Protecting Function Pointers in Binary

Chao Zhang
LiST, Peking University
Beijing, China
chao.zhang@pku.edu.cn

Tao Wei*
LiST, Peking University
UC Berkeley
wei_tao@pku.edu.cn

Zhaofeng Chen
LiST, Peking University
Beijing, China
chenzhaofeng@pku.edu.cn

Lei Duan
LiST, Peking University
Beijing, China
lei_duan@pku.edu.cn

Stephen McCamant
University of Minnesota
USA
mccamant@cs.umn.edu

Laszlo Szekeres
Stony Brook University
USA
lszekeres@cs.stonybrook.edu

ABSTRACT

Function pointers have recently become an important attack vector for control-flow hijacking attacks. However, no protection mechanisms for function pointers have yet seen wide adoption. Methods proposed in the literature have high overheads, are not compatible with existing development process, or both. In this paper, we investigate several protection methods and propose a new method called FPGate (i.e., Function Pointer Gate). FPGate rewrites x86 binary executables and implements a novel method to overcome compatibility issues. All these protection methods are then evaluated and compared from the perspectives of performance and ease of deployment. Experiments show that FPGate achieves a good balance between performance, robustness and compatibility.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

Keywords

Function Pointer Protection; Binary Rewriting

1. INTRODUCTION

Control-flow hijacking attacks [4, 16] have a long history, and corresponding protections [3, 17] have also existed for more than a decade, in an ongoing arms race. The current state of the art of protections pay much attention to the return addresses on the stack (i.e. targets of return instructions), but provides few protections against function pointers (i.e. targets of indirect call and jump instructions). And thus, function pointers have recently become an important attack vector. For instance, a buffer overflow or integer overflow vulnerability can be exploited to overwrite function pointers in the heap to hijack the control flow [6, 8]. Moreover, recent exploits [19] against use-after-free vulnerabilities replace objects' virtual function tables (i.e. `vtables`) to turn benign method calls into jumps to shellcode.

Representative protection schemes which protect function pointers from exploiting including PointGuard [7] and CFI (Control-Flow Integrity [1]). However, they require cooperation from developers or code-producers, or suffer from inefficiency and compatibility problems. We propose a new method called FPGate to achieve a balance.

First, FPGate needs no source-level information (e.g., PDB files or debug information needed by CFI). It utilizes relocation tables which are already required by ASLR (Address Space Layout Randomization [17]) in modern x86 binary executables, to disassemble binaries and identify all indirect transfer instructions and all valid jump targets.

Second, FPGate overcomes compatibility issues by encoding each valid function pointer into a pointer to a new *trampoline* chunk. Unlike other schemes, modules hardened by FPGate can inter-operate seamlessly with un-hardened ones.

Moreover, optimizations are introduced by FPGate to accelerate the run time execution, without any loss of security. As a result, FPGate introduces a negligible run time overhead, about 0.4% on SPECint2006.

FPGate fills most of the gap between existing lightweight protections on one hand, and CFI on the other. Combining FPGate with robust protections of return addresses is a sweet spot for security and usability. This combination provides a protection almost as strong as the original CFI. At the same time, it has low overhead and can be applied directly and progressively to a binary.

The key contributions of this paper are:

- We investigate several existing function pointer protection mechanisms, and point out their advantages and disadvantages.
- We propose a new efficient mechanism FPGate to protect function pointers in binary executables and provide incremental deployment.
- We compare FPGate with existing mechanisms in an experimental evaluation. The results show that FPGate achieves a good balance between performance, robustness and compatibility.

*Corresponding author.

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

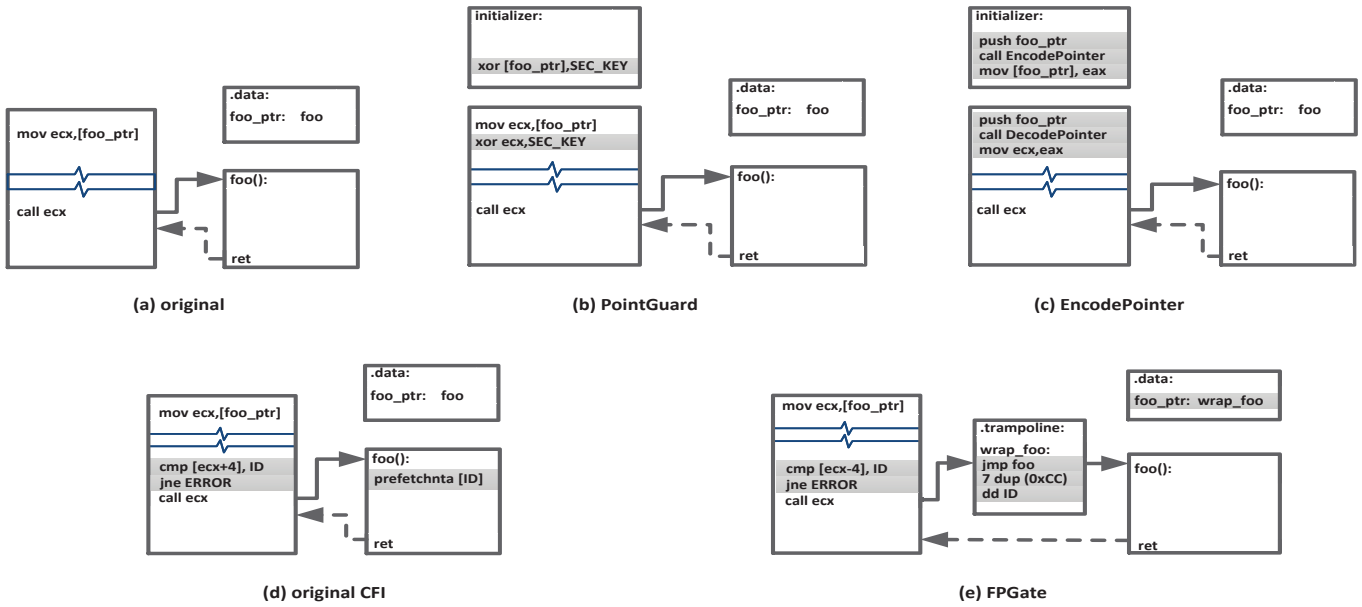

Figure 1: (a) An indirect call whose target is from memory. (b)-(e) Overview of existing function pointer protection schemes and FPGate.

2. STUDY OF EXISTING PROTECTIONS

As an important attack vector, the function pointer has drawn researchers' attentions for years. In existing protection techniques, legal function pointers are differentiated from illegal ones by encoding (or encrypting), by attaching identifiers (IDs), or by memory alignments. Several representative mechanisms are discussed here.

2.1 PointGuard

PointGuard [7] uses encryption to provide integrity for function pointers. A per-process secret key, i.e., `SEC_KEY`, is kept to encrypt and decrypt pointers and thus protect legal pointers from tampering. Before a function pointer is stored into memory, it is encrypted (i.e. XOR) with the secret key first. After the function pointer is read from memory, it is decrypted (i.e. XOR) and then stored in registers for use.

Figure 1(a) shows an indirect call example whose target is read from memory. After PointGuard is applied to this code snippet, the hardened version is shown in Figure 1(b). It is worth noting that a special initializer is instrumented by PointGuard to re-initialize all statically initialized pointers.

The integrity of function pointers hardened by PointGuard depends on the confidentiality of the secret key. However, the secret key is vulnerable to be stolen by attackers, such as through information leakage vulnerabilities. In addition, because the XOR operation is linear, an attacker may be able to manipulate an encoded pointer by overwriting only the lower bits of an address and launch an attack [2].

PointGuard works as a compiler extension and thus needs a target application's source code. Besides, with encryption and decryption, function pointers cannot flow between hardened modules and un-hardened ones.

2.2 EncodePointer

As PointGuard was never released, Microsoft implemented a similar approach beginning in Windows XP SP2. It provides two API routines that can be called at the discretion of the programmer, named `EncodePointer` and `DecodePointer` [13]. Figure 1(c) shows the hardened version using `EncodePointer` for the code snippet in Figure 1(a).

Rather than storing the secret key in user space like PointGuard, `EncodePointer` queries the system API `NtQueryInformationProcess` to acquire the secret key. It prevents a user-level attack from accessing the secret key directly, ameliorating the problem of information leakage vulnerabilities.

However, programmers have to decide which points to invoke `EncodePointer` and `DecodePointer` manually. In addition, each call to `EncodePointer/DecodePointer` queries the kernel, and introduces a significant run time overhead.

The requirement for source code and the compatibility problems between hardened and un-hardened modules in this protection scheme are analogous to those in PointGuard.

2.3 CFI

CFI [1] is a natural protection against control-flow hijacking attacks. It guarantees 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).

In general, it inserts an ID (e.g., the `prefetchnta` instruction in Figure 1(d)) before each legal target (e.g., a function entry), and a check before each indirect transfer instruction. Before transferring to the computed target at runtime, the dynamic check validates whether the target's ID is correct.

CFI defeats a broad range of shellcode injection attacks, including sophisticated ROP (Return Oriented Programming [4]). However, despite its long history, CFI has not seen wide industrial adoption.

It is in part because CFI imposes a significant overhead. Even if only function pointers are protected by CFI (called as CFI-fp), it also introduces an overhead of 7.2%, as shown in Section 4.3. In addition, it requires source-level information (e.g., debug information) which are usually not available in COTS binary. Moreover, modules hardened by CFI cannot inter-operate seamlessly with un-hardened ones. For example, the instrumented runtime check will fail if the computed jump target falls into an un-hardened module.

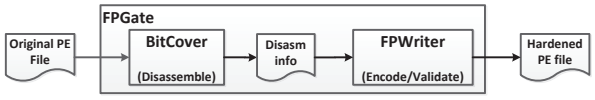

Figure 2: Architecture of FPGate

3. DESIGN & IMPLEMENTATION

3.1 Approach Overview

As PointGuard, FPGate encodes legal function pointers to differentiate them from illegal ones. Unlike PointGuard, pointers encoded by FPGate are valid code pointers, and can flow into un-hardened modules without compatibility issues.

On the other hand, a unique ID is attached to each encoded pointer for validating, as CFI. A novel code section, i.e., the *trampoline* section, is thus introduced for encoding pointers and holding IDs. Each valid function pointer is encoded into a pointer to a code chunk in the *trampoline*. Unlike CFI, most instructions' addresses will not change after rewriting. Meanwhile, with the *trampoline* layer, computed jump targets which fall into un-hardened modules can also be encoded to overcome compatibility issues.

The architecture of FPGate is shown in Figure 2. It consists of two core modules: BitCover and FPWriter. BitCover disassembles target binary and identifies all indirect call and jump instructions in addition to all valid jump targets. FPWriter then instruments runtime checks before indirect transfers, encodes valid jump targets with new code pointers, and creates the trampoline section. Figure 1(e) shows the code snippet which is rewritten by FPWriter.

3.2 Disassembling and Identifying

In general, it is challenging to disassemble an x86 PE [14] file correctly, because x86 is a CISC platform. However, we can take advantage of the fact that ASLR and DEP (Data Execution Prevention [3]) are widely adopted in Windows/x86 executables, particularly those whose developers care about security.

Combined with other policies, a custom disassembler BitCover [21] is built and can disassemble PE files generated by modern compilers. With BitCover, all indirect jumps or calls and their valid targets can be identified.

3.3 Validating Jump Targets

After identifying all indirect *call/jmp* instructions by BitCover, FPWriter instruments runtime checks before them.

As shown in Figure 1(e), a check which verifies the existence of a predefined ID around the jump target is inserted before the indirect call instruction (i.e., *call ecx*).

Similar as the CFI, the ID is carefully chosen in order to avoid conflicting with bytes in the application.

3.4 Encoding Legal Targets

Valid jump targets should be encoded to distinguish from invalid targets. According to our observation, legal jump targets fall into the following categories:

(A) Hard-coded function pointers in the executable file, such as those stored in the *vtables* and global variables.

(B) Entries in the import table.

(C) Function pointers generated at runtime by API, such as `GetProcAddress()`.

(D) Function pointers generated by `setjmp()`.

For function pointers of kind (A), they are simply encoded into a trampoline chunk pointer, as shown in Figure 1(e).

Considering the performance, hard-coded pointers which are directly used in `call/jmp` instructions, e.g., `call foo`, would not be encoded. Exception handlers used by the operating system are skipped encoding too. Moreover, code entries in switch tables [5] are not encoded. All these skipped pointers are protected by SafeSEH [12] or DEP, and cannot be tampered by attackers, and thus can be skipped safely.

The remaining cases B-D involve pointers from external modules, even un-encoded pointers, and thus cause compatibility issues. Unlike existing protections (e.g., CFI), FPGate can handle these issues with the extra *trampoline* layer.

3.5 Trampoline & Compatibility

FPWriter encodes each valid jump target into a pointer to a chunk in the trampoline. Each of these chunks holds a same ID and has 16 bytes. These chunks are continuous and can pass the validations instrumented before indirect jumps. As Figure 1(e) shows, the chunk will jump back to the original target to ensure applications behave correctly.

In the case of (B), function pointers are imported from external modules and will be updated when loading. As shown in Figure 3(a), a wrapper section (*.wrap*) is introduced to simulate the original IAT (Import Address Table, *.iat*). Whereas the IAT stores imported function pointers, the wrapper section stores pointers to trampoline chunks. Then, all references to the IAT are replaced with references to the wrapper section. Similarly, directly used imported function pointers, such as `call [imp_slot]`, are skipped encoding too in order to promote the performance.

For function pointers of kind (C), they are retrieved through API `GetProcAddress`, so they need be encoded at runtime. `GetProcAddress` is usually imported, and thus it can be processed in a same way as (B), except that the trampoline chunk for `GetProcAddress` does not jump back to original imported `GetProcAddress` directly. In fact, it jumps to a wrapper function for `GetProcAddress` (i.e. `gpa_wraper`), as shown in Figure 3(b). The wrapper function will call original `GetProcAddress`, and then stores the runtime retrieved function pointer into a preserved trampoline slot. And then the pointer to the preserved slot is returned as the new return value. As a result, the runtime retrieved function pointer is also encoded. It is worth noting that, the wrapper function has to set the preserved trampoline slots writable at runtime and turn it off when the wrapper function exits, in order to protect the preserved trampoline slots from tampering.

Function pointers of kind (D) are generated at runtime as well. The function `setjmp` retrieves its caller's (un-encoded) return address from the stack and saves it into a `jmp_buf` structure. Finally, this return address will be used by an indirect jump instruction in another API `longjmp` through querying the `jmp_buf` structure. We then replace each call instruction which invokes `setjmp` with two instructions: `PUSH ret_addr; JMP setjmp;`. The pushed return address `ret_addr` is a trampoline slot pointer, and it will jump back to the original return address, as shown in Figure 3(c).

As discussed above, the trampoline provides a demilitarized zone between hardened modules and un-hardened ones, and thus can help overcome the compatibility issues. This novel method can also be utilized and applied to other binary protection schemes, such as [21].

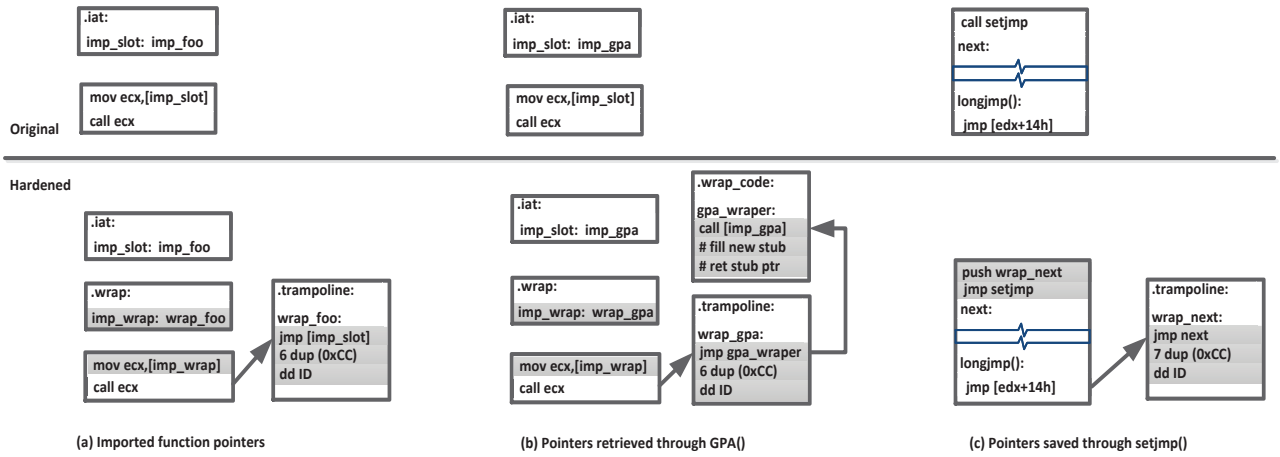

Figure 3: Special function pointers and FPGate's encoding countermeasures

4. EVALUATION

We implement a prototype of FPGate using C++ for the Windows platform targeting x86 PE executables. BitCover takes about 5K LOC and utilizes the Udis86 library (8K LOC, [18]) to parse x86 instructions, while FPWriter takes another 5K LOC and a custom PE file parser has 2K LOC.

We port PointGuard, EncodePointer and CFI discussed in Section 2 using binary rewriting, and then evaluate and compare their compatibility and performance with FPGate.

4.1 Correctness of FPGate

FPWriter encodes function pointers and instruments runtime checks before indirect `call/jmp` instructions in the executable. If there is an error in this rewriting process, the final hardened application will fail or crash at runtime. For example, if a function pointer fails to be identified by BitCover, then it will not be encoded by FPWriter, and finally cannot pass the instrumented runtime check.

Two groups of experiments are made to evaluate FPGate's correctness. First, FPGate is tested with 11 applications from the SPECint2006 benchmark [9]. Second, FPGate is tested with two real world browsers, Firefox 3.6.16 (FF3) and Internet Explorer 6 (IE6)[1]. In particular, the core modules `xul.dll` in FF3 and `mshtml.dll` in IE6 are hardened by FPGate, and other modules are left intact. These experiments show that hardened applications work correctly; i.e., they are semantically equivalent to original applications.

4.2 Compatibility Comparison

In this section, we discuss the compatibility of FPGate, and make a comparison with the other three protection schemes, i.e. PointGuard, EncodePointer and CFI-fp (a limited version of original CFI which protects only function pointers).

4.2.1 Compatible with Binary Applications

FPGate can be applied directly on binaries containing relocation tables, even without source-level information. For (seldom) binaries without relocation tables, FPGate work fine with the help of other tools or experts. For example, the commercial disassembler IDA Pro [10] can be easily utilized to generate a fake relocation table and help FPGate to be applied on these binaries.

On the other hand, original implementations of PointGuard and EncodePointer depend on source code, while the CFI-fp depends on debug information. However, in our porting, they are all ported successfully using binary rewriting.

4.2.2 Compatible with Un-hardened Modules

Function pointers may flow from hardened modules to unhardened modules and vice-versa, and thus cause compatibility issues. With countermeasures presented in Section 3.5, modules hardened by FPGate can work seamlessly with unhardened modules. The previous section which evaluates correctness of FPGate also proves this compatibility.

But for other three schemes, the compatibility issue is critical. Special efforts are need to fix this issue when porting.

In the case of PointGuard and EncodePointer, encoded function pointers are not valid pointers, i.e. they cannot be used directly in any un-hardened module. When porting, the binary has to be rewritten in such a way that each function pointer is decoded before it flows into external modules. For each specific application, lots of manual efforts are needed to identify and fix such pointer flows between modules.

On the other hand, function pointers from un-hardened modules may fail the runtime checks of the hardened modules. For imported function pointers, a bootstrap code chunk is inserted to encode them. For pointers retrieved by `GetProcAddress`, similar wrappers as FPGate are provided too. For `setjmp` and other cases, it is also handled like FPGate.

In the case of CFI-fp, compatibility issues occur when computed jump targets fall into un-hardened modules. In our porting, the error handler is set to be empty and will redirect the control flow to its original position.

In summary, FPGate has a good compatibility. PointGuard or EncodePointer needs a lot of manual effort to achieve a good compatibility. Compatibility issues of the original CFI implementation are hard to be fixed unless another layer (like the trampoline in FPGate) is introduced.

4.3 Performance Comparison

The performance experiments are made on a Windows 7 32-bit system, with an Intel Core2 CPU of 3.00GHz. The

[1]Core modules of newer IE cannot be replaced with custom modules because of the system protection. In addition, there are no public available exploits for newer browsers. As a result, we chose these two old softwares as a benchmark.

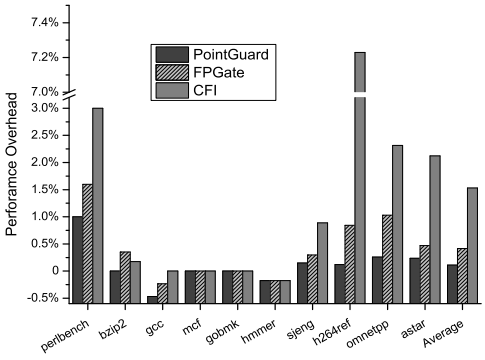

Figure 4: Overheads of PointGuard, FPGate and CFI-fp.

scripts in SPECint2006 are used to evaluate the original and hardened versions of applications' performance. And the average performance overheads are computed based on 9 trials.

Runtime overhead.

Figure 4 shows the performance overheads caused by FPGate, PointGuard and CFI-fp. The average/maximum overhead is about 0.11%/1% for PointGuard, about 0.42%/1.03% for FPGate, and about 1.53%/7.23% for CFI-fp. For most applications, the overhead brought by PointGuard is smallest, while that brought by CFI-fp is largest.

In addition, EncodePointer's performance is quite poor. Its average/maximum overhead is about 92%/750%, not shown in this figure due to the space limitation.

Statistics.

The upper half of Table 1 lists the count of modifications made by FPGate to the SPECint2006 applications, and the lower half for other real world applications.

More specifically, the columns under *Encoded pointers* in the table represent the count of pointers encoded by FPGate, including hard-coded function pointers, imported function pointers, pointers returned by `GetProcAddress` and pointers generated by `setjmp` functions.

The column under *Checks* records the count of indirect `call/jmp` instructions which are instrumented by FPGate, while the final column (i.e. the column under *Opt.*) collects the count of function pointers those are skipped encoding.

Performance Analysis.

PointGuard is fastest because it only needs one XOR instruction to encrypt and decrypt the jump targets. EncodePointer is slowest, because it queries the kernel heavily.

For the porting of CFI-fp, its overhead is about 1.5%, much faster than the original implementation in [1]. That is because this porting only protects function pointers but not return addresses.

But the CFI-fp is slower than FPGate, mainly due to the optimizations introduced by FPGate. As shown in the last column of Table 1, the count of function pointers skipped encoding is quite large, usually larger than the count of encoded function pointers. This optimization can greatly promote the performance of FPGate. However, this optimization cannot be deployed easily to CFI-fp. For example, for instruction *call foo*, FPGate can skip encoding the *foo* and thus introduce no overheads, but CFI cannot skip inserting an ID at the beginning of function *foo* because *foo* may be used as the target of another indirect jump.

In addition, the ID inserted by CFI-fp at the beginning of each jump target (e.g. function entries) is the `prefetchnta` instruction. This instruction has no side-effect but is extremely slow. For each direct (relative or absolute) jump or call instruction, the instrumented slow `prefetchnta` instructions at the beginning of the jump target will be always executed. As a result, CFI-fp is a little slower than FPGate.

Table 1: Modifications made by FPGate to applications

Benchmarks	Encoded pointers				Checks	Opt.
	#fp	#imp	#GPA	#setjmp	#indirect call/jmp	#skipped fp/import
SPECint2006						
400.perlbench	1,380	46	5	1	495	2,702
401.bzip2	73	20	3	0	123	290
403.gcc	3,501	26	3	1	774	10,254
429.mcf	73	20	3	0	105	238
445.gobmk	1,967	22	3	0	185	566
456.hmmer	117	23	3	0	125	429
458.sjeng	84	20	3	0	107	407
464.h264ref	185	20	3	0	465	369
471.omnetpp	4,798	30	3	1	1,853	1,065
473.astar	95	20	3	0	109	256
Browsers						
mshtml.dll (IE6)	1,526	139	21	0	10,452	29,557
xul.dll (FF3.6)	145,224	283	34	0	55,025	17,273

Table 2: Real World Exploit Samples Prevented by FPGate.

ID	App	Vul Type	Vul Module	Protected
CVE-2011-0065	FF 3.6	Use After Free	xul.dll	yes
CVE-2010-0249	IE 6	Use After Free	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

4.4 Security Comparison

FPGate limits indirect instructions transfer only to valid targets with IDs. There is only one ID in FPGate, and thus it provides a weaker protection than CFI. However, due to the complexity of building a complete control flow graph, the count of unique IDs introduced by CFI for a given application is also small. And thus, the security gap between FPGate and CFI is not so much.

We chose 7 publicly available exploits from Metasploit [11] and ExploidDB [15] and tested them in a virtual machine running Windows XP SP3 within a separate experiment network. These exploits are tested against the original vulnerable applications and corresponding hardened applications. Table 2 shows the 7 vulnerabilities attacked by exploits we used. Results show that all these protection schemes can protect target browsers from attacking.

4.5 Miscellaneous

FPGate disassembles target binary and rewrites them. The static analysis time is positively correlated with the target file size and is usually small. As our experiments shows, it costs 63 seconds for `gcc` which is 1,200KB, and costs only 0.16 seconds for `mcf` which is 80KB.

FPGate instruments a trampoline section and allocates 16-bytes for each encoded function. The runtime memory overhead it introduced is positively correlated with the count of functions. For example, `perlbench` has about 700 functions, and the runtime memory overhead is about 10KB.

5. DISCUSSION

FPGate relies on relocation tables to disassemble binary and rewrite them. A recently published work REINS [20] utilizes IDA Pro to provide similar protection without relocation tables. However, IDA Pro is heuristic and suffers significant disassembly errors. REINS cannot identify all valid function pointers and thus maintain a lookup table to encode function pointers at runtime. A larger runtime overhead is then introduced. Moreover, if the computed jump targets fall into external modules and are retrieved through methods like `GetProcAddress`, REINS fails to protect the control flow, and causes compatibility issues.

FPGate uses only one ID and thus permit jumps to any valid target with an ID. This is vulnerable to jump-to-libc attacks. Attackers can hijack controls to valid jump targets and launch an attack. Further work is needed to defend against this kind of attack.

6. CONCLUSION

In this paper, we investigate several protection methods against function pointers, including PointGuard, EncodePointer and CFI. Inspired by them, a new approach called FPGate is proposed. FPGate limits all indirect calls and jumps to known valid targets. It can block various attacks against function pointers.

FPGate can be applied through binary rewriting on executables generated by modern compilers. It is compared with existing protections from the perspectives of performance, robustness, and ease of deployment. Results show that FPGate achieves a good balance between performance, robustness and compatibility.

Acknowledgments

This research was supported in part by the National Natural Science Foundation of China under the grant No. 61003216 and No. 61003217, the Chinese NDRC InfoSec Foundation under Grant No.[2010]3044, the National Science Foundation under Grant No. 0842695, 0831501 CT-L, CCF-0424422 and CNS-0831298, the Office of Naval Research under MURI Grant No. N000140911081, N000140710928 and award FA9550-09-1-0539, the AFOSR under grant FA9550-09-1-0539, and DARPA under award HR0011-12-2-005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] S. Alexander. Defeating compiler-level buffer overflow protection. *The USENIX Magazine ;LOGIN*, 30(3):59–71, 2005.

[3] 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.

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, 2008.

[5] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the 7th International Workshop on Program Comprehension*, pages 192–199. IEEE, 1999.

[6] CORE Security Technologies Advisories. Core-2007-0219: Openbs's ipv6 mbufs remote kernel buffer overflow. `http://www.securityfocus.com/archive/1/462728/30/150/threaded`, 2007.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, volume 12, pages 91–104, 2003.

[8] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with javascript. In *Proceedings of the 2nd conference on USENIX Workshop On Offensive Technologies*, 2008.

[9] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, Sept. 2006.

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

[11] Metasploit Open Source Commitment. Metasploit penetration testing software & framework.

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

[13] MSDN online library. EncodePointer function: provide another layer of protection for pointer values. `http://msdn.microsoft.com/en-us/library/bb432254(v=vs.85).aspx`.

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

[15] Offensive Security. Expoit database.

[16] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7:14–16, 1996.

[17] PaX Team. Pax address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[18] V. Thampi. Udis86 disassembler library for x86. `http://udis86.sourceforge.net/`.

[19] P. Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. `http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf`, 2010.

[20] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, Orlando, FL, December 2012.

[21] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2013.