function pointer (rel32 is the offset from the stub to the original fp). Note that the last 4 bytes is the prolog for the next stub, which will be checked when decoding.

Listing 2: Encode sources: import fp

```
1 // original:    reloc_slot:imp_slot:imp_fp
2 // Hardened:     reloc_slot:wrap_slot:
   tramp_stub
3 // tramp_stub: 0x25FF 4bytes-imp_slot (jmp
   [imp_slot])
4 //              6bytes-0xCC 4bytes-
   PrologForNext
```

Function pointers of kind (b), they are updated when loading. So they cannot be encoded as for (a), since the encoded value will be overwritten by the loader. Instead, we use the fact that this kind of function pointer must be referenced through the addresses (called import slots) of import entries in the import table.

As shown in List. 2, each reloc slot which holds a import slot is filled with the address (called wrapper slot) of one wrapper entry. The wrapper slot is in a read-only section, its content is the address of one trampoline stub. The corresponding stub jumps to the original import function, e.g. *jmp [imp_slot]*.

Listing 3: Encode sources: return by GetProcAddress()

```
1 // in new_GetProcAddr:
2 //    call original GetProcAddress
3 //    mov original return value to
   preserved ret_slot
4 //    return address of ret_slot's
   associated trmp_stub
5 // tramp_stub: 0x25FF 4bytes-ret_slot (jmp
   [ret_slot])
6 //              6bytes-0xCC 4bytes-
   PrologForNext
7 new_GetProcAddr = gen_or_get_GetProcAddr()
8 // original: reloc_slot:imp_slot:
   imp_GetProcAddress
9 // hardened:  reloc_slot:wrap_slot:
   new_GetProcAddr
10 encode_imp_slot(reloc_slot, imp_slot)
```

For function pointers of kind (c), they are generated at runtime. If FPGate is applied onto the whole system, especially onto the executable itself and all external libraries it depends on, the return value of GetProcAddress() is already an encode function pointer, there is no need to encode it again here.

But, FPGate cannot be adopted by the industry immediately, e.g. kernel dlls in Windows 7 are not allowed to be replaced by user applications. As a result, currently, we cannot assume that the return value of GetProcAddress() is encoded.

In order to handle this issue, a wrapper function (denoted as wrap_getproc) for GetProcAddress is generated as shown in line 6 in List. 3. It moves the runtime-generated function pointer to a preserved return slot, and then returns the address of the associated preserved trampoline stub. Moreover, GetProcAddress itself is an import function, so it needs to be encoded as (b), as shown in line 9.

The preserved return slot can be written by the wrapper function wrap_getproc to the store function pointer returned at runtime. However these return slots should be protected from being altered by attackers, so we take the following approach. To start, all return slots are set read-only. When wrap_getproc tries to write to a return slot, it sets the corresponding memory page to be writable. After it writes the returned function pointer to this slot, the memory page is set back to be read-only. In such a way, the attacker cannot hijack the return slots to control the program's behavior. Due to the contest deadline limitation, this protection has not been deployed yet in our prototype. But GetProcAddress is not used in the performance critical paths for normal programs, so it does not influence the performance much.

Listing 4: Encode sources: generated by setjmp()

```
1 // in new_setjmp:
2 //     call original setjmp(jmp_buf)
3 //     encode the function pointer field of
      jmp_buf
4 new_setjmp = gen_or_get_setjmp()
```

Function pointers of kind (d) are generated at runtime as well. setjmp() retrieves its caller's return address from the stack (an unencoded function pointer) and saves it into the jmp_buf structure (setjmp's argument), and finally this function pointer (i.e. return address) will be used by longjmp(). Similar to (c), we generate a wrapper function for setjmp(), as shown in List. 4. This wrapper calls original setjmp() and then encodes the function pointer in the jmp_buf.

setjmp() is called limited in a just a few locations in the program. All these call sites can be identified offline. As a result, the function pointers generated at runtime by setjmp (i.e. all call sites' next instructions' addresses) can be identified. For each candidate function pointer *fp* generated by setjmp, a trampoline *stub* is preserved. And the map relationship from *fp* to *stub* is stored in a read-only memory.

The wrapper for setjmp encodes the function pointer of jmp_buf as follows: 1) it looks up the memory which stores the map relationship, and then 2) gets the address of the function pointer's associated trampoline stub, and finally 3) this stub address is stored back to the function pointer. As setjmp() is seldom used nowadays, this is not included in our current prototype.

3.3.2 Decode function pointer sinks

Function pointers' sinks include:

(a) Indirect *call/jmp* instructions in the executable file itself. Indirect instructions that use function pointers or import slots directly, such as *call [imp_slot]*, do not need decoding.

(b) The callback arguments of some imported functions, such as the first argument of *_onexit()* function in *MSVCR100.dll*.

For function pointer sinks of kind (a), as shown in List. 5, the original indirect control-transfer instruction is replaced with a jmp instruction. The target of the jump is an instrumented code block either in the padding area of the original executable image or in a new created section.

The instrumented code block validates the original jump target to ensure that it was a valid encoded address. It checks whether the address is 16-byte aligned, and whether the 4 bytes before the target (i.e. prev stub's last 4 bytes) is the predefined prolog. If the validation passes, the control flow transfers to the encoded target (i.e. the instrumented trampoline stub). Otherwise, it jumps to a predefined error handler and then exits safely.

Listing 5: Decode sinks: indirect instructions

```
1 // original: call tgt
2 // hardened:   jmp new_addr
3 // new_addr:
4 //     mov eax, tgt
5 //     test al, 0x0f
6 //     jnz error_handler
7 //     cmp [eax-4], prolog
8 //     jne error_handler
9 //     jmp eax
```

It is worth noting that, the predefined prolog is chosen carefully, i.e. 0xCC02EBCC in our prototype implementation. Besides, each instrumented prolog's address is in the form of $16 * x - 4$, i.e. its end is 16 bytes aligned. By comparison, such byte sequences rarely occur in original executables (much less 16-byte aligned), because 0xCC is a special breakpoint instruction *int 3*. In the worst case, there is a prolog whose end is 16 bytes aligned in the original executable (which could occur in theory occur as operand of an instruction), we can rewrite this instruction so that it no longer matches the prolog or is differently aligned.

Thus this uniqueness lets us conclude that attackers cannot find a faked trampoline stub in

original code sections to forge a indirect call/jmp's target.

For function pointer sinks of kind (b), encoded function pointers flow into external modules. As discussed earlier, if FPGate is applied onto the whole system, there is no need to consider how these function pointers flow into external modules.

But currently, FPGate can only be applied on a subset of the system, so function pointers should be decoded before flowing into external modules those not hardened by FPGate.

Interoperability Unlike other potential encode/decode schemes, the one we present here has an advantage on compatibility. That is, the encoded function pointers do not need to be decoded before flowing into external modules. Because the encoded function pointers are addresses of instrumented trampoline code, i.e. they are valid function pointers and can flow into external modules safely. In other word, function pointers which are encoded without decoding work fine everywhere. As a result, FPGate is a gradual hardening scheme. It is compatible with existing software.

Note that the converse is not true. A function pointer that is not encoded (e.g. one supplied by attacker) but decoded by FPGate will be caught at runtime. Catching such cases is the motivation for using FPGate in the first place. In the case where protected and unprotected code are mixed, this behavior could cause a false positive if a function pointer were passed from unprotected code to protected code, but this seldom happens in normal program hardening situations.

4 Evaluation

4.1 Compatibility

As shown earlier, BitRewrite modifies function pointers and rewrites indirect call and jmp instructions in the executable. If a functions pointer is missed by the rewriter and thus doesn't get encoded correctly, the program will fail at the decode points.

In order to evaluate FPGate's compatibility with existing real world software, and show that the rewriting produces semantically equivalent code, we tested it with the 13 applications included in the SPEC-CPU2006-INT benchmark test suite [2].

First, all the 13 applications were compiled using Microsoft Visual Studio 2010. We made release builds with the /GS (Buffer Security Check) flag on, but only 12 of them were built successfully. The application *libquantum* fails because MSVC does not support the C99 feature, e.g. type *_complex*.

Secondly, we used FPGate to automatically rewrite all these 12 successfully built applications. We only modified the executable image itself. We left all external modules (e.g. DLLs) intact.

Thirdly, we ran the 12 hardened applications using the test harness of the SPEC2006 benchmark suite and checked their results. All the rewritten programs ran succesfully, meaning that all test output was that same as the original, unmodified applications'. SPEC2006 uses a large test data set, so this is a good indication that the applications remained semantically equivalent. In other words, FPGate can work compatibly with existing software.

4.2 Performance

In order to evaluate the overhead brought by FPGate, we also compared the running time of original and the rewritten applications in the SPEC-CPU2006-INT test suite. We conducted the experiment on a Windows 7 32bit system, with the Intel Core(TM)2 Duo CPU E8400 @ 3.00GHz. The SPEC2006 benchmarks were configured to run within 1 thread in 1 core on 1 chip.

Runtime overhead Table 1 shows the performance overhead of FPGate. The average measured overhead is about 0.4%. Since the overhead is so minimal and measuring running time is not 100% accurate, in some cases the table shows small speedups as well. This only means that the the overhead is comparable to the measurement noise.

Runtime overhead The number of modifications made by FPGate is shown in table 2. The columns under #sources and #sinks in the table represent the the number of function pointers modified, the number of import function pointers modified, if there was a function pointer returned by GetProcAddr, and the number of indirect call/jmp instructions overwritten. Taking *perbench* as an example, its original file size was 1047 kB and after rewritten by FPGate, its file size incremented by 39kB, which is less then 4% size overhead. More specifically, there were 1409 function pointers which were encoded by FPGate, and each of which introduced a 16-bytes trampoline stub. Moreover, 485 indirect call/jmp instructions were rewritten, each of them introduced about 20 bytes overhead. Applications hardened by FPGate doesn't introduce runtime memory overhead, except for the statically instrumented sections.

As described earlier, 12 applications were built and hardened successfully by FPGate, but only 11 of them are listed in these two tables. This is because there is a special application *999.specrand*

Table 1: Performance Overhead

Spec2006 Benchmarks	Original (MSVC2010 Release) Run Time (s)	Hardened by FPGate Run Time (s)	Performance Overhead
400.perlbench	497	504	1.41%
401.bzip2	565	565	0.00%
403.gcc	407	406	-0.25%
429.mcf	340	339	-0.29%
445.gobmk	554	554	0.00%
456.hmmer	1126	1126	0.00%
458.sjeng	669	671	0.30%
464.h264ref	836	845	1.08%
471.omnetpp	374	379	1.34%
473.astar	418	421	0.72%
483.xalancbmk	288	287	-0.35%
		Average	0.36%

Table 2: Modifications made by FPGate to applications

Spec2006 Benchmarks	orig file size (k)	# sources			# sinks	new file size (k)	Δfile size (k)
		#func_ptr	#imp_slot	GetProcAddr	#indirect inst		
400.perlbench	1,047	1409	26	yes	485	1,086	39
401.bzip2	116	102	13	yes	114	128	12
403.gcc	3,038	3530	15	yes	763	3,085	47
429.mcf	80	102	13	yes	96	92	12
445.gobmk	3,218	1996	13	yes	173	3,244	26
456.hmmer	230	146	13	yes	116	246	16
458.sjeng	194	113	13	yes	98	199	5
464.h264ref	575	214	13	yes	456	599	24
471.omnetpp	810	4827	15	yes	1840	926	116
473.astar	117	124	13	yes	100	130	13
483.xalancbmk	3,728	235	13	yes	317	3,750	22

in the SPEC2006 benchmarks, which is not inted-ed to be used in performance benchmarks, only for corrrectness tests, so its is not shown here.

4.3 Protection Effects

We have built a demo program to emulate a buffer overflow vulnerability which allows altering a function pointer. The function pointer is called later in this vulnerable program.

The demo program will trigger the *calc.exe* to be executed if the program is supplied with our sample exploit input. After being hardened with FPGate, attacks against this vulnerability are caught at runtime by directing the execution to our error handler which terminates the execution.

5 Discussion

5.1 Possible Attacks

FPGate protects all indirect call/jmp transfer instructions. It restricts such instructions to jumping to legal targets, and these targets can be validated statically after binary rewriting.

Possible attacks against FPGate may come from three directions:

(a) An attacker may forge a valid target.

(b) An attacker may change pages' protection attributes to change instructions directly or to add forged targets.

(c) An attacker may use a dangerous target that is valid because the program also uses it

For (a) the attacker has to use a page which is writable and executable at the same time. For modern programs obeying W⊕X policy, this depends on the attack (b). Otherwise, it can only generate a DEP exception.

Some API functions are inherently dangerous ($WinExec$), or are dangerous because they can disable page protections ($VirutalProtect$, $VirtualAlloc$ and $VirtualAllocEx$, $Virt*$ for short). The use of these functions through indirect calls is rare in the case of regular applications, but FPGate can alert the user if such functions can get into the set of valid targets; in this case it is probably best to modify the application.

If a program calls $Virt*$ functions directly and only uses constant $flProtect$ or $flNewProtect$ arguments which doesn't make the page executable, it will be immune to such attacks of (a) or (b) after being hardened by FPGate. If a program calls $Virt*$ functions to make a page executable for JIT, attackers still have a chance to utilize these functions. We suggest that the program carefully verify the arguments before calling.

5.2 Practical CFI solutions based on FPGate

DEP and Safe-SEH are mature solutions and provide solid protections against attack (a)&(c) described in section 2.3. But existing mechanisms against attack (b) for return address do not provide full protection. For instance the /GS Buffer Security Check protects only against continuous stack overflows and not against direct overwrites (eg. in case of a Write-What-Where condition).

FPGate provides a solid solution against attack (d). Combining it with different existing protection approaches, we can have different CFI enforcement solutions.

5.2.1 Full CFI enforcement

We can choose strong protection for return addresses such as implementing a protected shadow stack, as described in [1]. Thus we can enforce CFI solidly. The overhead of our own shadow stack implementation is about 4.2%, benchmarked using the SPEC2006 suite.

This will be a very secure protection for software control-flow integrity. The only way to break it is to call *VirtualProtect* to change pages' attributes. It is very hard to do so under FPGate, as discussed in section 5.1.

5.2.2 Quasi CFI enforcement

Shadow stack is not adapted as widely as stack cookies such as GS or SSP. If we use GS or SSP to protect return addresses, we implement a quasi-CFI enforcement. Such a solution provides better performance than full CFI enforcement. When applying stack cookies to all functions, their overhead is from 2.38%(llvm) to 2.77%(gcc), as measured in our experiments on the SPEC2006 benchmarks.

The weakness of such solutions is that they don't protect against direct return address overwrites and also if an attacker can guess the secret cookie value by exploiting an information leakage, she can forge one to circumvent the protection.

5.2.3 CFI enforcement in JIT scenarios

In JIT scenarios, there are two main issues to solve:

- ◇ The JIT program will call *VirtualProtect* or *VirtualAlloc* with executable attributes legally.
- ◇ Most JITs don't respect W⊕X policy for performance reasons.

The first issue is discussed in section 5.1. For the second issue, software approaches to memory sandboxing [4] could be used to prevent most code in an executable from modifying the executable JIT code pages. We are interested in investigating this combination in future work.

6 Conclusion

In this paper, we propose a new approach called FPGate to ensure that indirect calls and jumps, including function pointers jump only to known targets. It can block various attacks against function pointers, including most Use-After-Free attacks.

FPGate can be applied through binary rewriting on executables generated by modern compilers. Its runtime overhead is very low (about 0.4% measured by SPEC2006). FPGate's techniques can also be used directly in the compilation process to provide protections for software.

Further, when combined with existing protection mechanisms, it can be used to enforce CFI, which provides a solid base for software protection.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):1–40, October 2009.

[2] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[3] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *NDSS*, 2012.

[4] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security - CCS '11*, page 29, New York, New York, USA, October 2011. ACM Press.