ISboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries

Liang Deng(✉), Qingkai Zeng, and Yao Liu

State Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology, Nanjing University,
Nanjing 210023, China
dengliang1214@smail.nju.edu.cn, zqk@nju.edu.cn

Abstract. Dynamically-linked libraries are widely adopted in application programs to achieve extensibility. However, faults in untrusted libraries could allow an attacker to compromise both integrity and confidentiality of the host system (the main program and trusted libraries), as no protection boundaries are enforced between them. Previous systems address this issue through the technique named data sandboxing that relies on instrumentation to sandbox memory reads and writes in untrusted libraries. However, the instrumentation method causes relatively high overhead due to frequent memory reads in code.

In this paper, we propose an efficient and practical data sandboxing approach (called ISboxing) on contemporary x86 platforms, which sandboxes a memory read/write by directly substituting it with a self-sandboxed and function-equivalent one. Our substitution-based method does not insert any additional instructions into library code and therefore incurs almost no measurable runtime overhead. Our experimental results show that ISboxing incurs only 0.32%/1.54% (average/max) overhead for SPECint2000 and 0.05%/0.24% (average/max) overhead for SFI benchmarks, which indicates a notable performance improvement on prior work.

Keywords: Data sandboxing · Instruction substitution · Untrusted libraries · Instruction prefix

1 Introduction

Applications commonly incorporate with dynamically-linked libraries to achieve extensibility. However, a library, which might be buggy or even come from a malicious source, could be used by attackers to disrupt both integrity and confidentiality of the host system. Even though the host system contains no vulnerabilities, bugs and malicious behaviors in the library can lead to compromise of the entire application since no protection boundaries are enforced.

One major type of protections to address this issue is through software-based fault isolation (SFI) which isolates untrusted libraries from a trusted host system. The original SFI technique named *sandboxing* (including data sandboxing

H. Federrath and D. Gollmann (Eds.): SEC 2015, IFIP AICT 455, pp. 386–400, 2015.
DOI: 10.1007/978-3-319-18467-8_26

and code sandboxing) was first proposed by Wahbe et al. [1]. Data sandboxing prevents sandboxed libraries from accessing memory outside a designated data region by inserting inline guards before their memory-access instructions (memory writes and memory reads). Thus both integrity and confidentiality of data in the trusted host system are protected. Code sandboxing instruments indirect-jump instructions to restrict the control flows in sandboxed libraries to a designated code region. A carefully designed and validated interface is also required when sandboxed libraries invoke the host system. Additionally, control flow integrity (CFI) [14] is a more restrictive enforcement than code sandboxing, which further guarantees that the control flows must follow a static control flow graph (CFG).

Recent work [9,15] has realized both code sandboxing and CFI with minimal performance overhead, however data sandboxing, despite of its long history(first proposed in 1993 [1]), still suffers from a perception of inefficiency which may hinder practical applications.

For avoiding high overhead, many previous approaches [2,5,10] only sandbox memory writes for integrity, but ignore protecting confidentiality due to the high cost of sandboxing memory reads that appear more frequently in code. However, in security-critical systems (e.g., military or financial systems), confidentiality is of importance as an exploited library would read secrets in the host system. NaCI-x86-32 [3] and Vx32 [4] leverage hardware segmentation to efficiently restrain memory access. However, hardware segmentation is unavailable on contemporary x86-64. As a result, NaCI-x86-64 [5] designed for x86-64 relies on instrumentation to sandbox memory writes only (or both reads and writes with significant performance overhead [12]).

Recent researches [7,8] utilize a series of performance optimizations that remove redundant instrumentation instructions, for the case of protecting both integrity and confidentiality. However, their methods are not so easy to implement correctly. For example, they need considerable efforts, which are complex and error prone, to verify the security of removed checks. Their optimizations also require static analysis (e.g., control flow analysis, register liveness analysis) and compiler-level support, which however are not compatible with existing libraries that are released as pure binaries. Additionally, due to the high frequency of memory reads in code, they incur some overhead even after optimizations, e.g., the reported overhead in Strato [8] is 32%/62%(average/max) for SFI benchmarks, and 17%/45%(average/max) for SPECint2000.

In this paper, we present ISboxing, an approach to sandbox both memory reads and writes in untrusted libraries on contemporary x86-64 platforms. Unlike previous instrumentation-based approaches, ISboxing sandboxes a memory-access instruction by directly substituting it with a *self-sandboxed* and function-equivalent one, which takes advantage of the flexibility offered by the extensive x86 instruction set.

We argue that while recent approaches focus on improving performance, they do not address the issue of practicality and hence are of limited applicability. In addition, their non-trivial overhead may still be an obstacle in some applications.

Instead, ISboxing trades some protection granularity for both practicality and efficiency. We highlight three key features which distinguish ISboxing from prior work:

- **Easy-to-implement.** ISboxing achieves data sandboxing efficiently without complex optimization work. It just needs to identify each memory-access instruction and substitute it, which is easy to implement correctly.
- **Binary-only.** ISboxing is implemented as pure binary transformation requiring no cooperation from source code or debugging symbols, and thus compatible with legacy libraries.
- **Efficient.** ISboxing incurs negligible overhead for sandboxing both memory reads and writes, since the substitution-based method requires no additional instrumentation instructions.

We have implemented ISboxing to sandbox user-space libraries on Windows. The implementation has a clear architecture comprised of a disassembler, a rewriter and a verifier. The disassembler disassembles a given library's binary executable (e.g., a PE file) and identifies memory-access instructions. The rewriter then statically substitutes each memory-access instruction. ISboxing includes a tiny verifier at the end of its workflow to validate the output of the rewriter. In this way, we can remove the disassembler and the rewriter from the TCB.

2 Related Work

SFI. Since SFI was first proposed by Wahbe et al. [1], a main portion of its following work focuses on reducing the runtime overhead of sandboxing especially on popular hardware platforms (e.g., x86). PittSFIeld [2] is the first work that applies SFI to x86 platforms which feature variable-length instructions. NaCl-x86-32 [3] and Vx32 [4] provide efficient data sandboxing relying on x86-32 segmentation. NaCl-x86-64 [5] further adapts SFI to contemporary x86-64 platforms, but it only sandboxes memory writes for performance and requires compiler-level supports. XFI [6] provides a stronger and more comprehensive protection system for executing untrusted code. Besides sandboxing, XFI provides a high-integrity stack protection model for protecting return addresses. The system of Zeng et al. [7], on the other hand, focuses on exploring a more efficient support for data sandboxing by combining CFI and static analysis. Strato [8] explores the building of a re-targetable framework for CFI and data sandboxing on a compiler intermediate representation where many optimizations can be realized without sticking to a certain hardware platform. REINS [9] is the first work that implements SFI through pure binary rewriting with trivial performance overhead, however it actually only implements code sandboxing, without discussing data sandboxing. Additionally, a series of papers [10,11] study how to enforce fine-grained SFI and memory access control. MIP [13] proposes an x86-64 SFI approach similar to ours. However, it still needs to insert additional instructions for data sandboxing, and thus it avoids sandboxing memory reads

for performance. Its method also does not address the issue of practicability, e.g., it requires complex compiler-level register liveness analysis to find scratch registers.

CFI. CFI was first introduced by Abadi et al. [14] for enforcing the integrity of a program's CFG constructed by sophisticated pointer analysis. CFI is a generic software methodology, which in theory can be applied to any systems (e.g., smartphones [17,18], and commodity OS kernels [19]). However, the construction of a precise CFG is difficult on pure binaries, and enforcing precise CFG in a program often incurs high performance overhead. For practical applications, researchers have attempted to leverage a relaxed CFI model and apply it to legacy binaries [15,16].

3 Assumptions and Attack Model

As with prior work, we assume the host system and the verifier are correct. We assume code regions and data regions in executables to be separated. The data regions are not executable and the code regions are not writable. This requirement is satisfied due to the wide deployment of W⊕X protection in modern OSes (e.g., DEP in Windows). We assume that the libraries do not self-modify their code or dynamically generate code. This requirement is always satisfied for traditional executables compiled from high-level languages. We assume that the libraries have well defined APIs which specify their parameter types and calling conventions. This is reasonable since it is necessary for the user of a library to understand how to use it. However, source code or debugging information is not required.

Two sources of attacks we consider are libraries designed by malicious authors, and libraries with bugs that could be subverted by attackers. Since untrusted libraries run in the same address space with the host system, when compromised, they would access arbitrary data and execute arbitrary code in memory. Therefore, the data integrity, data confidentiality and control flow integrity of the host system cannot be guaranteed any longer.

4 Instruction Substitution Based Data Sandboxing

4.1 Data Sandboxing Policy

ISboxing divides the address space into a protected domain and a trusted domain. ISboxing runs untrusted libraries in the protected domain to contain faults, while the host system (the main program and trusted libraries) runs in the trusted domain. Hereafter we refer to the libraries running in the protected domain as *sandboxed libraries*. The data sandboxing policy dictates that both memory reads and memory writes in sandboxed libraries must be restricted to a designated continuous data region, so that the integrity and confidentiality of the host system can be guaranteed. Hereafter we refer to this data region as *sandbox region*. In the following, we will detail ISboxing's data sandboxing approach to enforce this policy. For a better understanding, we start with the background.

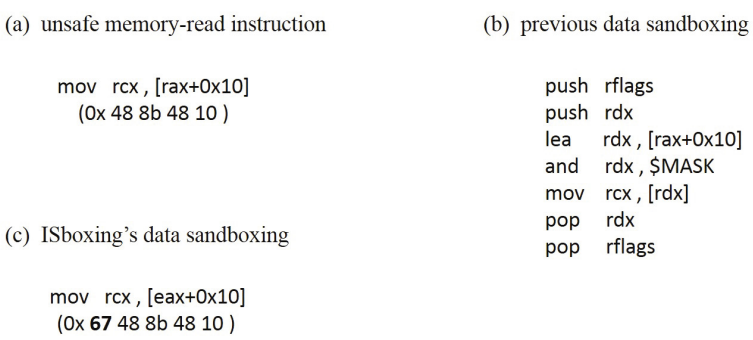

Fig. 1. An example to illustrate previous data sandboxing and ISboxing's data sandboxing

4.2 Background: x86 Memory Addressing

We first introduce the memory addressing on x86 platforms [20]. In a memory-access instruction, the address of its memory operand is referenced by means of a segment selector and an offset. In x86-64, segmentation is generally disabled, that is, the effective address of the memory operand is directly the offset. The offset can be specified as a *direct* offset which is a static value encoded in code, or an *indirect* offset through an address computation made up of one or more of the following components: displacement, base register, index register and scale. Rip-relative addressing is also available, which calculates the offset by adding a displacement to the value of current *rip* register. Data sandboxing only needs to sandbox memory-access instructions with indirect offset, because memory-access instructions with direct offset and rip-relative addressing instructions can be statically verified.

Operand Size and Address Size. On x86 platforms, the operand of an instruction has an *operand size* and an *address size* [20]. The operand size selects the size of operand and the address size determines the size of address when the operand accesses memory. In 64-bit mode, the default address size is 64 bits and the default operand size is 32 bits, but defaults can be overridden using instruction prefixes. For example, when an instruction uses the operand-size prefix (0x66) or the REX opcode prefix (0x48), its default operand size (32 bits) will be overwritten to 64 bits. When a memory-access instruction uses the address-size prefix (0x67), its default address size (64 bits) will be overwritten to 32 bits. As a result, it cannot address the memory outside 32 bits. The original motivation of these prefixes is to allow mixed 32/64-bit data and 32/64-bit addresses at instruction granularity.

4.3 Previous Data Sandboxing

In Figure 1(a), we give an example of a memory-access instruction with indirect offset (a memory read) in pseudo assembly syntax. In the example, the

instruction loads data from the memory operand (*[rax+0x10]*) to the *rcx* register. The memory operand is referenced by an indirect offset computed from a base register (the *rax* register) and a displacement (*0x10*). We show the x86 binary encoding of the instruction within the parentheses in the figure. We assume this instruction is a memory read in an untrusted library. In the following, we will illustrate how to sandbox it with previous data sandboxing.

For protecting confidentiality, the memory address represented by *[rax+0x10]* should be checked before the memory read. Figure 1(b) presents a sequence of instructions that should be inserted to perform the check. In the sequence, the register *rdx* is used as a scratch register for the check. Since the original value of *rdx* would be needed, the sequence should first save its value on the stack and restore it after the check. The *rflags* register also needs to be saved and restored in case that the *and* instruction would change its value and influence the subsequent computation. In the *and* instruction, the constant *$MASK* denotes the data-region mask that guarantees the memory read is restricted to the sandbox region.

4.4 ISboxing's Data Sandboxing

For comparison, we use the same example to illustrate how ISboxing's data sandboxing works.

Overall Idea. From another perspective, the memory-read instruction (Figure 1(a)) is unsafe because it is an "all-powerful-addressing" instruction whose address size is 64 bits. That is to say, an untrusted library can generate arbitrary value in the *rax* register, and use this instruction to access arbitrary memory in the whole 64-bit address space (the range of $[0,2^{64}]$). This observation inspires us to sandbox the memory-read instruction by changing its address size.

Without inserting any instructions, we only substitute the memory-read instruction. In the new substitute (as shown in Figure 1(c)), we only add the address-size prefix (0x67) which transforms the address size from 64 bits to 32 bits (the upper 32 bits of the address will be zero-extended by the processor). In this way, the new substitute is self-sandboxed by its address-size prefix, as if we implicitly inserted a bitwise *and* instruction whose data-region mask is 0x00000000ffffffff. Therefore, the sandbox region of ISboxing is the range of $[0,2^{32}]$, and using the new substitute to access memory outside the sandbox region becomes impossible.

Most importantly, to guarantee the correctness of the substitution, we must ensure that the new substitute is function-equivalent with the original one. To achieve this, we should ensure that the address of the memory operand is originally within the range of $[0,2^{32}]$ (the sandbox region), so that the new substitute will perform the same computation when ignoring the upper 32 bits. This requires us to relocate all the library's memory to the sandbox region, as detailed in the following sections.

Handling Stack Instructions. In x86, there is one exception that the address size of the stack is always 64 bits when stack instructions (*push*, *pop*, *call* and *ret*)

are performed to read/write data on the stack. Therefore, an untrusted library may maliciously set the stack pointer (the *rsp* register) to an address outside the sandbox region and use stack instructions to access disallowed memory. In this situation, our data sandboxing relying on address size cannot work.

We address this issue by sandboxing the *rsp* register based on instruction substitution. In x86, the operand size of each instruction determines the number of valid bits in the register (e.g., the *rsp* register): 64-bit operand size generates a 64-bit result in the register, while 32-bit operand size generates a 32-bit result, zero-extended to a 64-bit result in the register [20]. With this, we substitute each instruction modifying *rsp* (hereafter named rsp-modify instruction, e.g., *sub rsp,$0x10*) with an equivalent instruction (*sub esp,$0x10*) whose operand size is changed to 32 bits. The substitution work is easily performed by just removing the REX opcode prefix of the original instruction. Then the new substitute cannot be used to set the *rsp* register to any value outside the sandbox region, because the upper 32 bits of the *rsp* register are always zero-extended by the processor. It seems as if we implicitly inserted a bitwise *and* instruction.

Additionally, in x86, stack instructions (*push*, *pop*, *call* and *ret*) can implicitly modify the *rsp* register. Although they can only increase/decrease the *rsp* register by at most 8 bytes for each time, an attacker would chain a number of stack instructions to manipulate the *rsp* register (e.g., using ROP attacks [24,25]). We prevent this by simply inserting a guard page at the end of the sandbox region. The guard page is mapped as neither readable nor writable in the address space and thus any read/write on it will cause a page fault. Specifically, when an untrusted library uses stack instructions to modify the *rsp* register, the guard page cannot be crossed since the *rsp* register can only be increased by 8 bytes for each time. When the stack pointer reaches the guard page, executing any stack instruction again will cause a page fault and crash the untrusted library.

Constraints and Analysis. Comparing to previous data sandboxing whose sandbox region can be any size and in any position, ISboxing restricts the sandbox region to a fixed size and a fixed position in the address space. However, this is not a problem for sandboxing untrusted libraries in practice. First, the fixed size of ISboxing's sandbox region (4 GB) is large enough to contain quite a few libraries, since the virtual memory consumption of a real-world library is much smaller than 4 GB. In our observation, the virtual memory consumption of even a whole application is usually much smaller than 4 GB. Second, the constraint of the fixed position requires us to relocate libraries' memory to the sandbox region. As discussed in the next section, the memory relocation can be practically realized on libraries' executable binaries without any aids from source code or debug information.

5 Sandboxing Untrusted Libraries

Based on ISboxing's data sandboxing approach above, we further describe how to sandbox untrusted libraries on Windows. The implementation is realized through

pure binary transformation on application binaries without any special support from underlying OS.

5.1 Binary Disassembling and Rewriting

We adopt CCFIR's disassembling method [15] which can correctly disassemble an x86 binary without source code or debug information. We take advantage of the fact that ASLR and DEP are widely adopted on Windows, and leverage the relocation information to disassemble binary code recursively and identify all possible instructions. Then we can find and substitute all memory-access instructions and rsp-modify instructions to enforce ISboxing's data sandboxing.

5.2 Memory Relocation

Data sandboxing requires that the memory of sandboxed libraries must reside within the sandbox region and the memory of the host system must reside outside the sandbox region. In this way, sandboxed libraries cannot access the memory of the host system. However, in real-world applications, their memory regions are often overlapped with each other. For example, the libraries and the host system often use the same Window's API *HeapAlloc* to allocate heap memory from the same heap, and their heap memory may be overlapped. In the following, we will detail the memory relocation for sandboxed libraries. The memory relocation for the host system is essentially identical.

Executable Relocation. Due to the ASLR mechanism, a binary executable's load base can be randomized without affecting the execution. To relocate an executable, we simply disable system's ASLR for the executable and modify the *ImageBase* field in the executable's file header which specifies the load base of the executable. In addition, some code and data in the executable should also be adjusted since they are generated based on the original *ImageBase*. Though system's ASLR is disabled, existing practical and more fine-grained randomization techniques [15,26,27] can be added to ISboxing.

Stack Relocation. The host system associates a separate stack with each thread that executes in a sandboxed library. Like XFI [6], our current implementation uses a memory pool from which host-system threads draw stacks when they call the library. The stacks in this pool are all allocated in the sandbox region, and managed by a state array and a single lock. The size of the pool can be adjusted at runtime.

Heap Relocation. On Windows, applications invoke Windows APIs to allocate heap memory from the heap provided by system. The address of the heap memory is determined by system heap manager and is transparent to applications. For heap relocation, ISboxing realizes another heap manager (named ISboxing heap manager) to satisfy heap allocations for both sandboxed libraries and the host system. In our current implementation, ISboxing heap manager is a simplified version of Glibc's heap manager. It wholesales large memory chunks

from the sandbox region using Windows API *VirtualAllocEx* and provides small memory blocks for sandboxed libraries. In this way, sandboxed libraries' heap memory will always reside within the sandbox region.

File Mapping Relocation. File mapping is the association of a file's contents with a portion of the virtual memory (file-mapping memory). Fortunately, unlike heap memory, applications can use Windows API (*MapViewOfFileEx*) to specify the base address of file-mapping memory. This facilitates the realization of our file mapping relocation. ISboxing reserves a range of virtual memory in the sandbox region dedicated to file mapping for sandboxed libraries, and redirects libraries' file mapping requests to the API *MapViewOfFileEx*.

5.3 CFI Enforcement

Instead of code sandboxing, we leverage CFI, which is a more restrictive enforcement, to sandbox the control flows of untrusted libraries. The CFI enforcement, which guarantees a single stream of intended instructions, also ensures that ISboxing's data sandboxing cannot be bypassed. While CFI enforcement technique has been practically realized in CCFIR [15] with low performance overhead, we adopt CCFIR's method but use a simplified 1-ID CFI protection model. However, the code sandboxing approach discussed in REINS [9] can also be used as an alternative.

As with CCFIR, we introduce a code section called Springboard. For each legal indirect target in library code, the Springboard has an associated stub containing a *direct* jump to it. Then, we instrument each indirect jump in the library to make sure that any indirect control flow will first jump to the Springboard and then use its stubs (containing direct jumps) to complete the control flow transition. Since the Springboard only contains direct jumps whose targets are legal, the CFI is ensured. In addition, a restrictive and validated host-system interface is enforced when the libraries invoke the host system. The interface only allows three kinds of control flows (which can also be specified and validated by host-system policies): 1) Imported function calls whose targets are referenced by sandboxed libraries' import address table (IAT). 2) Function calls whose targets are resolved at runtime by special API, e.g., *GetProcAddress*. 3) Returns to the host system. The details of how to protect these control flows are well discussed in CCFIR's Section IV-D.

The CFI enforcement also inherits the deficiencies of CCFIR. For example, the relaxed CFI enforcement was shown to be broken in face of new control-flow attacks [21–23]. Nevertheless, it is enough to sandbox untrusted libraries' control flows, and is more restrictive than the code sandboxing approach in previous SFI work.

6 Implementation

In our implementation, we have developed three major tools (a disassembler, a rewriter and a verifier) to transform library binaries (PE executables) for

enforcing all ISboxing's protection on them. The implementation of the disassembler is similar to CCFIR and will not be repeated. The rewriter, whose input is the output of the disassembler, mainly performs the following work. First, the rewriter rewrites each memory-access instruction by adding the address-size prefix and each rsp-modify instruction by removing the REX opcode prefix. Second, it instruments each indirect jump and creates the Springboard section for CFI enforcement. Third, it modifies executable's file headers, relocation information and redirects heap allocation APIs to realize memory relocation. A library only needs to be rewritten once, and the rewritten binary can be shared by different applications for code reuse. In our current implementation, the rewriter takes about 3.5k LOC.

We provide a separate verifier to validate the correctness of ISboxing's protection. First, the verifier identifies all possible instructions in the rewritten binary. This is realized by scanning instructions one by one started from all indirect jump targets (in Springboard section), export table entries and the EntryPoint of the binary. For instructions not identified in this way, the CFI enforcement guarantees that they will never be executed in the runtime, because no control flows are allowed to be transferred to them. Then, for each possible instruction, the verifier checks if it conforms to the following rules: 1) If it is an indirect jump, it has been checked and redirected to the Springboard section. 2) If it is a memory-access instruction with indirect offset, its address size has been changed to 32 bits. 3) If it is an rsp-modify instruction, its operand size has been changed to 32 bits. 4) Memory-access instructions with direct offset, direct jumps and rip-relative addressing instructions are also statically validated since the code may come from a malicious source. In our current implementation, the verifier is self-contained and takes about 2.5k LOC, most of which are interpretation for x86 opcode decoding.

We have also developed a tool to transform host-system binaries (the main program and libraries) so that their memory will reside outside the sandbox region in the runtime. The tool also identifies host system's calls to the sandboxed libraries and adds a communication runtime for wrapping them (e.g., by identifying and wrapping imported function pointers and dynamically resolved function pointers). The communication runtime, which runs as a DLL in the host system, completes tasks such as copying arguments and results, switching stack and enforcing host system's policies, before transferring to the sandboxed libraries. In addition, the ISboxing heap manager used for heap relocation is developed as a dynamic library (a DLL in the host system) that satisfies heap allocations for both sandboxed libraries and the host system.

We have successfully applied ISboxing's implementation to SPECint2000, SFI benchmarks and some third-party libraries (e.g., JPEG decoder, 7-ZIP, plugins in Google Chrome), all of which are pure binaries (EXEs or DLLs) on Windows. It is worth noting that all the work is performed offline and does not influence the runtime performance.

Table 1. ISboxing's runtime overhead on SPECint2000

	CFI	RELOC	RELOC+DS	CFI+RELOC+DS
gzip	2.41%	-0.37%	0.72%	3.78%
vpr	0.01%	-0.33%	0.43%	0.33%
gcc	6.12%	-1.55%	-1.83%	5.01%
mcf	0.64%	-0.55%	-0.57%	0.01%
crafty	0.88%	-0.08%	0.12%	0.88%
parser	8.55%	-0.17%	0.14%	7.81%
eon	3.89%	-1.18%	-1.01%	3.14%
perlbmk	9.76%	-2.81%	-2.28%	7.27%
gap	4.81%	0.11%	0.00%	4.87%
vortex	5.72%	-1.70%	-1.23%	5.23%
bzip2	2.31%	-0.59%	0.95%	1.89%
twolf	0.04%	-2.49%	-3.33%	-2.85%
avg	3.76%	-0.98%	-0.66%	3.11%

Table 2. ISboxing's runtime overhead on SFI benchmarks

	CFI	RELOC	RELOC+DS	CFI+RELOC+DS
md5	0.69%	0.01%	-0.08%	0.81%
lld	0.95%	-0.21%	-0.21%	0.57%
hotlist	2.79%	-0.05%	0.19%	3.01%
avg	1.48%	-0.08%	-0.03%	1.46%

7 Evaluation

7.1 Performance Evaluation

To evaluate our implementation, we conducted experiments on a Dell Optiplex 9010 computer configured with an Intel i7-3770 (4 cores) 3.40 GHz processor, 8 GB RAM, an Intel 82579LM Gigabit Ethernet card and a Windows 7 64-bit system. We measured the execution-time overhead of sandboxing a wide range of benchmark binaries on Windows, including SPECint2000 and SFI benchmarks. SFI benchmarks contain three programs (hotlist, lld and md5) which have been widely used by previous SFI work for evaluation [8]. We treat each benchmark as if it were an untrusted library. All experiments were averaged over five runs.

Table 1 presents the execution-time percentage increases of SPECint2000, compared to the unmodified version. The standard deviation is less than 0.8 percent. The *CFI* column shows the results of CFI enforcement. CFI enforcement is necessary for data sandboxing because it ensures a single stream of intended instructions. We are pleased to see that, with CCFIR's method, the overhead (average/max) is only is 3.76%/9.76% for x86-64 binaries. The *RELOC* column reports the overhead of the memory relocation discussed in Section 5.2. As we see, ISboxing gains some performance improvement. Through analysis, this is because ISboxing uses a simplified heap manager for memory relocation in our current implementation. ISboxing's users are free to choose other heap managers

Table 3. Code-size increase on SPECint2000

	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	gap	vortex	bzip2	twolf	avg
increase (%)	5.8%	6.3%	6.5%	5.8%	6.2%	6.2%	7.1%	7.2%	7.7%	7.8%	5.6%	7.6%	6.7%

either for high performance [28] or for high level of security [29]. This performance difference can be mostly eliminated if they use a heap manager similar to Window's. The *RELOC+DS* column gives the results when both memory relocation and data sandboxing(for both memory reads and writes) are enabled, yet CFI enforcement is disabled at this time. As shown in the table, if we ignore the performance improvement of memory relocation(*RELOC+DS* minus *RELOC*), the pure data sandboxing overhead in ISboxing is only **0.32%/1.54%(average/max)** on SPECint2000. This overhead is unsurprising as ISboxing does not insert any additional instructions into the benchmarks to achieve data sandboxing. The *CFI+RELOC+DS* column shows the overhead when all ISboxing's protections are enabled.

Table 2 presents the results of SFI benchmarks. The standard deviation is less than 0.6 percent. The results show that the pure data sandboxing overhead is **0.05%/0.24%(average/max)** for SFI benchmarks.

Performance Comparison with Related Systems. We next compare ISboxing with the system of Zeng et al. [7] and Strato [8], which sandbox memory writes as well as reads with a number of optimizations to reduce runtime overhead. The pure data sandboxing overhead reported in Strato is as large as 17%/45%(average/max) for x86-64 SPECint2000 and 32%/62%(average/max) for SFI benchmarks (without adding CFI and memory relocation overhead). In Zeng et al.'s system, the pure data sandboxing overhead is 19%/42%(average/max) for x86-32 SPECint2000. It does not provide the results for x86-64, and thus the comparison is preliminary. Although other systems [2–5,13] incur low overhead, they either ignore sandboxing memory read or use hardware segmentation not available on contemporary x86 platforms.

7.2 Code-Size Increase

As a side benefit, ISboxing requires very small code-size increase to realize data sandboxing, since no additional instrumentation instructions are added to code. Table 3 presents the text-section size increase comparing to unmodified version for SPECint2000. On average, the text section grows only 6.7% for data sandboxing when CFI is not equipped.

8 Discussion

ISboxing trades some protection granularity for both efficiency and practicability. It only provides a single sandbox (the protected domain) running all untrusted libraries isolated from the host system, but cannot further provide

multiple sandboxes. Thus it cannot deal with inter-module data accesses between untrusted libraries. Nevertheless, we believe ISboxing has a wide range of applications such as isolating untrusted third-party libraries or browser plugins from the host system (trusted libraries and the main program), especially when source code is unavailable and confidentiality is of importance.

In addition, an alternative method is to sandbox only memory reads (performance critical) in a single sandbox with ISboxing, while further sandboxing memory writes (less performance critical) in multiple sandboxes using previous instrumentation methods. In this way, we can achieve a better tradeoff between performance and granularity. In real world, security secrets (e.g., secret keys) often reside in the host system, thus a single sandbox for memory reads is enough to protect their confidentiality. If a library contains security secrets, we can choose to run it in the host system.

9 Conclusion

In this paper, we present an instruction substitution based data sandboxing, which is quite different from previous instrumentation based approaches. Unlike pure software approaches, we explore how an x86 feature (instruction prefix) can help build an efficient, practical and validated data sandboxing on contemporary x86-64 platforms. We apply our approach to practically sandboxing untrusted libraries on Windows, and perform a set of experiments to demonstrate the effectiveness and efficiency.

Acknowledgments. This work has been partly supported by National NSF of China under Grant No. 61170070, 61431008, 61321491; National Key Technology R&D Program of China under Grant No. 2012BAK26B01.

References

1. Wahbe, R., Lucco, S., Anderson, T., Guaham, S.: Efficient software-based fault isolation. In: ACM Symposium on Operating Systems Principles (1993)
2. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC architecture. In: USENIX Security Symposium (2006)
3. Yee, B., Sehr, D., Dardyk, G., Chen, J., Muth, R., Orm, T., Okasaka, S., Narula, N., Fullagar, N.: Native client: a sandbox for portable, untrusted x86 native code. In: IEEE Symposium on Security and Privacy (2009)
4. Ford, B., Cox, R.: Vx32: lightweight user-level sandboxing on the x86. In: USENIX Annual Technical Conference (2008)
5. Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., Chen, B.: Adapting software fault isolation to contemporary CPU architectures. In: Usenix Security Symposium (2010)
6. Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., Necula, G.: XFI: Software guards for system address spaces. In: Symposium on Operating Systems Design and Implementation (2006)

7. Zeng, B., Tan, G., Morrisett, G.: Combining control flow integrity and static analysis for efficient and validated data sandboxing. In: ACM Conference on Computer and Communications Security (2011)
8. Zeng, B., Tan, G., Erlingsson, U.: Strato: a retargetable framework for low-level inlined-reference monitors. In: USENIX Security Symposium (2013)
9. Wartell, R., Mohan, V., Hamlen, K., Lin, Z.: Securing untrusted code via compiler-agnostic binary rewriting. In: 28th Annual Computer Security Applications Conference (2012)
10. Castro, M., Costa, M., Martin, J., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., Black, R.: Fast byte-granularity software fault isolation. In: ACM Symposium on Operating Systems Principles (2009)
11. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: Usenix Security Symposium (2009)
12. Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D., Sehr, D., Biffle, C., Yee, B.: Language-independent sandboxing of just-in-time compilation and self-modifying code. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2011)
13. Niu, B., Tan, G.: Monitor integrity protection with space efficiency and separate compilation. In: ACM Conference on Computer and Communications Security (2013)
14. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control flow integrity. In: ACM Conference on Computer and Communications Security (2005)
15. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, L., Song, D., Zou, W.: Practical control flow integrity & randomization for binary executables. In: IEEE Symposium on Security and Privacy (2013)
16. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: USENIX Security Symposium (2013)
17. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nurnberger, S., Sadeghi, A.: MoCFI: a framework to mitigate control-flow attacks on smartphones. In: Annual Network and Distributed System Security Symposium (2012)
18. Pewny, J., Holz, T.: Control-flow restrictor: compiler-based CFI for iOS. In: Annual Computer Security Applications Conference (2013)
19. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: complete control-flow integrity for commodity operating system kernels. In: IEEE Symposium on Security and Privacy (2014)
20. Intel Corporation: Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture (2013)
21. Davi, L., Sadeghi, A., Lehmann, D., Monrose, F.: Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: USENIX Security Symposium (2014)
22. Goktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: IEEE Symposium on Security and Privacy (2014)
23. Carlini, N., Wagner, D.: Rop is still dangerous: breaking modern defenses. In: USENIX Security Symposium (2014)
24. Shacham, H.: The geometry of ennocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM Conference on Computer and Communications Security (2007)
25. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: ACM Conference on Computer and Communications Security (2010)

26. Wartell, R., Mohan, V., Hamlen, K., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: ACM Conference on Computer and Communications Security (2012)
27. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.: ILR: whered my gadgets go? In: IEEE Symposium on Security and Privacy (2012)
28. Berger, E., Zorn, B., McKinley, K.: Composing high performance memory allocators. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2001)
29. Novark, G., Berger, E.: DieHarder: securing the heap. In: ACM Conference on Computer and Communications Security (2010)